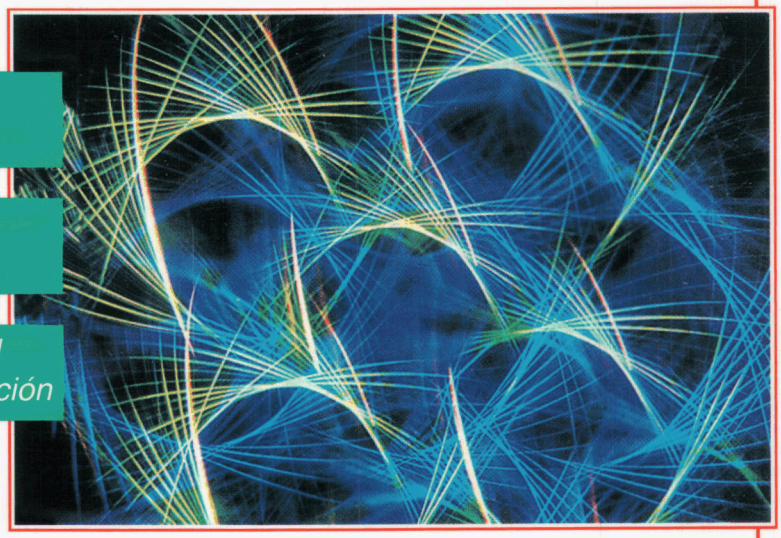
UNIX® Programación Práctica

Guía para la Concurrencia, la Comunicación y los Multihilos

- *RPC y el modelo cliente-servidor*
- *Señales de tiempo real POSIX*
- *Multihilos de control POSIX y sincronización*



Kay A. Robbins
Steven Robbins

Prentice Hall

UNIX PROGRAMACIÓN PRÁCTICA

UNIX PROGRAMACIÓN PRÁCTICA

PRIMERA EDICIÓN

Kay A. Robbins y Steven Robbins

The Univesity of Texas at San Antonio

TRADUCCIÓN:

Roberto Escalona García
M. en C., UNAM, México

REVISIÓN TÉCNICA:

Gabriel Guerrero
Doctor en Informática
Universidad de París VI

México • Argentina • Brasil • Colombia • Costa Rica • Chile • Ecuador
España • Guatemala • Panamá • Perú • Puerto Rico • Uruguay • Venezuela

EDICIÓN EN INGLÉS:

Editorial/production supervision: Jane Bonnell
Cover design director: Jerry Votta
Cover design: Lundgren Graphics, Ltd.
Manufacturing manager: Alexis R. Heydt
Acquisitions editor: Gregory G. Doench
Editorial assistant: Meg Cowen

ROBBINS: UNIX PROGRAMACIÓN PRÁCTICA, 1a. ed.

Traducido del inglés de la obra: **PRACTICAL UNIX PROGRAMMING A Guide to Currency, Communication, and Multithreading**

PRENTICE-HALL HISPANOAMERICANA, S.A.
Atlamulco No 500 5o piso
Colonia Industrial Atoto Naucalpan de Juárez
Edo de México C.P. 53519
ISBN 968-880-959-4

Miembro de la Cámara Nacional de la Industria Editorial, Reg. Núm. 1524.

Original English Language Edition Published by Prentice-Hall, Inc.
A Simon & Schuster Company

ISBN 0-13-443706-3

IMPRESO EN MÉXICO/ PRINTED IN MEXICO

Contenido

Prefacio

Los sistemas de cómputo están evolucionando con rapidez de las computadoras grandes, a las que se tiene acceso mediante terminales, hacia redes de estaciones de trabajo con varios procesadores. Ideas tales como la concurrencia, la comunicación y el uso de multihilos de control (*multithreading*) han trascendido la comunidad de investigación y se han extendido hacia el mundo comercial. El programador de aplicaciones debe comprender estos conceptos, y el presente libro está diseñado para hacerlos accesibles en todos sus detalles.

El libro utiliza un enfoque "manos a la obra" en un sentido no tradicional. En el enfoque tradicional, los programadores implementan o modifican un sistema operativo existente para añadirle funcionalidad. Este enfoque brinda una comprensión profunda del diseño básico de sistemas operativos, pero es difícil que los programadores profesionales lo sigan de manera independiente. Cuando es utilizado en una universidad promedio, los profesores invierten un tiempo considerable de clase en cubrir detalles de la implantación, lo que deja poco tiempo para abarcar una gran variedad de temas. Por otra parte, el enfoque tradicional usualmente no proporciona experiencia de programación práctica con construcciones avanzadas de sincronización y comunicación. Una alternativa es la presentación teórica. Si bien este tipo de cursos abarca mucho más material, no proporciona al lector una comprensión profunda de los conceptos en la práctica.

Programación práctica en UNIX : Una Guía de la concurrencia, la comunicación y el uso de multihilos de control llena el hueco entre los enfoques práctico y teórico al estudio de sistemas operativos al tratar la programación bajo UNIX estándar. El programador profesional puede emplear el libro de manera independiente o como complemento de un libro de referencia, tal como el *Advanced Programming in the UNIX Environment* [86] de Stevens, con lo que obtendrá un mejor conocimiento de los sistemas operativos y de la programación de sistemas. Los estudiantes pueden usar del libro como complemento de un texto tradicional, como el *Operating Systems Concepts* [80] de Silbershatz y Galvin o el *Sistemas Operativos Modernos* [92] de Tanenbaum, para aprender sistemas operativos con un enfoque práctico.

Los ejercicios y proyectos hacen que este libro sea único. De hecho, el libro se inició como un proyecto de cuaderno de ejercicios. Después del desarrollo preliminar, resultó evidente que el material necesario para realizar los proyectos se encontraba disperso en muchos lugares, a menudo en libros de referencia que proporcionaban muchos detalles pero un panorama conceptual reducido. Desde entonces, el libro ha evolucionado como una referencia autocontenida que descansa sobre los últimos estándares UNIX.

El libro está organizado en cuatro partes, las cuales contienen capítulos de tema y capítulos de proyecto. Un capítulo de tema cubre material específico y contiene muchos ejemplos y ejercicios breves de la forma "intente esto" o "qué pasa si." Los capítulos de tema terminan con una o más secciones de ejercicios. El libro proporciona ejercicios de programación para muchos conceptos fundamentales en administración de procesos, concurrencia y comunicación. Estos ejercicios satisfacen la misma necesidad que los experimentos de laboratorio en un curso tradicional de ciencias. Para comprender los conceptos es necesario utilizarlos amplia-

mente. Los ejercicios están especificados por un desarrollo por pasos y muchos pueden implantarse en menos de 100 líneas de código.

Los capítulos de proyecto integran material de varios capítulos de tema mediante el desarrollo de una aplicación extensa. Los proyectos trabajan en dos niveles. Además de ilustrar las ideas de programación, los proyectos conducen a la comprensión de un tema avanzado relacionado con una aplicación. El diseño de éstos se hace por etapas, y muchas ejecuciones completas tienen varios cientos de líneas. Dado que no es necesario escribir una cantidad grande de código, el programador puede concentrarse en la comprensión de los conceptos más que en la depuración de código. Para simplificar la programación, los autores tienen bibliotecas disponibles para comunicación por redes.

La siguiente tabla presenta un resumen de la organización del libro, 15 capítulos agrupados en cuatro partes. Los nueve capítulos de tema no dependen de los seis capítulos de proyecto, y el lector puede omitir éstos al hacer una primera lectura del libro.

Parte	Tema del capítulo	#	Proyecto del capítulo	#
I Fundamentos	Concurrencia	1		
	Proceso	2		
	Archivos	3		
			El anillo de fichas	4
II Eventos asíncronos	Señales	5		
			Temporizadores	6
			Capas cracking	7
III Concurrencia	Semáforos	8		
	Hilos POSIX	9		
	Sincronía de hilos	10		
			Máquina virtual	11
IV Comunicación	Cliente-servidor	12		
			Radio por Internet	13
	RPC	14		
			Espacio de tuplas	15

Existen muchas formas de leer el libro. Los tres capítulos de tema de la parte I son un requisito previo para el resto del libro. Los lectores pueden estudiar las partes II a la IV en cualquier orden después de leer los capítulos de tema de la parte I. La excepción es el estudio, al final de los últimos capítulos, sobre las interacciones (esto es, la forma en que los hilos de control interactúan con las señales).

Los autores *suponen que los lectores* de este libro son buenos programadores en C, aunque no necesariamente programadores en C de UNIX. El lector debe estar familiarizado con la

programación en C y las estructuras de datos básicas. El apéndice A cubre los aspectos esenciales del desarrollo de programas para lectores que no tienen experiencia en UNIX. Es probable que los programadores de UNIX conozcan ya buena parte del material presentado en los capítulos 2 y 3, pero la cobertura es bastante detallada y muchos de los proyectos finales dependen en buena medida de este material.

El lector no debe dar por sentado que la lectura de un capítulo es suficiente para comprender los conceptos, a menos que desarrolle con éxito algunos programas que hagan uso de éstos. Para un programador profesional, los ejercicios que se encuentran al final de los capítulos de tema proporcionan una introducción práctica mínima al material. En general, el profesor que haga uso de este libro para un curso de sistemas operativos puede seleccionar varios ejercicios más uno de los proyectos grandes para ejecutarlos durante el semestre. Cada proyecto tiene muchas variaciones, así que puede emplearse en varios semestres.

El libro incluye una sinopsis de muchas funciones estándares. Los estándares importantes que especifican la función aparecen en la esquina inferior derecha del cuadro de sinopsis. En general, ISO C es un subconjunto de POSIX.1 y Spec 1170. En algunos casos, la lista de archivos de encabezado requerida difiere entre estándares, donde algunos archivos de encabezado son opcionales para algunos estándares. En estos casos, el cuadro de sinopsis contiene una lista con todos los archivos de encabezado importantes.

Un libro como este nunca termina, pero teníamos que detenernos. Agradeceremos los comentarios y sugerencias de los lectores, los cuales pueden enviarlos por (correo electrónico) a `pup@vip.cs.utsa.edu`. Puede obtenerse información sobre el libro en WWW `http://vip.cs.utsa.edu/pup`. Todo el código incluido en el libro puede obtenerse de WWW o por ftp anónimo a `vip.cs.utsa.edu`, en el directorio `pub/pup`.

Agradecimientos

Nuestro reconocimiento más sincero es para Neal Wagner —amigo, colega y policía de la voz pasiva. Neal es el responsable, más que cualquier otro, de la escritura de este libro. Después de todo, argumentaba, si ya tienen los proyectos, ¿cuánto trabajo más representa escribir un libro? Después de esto, él nos proporcionó mucha ayuda —lectura y crítica una y otra vez de las versiones previas del material, lo que mejoró considerablemente el libro.

Nuestro segundo reconocimiento es para la audiencia de los más de doce cursos sobre sistemas operativos que impartimos entre 1988 y 1995, que es el periodo durante el cual se desarrolló este material. Agradecemos a los estudiantes de estos cursos por sufrir las versiones previas del libro durante el desarrollo de éste y por poner a prueba los proyectos a medida que iban surgiendo. Los errores en sus programas, sus comentarios, las quejas y sugerencias mejoraron mucho el libro y nos proporcionaron una idea más acabada de la forma en que se relacionan los temas.

También damos las gracias a muchas otras personas que leyeron el libro e hicieron sugerencias o correcciones para mejorarlo, o que nos ayudaron de otras maneras. Dennis Cadena leyó la versión preliminar e hizo muchas sugerencias. Otras personas que nos hicieron comentarios o ayudaron en otras formas son Jeff Adamek, Laura Connor, Sandy Dykes, Richard Hatch, Philip Helsel, Robert Hiromoto, Clint Jeffery, George Leach, C. Ed Nicol, Richard Rybacki, Robert Shenk, Devang Shah, Dennis Wenzel y Andrea Whitlock.

También damos las gracias a Greg Doench, nuestro editor en Prentice Hall, y a Meg Cowen, su asistente, por orientarnos en todo el proceso. Ellos nos ayudaron a mejorar el libro haciendo sugerencias y brindándonos su amable apoyo sin presionarnos. También queremos dar las gracias a Jane Bonnell, nuestro editor de producción, por toda la ayuda que nos brindó para publicar el libro. Fue un placer trabajar con estos profesionales. Muchos revisores anónimos hicieron sugerencias excelentes que nos ayudaron mucho en la revisión final del libro. La formación del libro se hizo utilizando LaTeX2$_\varepsilon$, y deseamos expresar nuestro aprecio a sus productores por proporcionar este *software* sin costo alguno.

Gracias especialmente a nuestras esposas e hijos por la paciencia que tuvieron mientras preparábamos este libro. Finalmente, queremos hacer un reconocimiento a la National Science Foundation por proporcionarnos apoyo a través del financiamiento NSF-ILI USE-0950497 para construir un laboratorio de modo que tuviéramos la oportunidad de desarrollar el currículo original en el que se basa este libro.

Parte I

Fundamentos

Capítulo 1

¿Qué es la concurrencia?

La concurrencia se refiere al hecho de compartir recursos en el mismo marco de tiempo. Esto por lo común significa que varios procesos comparten la misma CPU (esto es, son ejecutados concurrentemente) o la memoria o un dispositivo de E/S. El manejo incorrecto de la concurrencia puede dar como resultado programas que fallan sin razón aparente, incluso con la misma entrada para la que parecía que trabajaban perfectamente. Los sistemas operativos administran los recursos compartidos, y en el pasado los programadores podían permitir que el sistema manejara todos los aspectos de la concurrencia. En el caso de los programas complejos desarrollados en la actualidad, los cuales necesitan ejecutarse con eficiencia en las computadoras modernas, ya no es así. Las máquinas de escritorio con varios procesadores y los sistemas distribuidos son ejemplos de arquitecturas en las que el control de la concurrencia adquiere un significado nuevo e importante para los diseñadores de sistemas. Este capítulo presenta el tema de la concurrencia y proporciona lineamientos para la programación en sistemas UNIX en un ambiente concurrente.

La potencia de cómputo ha aumentado de manera geométrica durante casi 50 años [60] en muchas áreas, incluyendo la velocidad de cómputo, la capacidad de memoria y de almacenamiento masivo, la complejidad de los circuitos, la confiabilidad del *hardware* y el ancho de banda E/S. Este crecimiento continuó en la década pasada, junto con entubamientos (pipelines), instrucciones de ejecución traslapada en una sola CPU, la colocación de varias CPU en un escritorio, y la explosión en la conectividad de redes.

El aumento tan marcado en la comunicación y la potencia de cómputo ha iniciado cambios fundamentales en el *software* comercial. Las bases de datos grandes así como otras aplicaciones, que antes se ejecutaban en una máquina central conectada con varias terminales, ahora están distribuidas sobre máquinas más pequeñas y de menor precio. Las terminales han abierto camino a las estaciones de trabajo de escritorio, con interfaces de usuario de gráficas y capaci-

dades para multimedia. En el otro extremo del espectro, las aplicaciones de cómputo personal han evolucionado hacia el uso de comunicaciones por redes. Una hoja de cálculo ya no es un programa aislado que sólo apoya a un usuario debido a que una actualización de la hoja de cálculo puede provocar la actualización automática de otras aplicaciones ligadas con ella, por ejemplo, gráficas de datos o la realización de las proyecciones de ventas. Aplicaciones tales como la edición cooperativa, las conferencias, y los pizarrones blancos, facilitan el trabajo y la interacción entre grupos. Las tendencias de cómputo van hacia la compleja compartición de datos, la interacción en tiempo real de las aplicaciones, las interfaces de usuario inteligentes, y los flujos de datos complejos que incluyen audio y video, además de texto.

Todos estos desarrollos dependen de la comunicación y la concurrencia. La *comunicación* es el transporte de información de una entidad a otra. La *concurrencia* es la compartición de recursos en el mismo marco de tiempo. Cuando dos programas se ejecutan en el mismo sistema de modo que la ejecución de éstos está entrelazada en el tiempo, entonces ellos comparten un recurso del procesador. Los programas también pueden compartir datos, código y dispositivos. Las entidades concurrentes pueden ser hilos de control (*threads*) de ejecución dentro de programas o dentro de otros objetos abstractos. La concurrencia también puede presentarse en un sistema con una sola CPU. De hecho, uno de los trabajos más importantes de un sistema operativo moderno es administrar las operaciones concurrentes de un sistema de cómputo.

Tratar con la concurrencia no es fácil: los programas concurrentes no siempre se comportan como se espera, y los errores comunes en estos programas no siempre aparecen de manera regular. (El problema puede aparecer sólo una vez en un conjunto de un millón de ejecuciones.) No hay ningún sustituto a la experiencia práctica con estos conceptos. Este capítulo describe ejemplos de concurrencia, comenzando con la más simple de las situaciones en las que puede ocurrir. El resto del libro amplía estas ideas y proporciona ejemplos específicos.

1.1 Multiprogramación y multitarea

Los sistemas operativos administran los recursos del sistema —procesadores, memoria y dispositivos de E/S, incluyendo teclados, monitores, impresoras, ratones, disquetes, cintas, unidades de CD-ROM, e interfaces de red. La manera tan complicada en la que al parecer trabajan los sistemas operativos es una consecuencia de las características de los dispositivos periféricos, en particular de la velocidad relativa de éstos con respecto de la CPU o el procesador. La tabla 1.1 proporciona velocidades típicas, en nanosegundos, para el procesador, la memoria y los dispositivos periféricos. La tercera columna indica las mismas velocidades pero escaladas por un factor de 100 millones, lo que brinda tiempos en términos de los seres humanos. El tiempo escalado de una operación por segundo es, a grandes rasgos, la rapidez de las viejas calculadoras mecánicas de hace cincuenta años. Las velocidades citadas se encuentran en constante cambio, pero la tendencia es que las que corresponden al procesador aumenten de manera exponencial, provocando con ello un hueco cada vez mayor en el desempeño entre los procesadores y los periféricos.

Las unidades de disco han mejorado en lo que respecta a la velocidad de acceso, pero la naturaleza mecánica de éstas limita su desempeño y los tiempos de acceso no mejoran exponencialmente.

Aspecto	Tiempo	Tiempo escalado en términos humanos (100 millones de veces más lento)
Ciclo del procesador	10 ns (100 MHz)	1 segundo
Acceso al caché	30 ns	3 segundos
Acceso a la memoria	200 ns	20 segundos
Conmutación de contexto	10,000 ns (100 μs)	166 minutos
Acceso a disco	10,000,000 ns (10 ms)	11 días
Quantum	100,000,000 ns (100 ms)	116 días

Tabla 1.1: Tiempos comunes para componentes de un sistema de cómputo. Un nanosegundo (ns) es igual a 10^{-9} segundos, un microsegundo (μs) es igual a 10^{-6} segundos, y un milisegundo (ms) es igual a 10^{-3} segundos.

La disparidad entre los tiempos del procesador y los de acceso al disco continúa creciendo, hasta ahora con una tasa que es, aproximadamente, de 1 a un millón para un procesador de 100 MHz.

Un *proceso* es una instancia de un programa en ejecución. El capítulo 2 estudia el mecanismo mediante el que un programa se convierte en un proceso. El *tiempo de conmutación de contexto* (context switch) es el tiempo necesario para cambiar la ejecución de un proceso a otro. El *quantum* es, a grandes rasgos, la cantidad de tiempo de la CPU asignada al proceso antes de que éste deje que otro se ejecute. En cierto sentido, un usuario en un teclado es un dispositivo periférico. Un mecanógrafo rápido puede presionar una tecla cada 100 milisegundos. Este tiempo es del mismo orden de magnitud que el quantum de planificación de procesos, y no es coincidencia que estos números sean comparables para los sistemas interactivos de tiempo compartido.

Ejercicio 1.1

Un módem es un dispositivo que permite a la computadora comunicarse con otra sobre una línea telefónica. Las velocidades típicas de los módems son 2 400 bps, 9 600 bps, 14 400 bps y 28 800 bps, donde bps significa "bits por segundo". Suponiendo que se requieren 8 bits para transmitir un octeto (byte), estime el tiempo necesario para transmitir la cantidad necesaria de caracteres para llenar una pantalla de monitor con 25 líneas y 80 caracteres para cada velocidad de transmisión. Ahora considere una presentación gráfica formada por un arreglo de 1024 por 768 pixeles. Cada pixel tiene un valor de color que puede ser uno de 256 colores posibles. Suponga que cada valor de pixel puede transmitirse vía módem en ocho bits. Estime el tiempo necesario para transmitir la cantidad suficiente de pixeles para llenar la presentación gráfica para cada una de las velocidades de transmisión dadas anteriormente suponiendo que no se emplea ningún esquema de compresión. ¿Qué tasa de compresión se necesita para que un módem de 14.4 Kbps pueda llenar la pantalla gráfica con la misma velocidad con la que uno de 2 400 bps llena la misma pantalla pero con texto?

bps	Tiempo para texto (segundos)	Tiempo para gráficas (segundos)
2 400	6.67	2621
9 600	1.67	655
14 400	1.11	437
28 800	0.56	218

Tabla 1.2: Comparación de estimaciones del tiempo necesario para llenar una pantalla de texto y otra con gráficas, sobre un enlace por módem, con cierta velocidad.

Respuesta:

La pantalla de texto tiene $80 \times 25 = 2{,}000$ caracteres, de modo que es necesario transmitir 16 000 bits. La presentación gráfica tiene $1024 \times 768 = 786\,432$ pixeles, así que se requiere la transmisión de 6 291 456 bits. La tabla 1.2 proporciona los tiempos. ¡Se necesita una tasa de compresión mayor que 65! Las estimaciones de la tabla 1.2 no toman en cuenta la compresión o el tiempo adicional requerido por el protocolo de comunicación.

Observe en la tabla 1.1 que un proceso que lleva a cabo una operación E/S en disco no utiliza la CPU de manera eficiente: 10 nanosegundos contra 10 milisegundos, o en términos humanos, un segundo contra 11 días. Dada esta disparidad en tiempo, muchos sistemas operativos modernos hacen uso de la multiprogramación. El término *multiprogramación* significa que más de un proceso puede estar listo para su ejecución. El sistema operativo escoge uno de estos procesos. Cuando el proceso necesita esperar por un recurso (por ejemplo, que algún usuario presione una tecla o por un acceso a disco), el sistema operativo guarda toda la información necesaria para continuar el proceso donde éste se quedó y entonces selecciona otro para ejecutar. Es sencillo ver cómo trabaja lo anterior puesto que la solicitud de un recurso (como `read` o `write`) da como resultado una solicitud al sistema operativo (una llamada al sistema) y durante esa llamada el sistema puede ejecutar el código para conmutar a otro proceso.

UNIX no sólo hace multiprogramación, sino que también es de *tiempo compartido*. La idea es que parezca que el sistema operativo ejecuta varios procesos simultáneamente. Si existe sólo una CPU, entonces en cualquier momento puede ejecutarse únicamente una instrucción de un solo proceso. Puesto que la escala de tiempo humana es millones (si no es que billones) de veces más lenta que la de las computadoras modernas, el sistema operativo puede hacer la conmutación con rapidez entre varios procesos y dar la impresión de varios procesos ejecutándose al mismo tiempo.

Considérese la analogía siguiente. Supóngase que una tienda de dulces tiene varios contadores de verificación (procesos) pero solamente un verificador (la CPU). El verificador examina un artículo de un cliente (la instrucción) y entonces hace lo mismo con el artículo siguiente del mismo cliente. La verificación continúa hasta que se necesita verificar un precio (solicitud de recurso). En lugar de esperar el precio y no hacer nada, el verificador se mueve a otro contador y verifica los precios de otro cliente. El verificador (CPU) se encuentra ocupado siempre y cuando existan clientes (procesos) en espera de comprobación. Esto es la

multiprogramación. El verificador es eficiente, pero es probable que los clientes no deseen hacer sus compras en dicha tienda debido al tiempo tan grande que tienen que esperar cuando el pedido es grande y no tiene verificación de precios alguna (un proceso asociado con la CPU).

Ahora supóngase que el verificador examina artículos para un cliente durante un máximo de 30 segundos (el quantum). Cuando el verificador comienza con un cliente, también da inicio un temporizador de 30 segundos. Si el tiempo termina, el verificador se mueve hacia otro cliente, incluso si no es necesario verificar el precio. Esto es tiempo compartido. Si el verificador es lo suficientemente rápido, entonces la situación casi es equivalente a tener un verificador lento en cada caja. Ahora considere el hecho de hacer un video de tal conjunto de cajas y luego reproducirlo a una velocidad de 100 veces la velocidad normal. Lo que se vería es que el verificador atiende varios clientes al mismo tiempo.

Ejercicio 1.2

Suponga que el verificador puede comprobar un artículo por segundo, lo que corresponde en la tabla 1.1 a un ciclo de procesador de un segundo. De acuerdo con esta tabla, ¿cuál es el tiempo máximo que el verificador debe pasar con un cliente antes de atender al siguiente?

Respuesta:

El tiempo es el quantum escalado en la tabla a 116 días. Un programa puede ejecutar billones de instrucciones en un quantum —poco más del número de artículos de la dulcería que el comprador promedio adquiere.

Si el tiempo para ir de un cliente a otro (el tiempo de conmutación de contexto) es pequeño comparado con el tiempo entre conmutaciones (tiempo de ráfaga de la CPU), entonces el verificador manejará a todos los clientes de manera satisfactoria. Un problema con el tiempo compartido es el tiempo desperdiciado en la conmutación entre clientes, pero tiene la ventaja de no desperdiciar el tiempo del verificador durante una comprobación de tiempo, y los clientes con pedidos pequeños no tienen que esperar mucho con respecto de los clientes que tienen pedidos grandes.

La analogía puede ser más cercana a lo que sucede en un sistema operativo si en lugar de varios contadores de verificación sólo existe uno, con los clientes rodeándolo. Para poder hacer la conmutación del cliente A al B, el verificador guarda el contenido de la cinta de registro (el contexto) y lo restaura al estado en que se encontraba cuando procesó por última vez al cliente B. El tiempo de conmutación de contexto puede reducirse si la registradora tiene varias cintas y puede retener al mismo tiempo el contenido de los pedidos de varios clientes. De hecho algunos sistemas de cómputo tienen *hardware* especial para retener muchos contextos al mismo tiempo.

Los *sistemas con multiprocesadores* tiene varios procesadores con acceso a la memoria compartida. En la analogía del verificador para un sistema multiprocesador, cada cliente tiene una cinta de registro individual y existen varios verificadores que se encargan de trabajar en los pedidos de los clientes a los que no se ha atendido. Muchas dulcerías tienen empacadores que hacen esto.

1.2 Concurrencia a nivel de aplicaciones

La concurrencia se presenta tanto a nivel de *hardware* como de *software*. A nivel de *hardware* la concurrencia aparece porque muchos dispositivos operan al mismo tiempo, los procesadores tienen un paralelismo interno y trabajan sobre varias instrucciones simultáneamente, los sistemas tienen varios procesadores e interactúan con otros sistemas a través de redes de comunicaciones. A nivel de aplicaciones, la concurrencia aparece en el manejo de señales, en el traslape de operaciones de E/S y procesamiento, en comunicaciones, y al compartir recursos entre procesos o entre hilos de control (*threads*) del mismo proceso. Las aplicaciones descritas en este capítulo dependen mucho de estos **tipos básicos** de concurrencia. Esta sección proporciona un panorama de la concurrencia desde el nivel del *hardware*.

1.2.1 Interrupciones

La ejecución de una instrucción en un programa a *nivel de máquina convencional* es el resultado del ciclo de instrucción del procesador. En cualquier momento, el procesador ejecuta el programa cuya dirección se encuentra en el contador de programa. (Muchos procesadores tienen un paralelismo interno, como la ejecución traslapada, que reducen el tiempo de ejecución, pero el tema como es presentado, no considera esta complicación). En el nivel de máquina convencional, la concurrencia aparece porque además de ejecutar el ciclo de instrucción, el procesador controla los dispositivos periféricos. Un periférico puede generar una señal eléctrica denominada *interrupción* para activar una bandera de *hardware* dentro del procesador. La detección de una interrupción es parte del propio ciclo de instrucción. En cada ciclo de instrucción, el procesador verifica las banderas de *hardware* para saber si existen dispositivos periféricos que necesiten atención. Si el procesador detecta la presencia de una interrupción, entonces guarda el valor del contador de programa y pone uno nuevo en él, que es la dirección de una función especial conocida como *rutina de servicio de interrupción*.

Se dice que un evento es *asíncrono* para una entidad si el tiempo en que aparece no está determinado por dicha entidad. Las interrupciones generadas por los dispositivos de *hardware* externos son, en general, asíncronas con respecto de los programas que están en ejecución en el sistema. Las interrupciones no siempre se presentan en el mismo punto de la ejecución de un programa, y los resultados obtenidos por éste deben ser los mismos sin importar el momento en que sea interrumpido. En contraste, un evento como la división por cero es síncrono en el sentido de que siempre ocurre durante la ejecución de una instrucción particular si se dan a ésta los mismos datos.

Aunque la rutina de servicio de interrupción puede ser parte del programa que se interrumpe, el procesamiento de esta rutina es una entidad distinta en relación con la concurrencia debido a que las interrupciones son asíncronas con respecto del proceso. Usualmente el sistema operativo cuenta con rutinas que manejan las interrupciones generadas por los dispositivos periféricos y que se conoce como *controladores de dispositivo*. Estos controladores notifican, entonces, a los procesos importantes la ocurrencia de un evento.

Los sistemas operativos utilizan las interrupciones para implantar el tiempo compartido. Muchas máquinas cuentan con un dispositivo denominado *temporizador* (tomer) que puede

generar una interrupción después de un intervalo específico de tiempo. El sistema operativo pone en marcha al temporizador antes de cargar el contador de programa para ejecutar el programa de un usuario. Cuando el temporizador termina, genera una interrupción la cual hace que la CPU ejecute la rutina de servicio de interrupción del temporizador. Esta rutina escribe en el contador de programa la dirección del código del sistema operativo, con lo que éste toma de nuevo el control. Cuando un proceso pierde la CPU de la manera antes descrita, se dice que su quantum ha *expirado*. El sistema operativo pone el proceso en la fila de procesos que están listos para ejecutarse (ready), y donde hay procesos en espera de su turno de volver a ser ejecutado.

1.2.2 Señales

Una *señal* (signal) es la notificación por *software* de un evento. A menudo la señal es una respuesta del sistema operativo a una interrupción (un evento de *hardware*). Por ejemplo, al presionar una combinación de teclas como `ctrl-c` se genera una interrupción para el controlador de dispositivo que maneja el teclado, el cual notifica los procesos apropiados mediante el envío de una señal. El sistema operativo también puede enviar una señal a un proceso para notificarle el término de una operación de E/S o un error.

La señal *se genera* cuando ocurre el evento que la provoca. Cuando se genera una señal, el sistema operativo activa una bandera que corresponde a dicha señal para el proceso. Las señales pueden generarse de manera síncrona o asíncrona. Una señal se genera de manera síncrona si ésta es generada por el proceso o hilo de control (*thread*) que la recibe. La ejecución de una instrucción no válida o de una división por cero puede generar una señal síncrona. El `ctrl-c` de teclado genera una señal asíncrona. Las señales (capítulo 5) pueden ser empleadas para temporizadores (capítulo 6), terminación de programas (sección 7.5), control de trabajos (sección 7.7), E/S asíncronas (secciones 5.9 y 9.1.2), o vigilancia de programas (sección 5.11).

Una señal es *atrapada* si el proceso que la recibe ejecuta un manejador de ella. El programa que atrapa una señal tiene al menos dos partes concurrentes, el programa principal y el manejador de la señal. La concurrencia en potencia restringe lo que puede hacerse dentro de un manejador de señal (sección 5.6). Si el manejador modifica variables externas que el programa puede cambiar en cualquier otra parte, entonces la ejecución apropiada requiere que estas variables estén protegidas, de lo contrario pueden presentarse problemas.

1.2.3 Entrada y salida

Uno de los aspectos más importantes que deben abordar los sistemas operativos es coordinar recursos que tienen tiempos de operación muy diferentes. Tal vez sea necesario que una aplicación tenga que manejar este problema directamente para asegurar una ejecución eficiente. Mientras un programa está en espera de completar un acceso a disco, el procesador puede efectuar millones de operaciones más. En un ambiente de tiempo compartido, normalmente el sistema operativo escoge ejecutar otro proceso mientras el que se encuentra activo espera el término de la petición de E/S. El proceso mismo puede hacer otro trabajo mientras espera el uso de E/S asíncrono (secciones 5.9 y 9.1.2) o de hilos de control (*threads*) dedicados (capítulo 9) en lugar de hacer uso del bloqueo E/S ordinario. El compromiso se encuentra entre el rendi-

miento adicional y el costo extra que significa añadir más programación para el uso de la concurrencia.

Se presenta un problema adicional cuando una aplicación monitora dos o más canales de entrada, tal como la entrada que proviene de fuentes diferentes en una red. Si se emplea el bloqueo de la E/S estándar, la aplicación espera la entrada cuando inicie la lectura de una de las fuentes de datos, de modo que no será capaz de manejar la entrada de otra fuente aun cuando ésta se encuentre disponible. El capítulo 9 presenta cinco métodos para manejar el problema de fuentes de datos múltiples. Los métodos son: a) sondeo sin bloqueo E/S, b) E/S asíncrono, c) `select`, d) `poll` (el que a pesar de su nombre no utiliza el sondeo) y e) hilos de control (*threads*).

1.2.4 Hilos de control y recursos compartidos

Un método tradicional para alcanzar la ejecución concurrente en UNIX es que el usuario cree varios procesos mediante la llamada `fork` (bifurcación) al sistema. Usualmente, es necesario que los procesos coordinen su funcionamiento de alguna manera. En el caso más simple, tal vez sólo sea necesario coordinar el momento en que terminan su ejecución. Aun cuando el problema de terminación es más difícil de lo que parece ser, el capítulo 2 aborda la estructura y administración de procesos e introduce las llamadas al sistema UNIX `fork`, `exec` y `wait`. Las secciones 10.2 y 10.3 abordan aspectos más sutiles de la terminación de procesos.

Los procesos que tienen un ancestro común pueden comunicarse con un mecanismo sencillo denominado entubamiento (pipes) (capítulo 3). La comunicación entre procesos sin ancestro común puede hacerse con el empleo de señales (capítulo 5), FIFOs (sección 3.9), semáforos (sección 8.2) espacios de direcciones compartidos (capítulo 9) o mensajes (capítulo 12).

La presencia de varios hilos de control de ejecución puede proporcionar concurrencia dentro de un proceso. Cuando se ejecuta un programa, la CPU utiliza el contador del programa para determinar qué instrucción es la siguiente por ejecutar. El flujo de instrucciones resultante recibe el nombre de hilo de ejecución del programa. Éste es el flujo de control del proceso. Si dos hilos de control de ejecución distintos comparten un recurso dentro de un marco de tiempo, entonces debe tenerse cuidado que no se interfieran uno al otro. Los sistemas con varios procesadores amplían la oportunidad de aprovechar la concurrencia y la compartición entre aplicaciones y dentro de éstas. Cuando una aplicación multihilos tiene más de un hilo de ejecución concurrente activo sobre un sistema con varios procesadores, entonces pueden ejecutarse al mismo tiempo muchas instrucciones del mismo proceso.

Hasta hace poco no había estándares para el empleo de hilos de control, y los programadores se resistían a utilizarlos debido a que el paquete de hilos de control de cada vendedor era diferente. Un paquete de este tipo puede sufrir cambios con cada nueva versión del sistema operativo. Recientemente, se ha incorporado un estándar para hilos de control en el estándar POSIX.1. Los capítulo 9 y 10 estudian este nuevo estándar.

1.2.5 La red como la computadora

Otra tendencia importante es la distribución del cómputo sobre una red. La concurrencia y la comunicación se asocian para formar aplicaciones nuevas. El modelo más utilizado de cómputo distribuido es el modelo *cliente-servidor*. Las entidades básicas de este modelo son el servidor de procesos, el cual administra los recursos, y los procesos cliente que requieren el acceso a los recursos compartidos. (Un proceso puede ser tanto un servidor como un cliente.) El proceso cliente comparte un recurso enviando una solicitud al servidor. El servidor satisface la solicitud a favor del cliente y le envía una respuesta. Ejemplos de aplicaciones basadas en el modelo cliente-servidor incluyen la transferencia de archivos (`ftp`), el correo electrónico y los servidores de archivos. El desarrollo de aplicaciones cliente-servidor requiere la comprensión de la concurrencia y la comunicación. Un mecanismo natural para implantar el modelo cliente-servidor es la llamada a procedimiento remoto, o RPC, estudiada en los capítulos 14 y 15.

El modelo *basado en objetos* es otro modelo de cómputo distribuido. Cada recurso en el sistema es visto como un objeto que cuenta con una interfaz para manejar mensajes. Con esto, el acceso a todos los recursos compartidos se logra de manera uniforme. El modelo basado en objetos comparte las ventajas de la programación basada en objetos [60] en lo que respecta al desarrollo incremental controlado y a la reutilización de código. Los marcos de referencia de objeto definen las interacciones entre los módulos de código. El modelo de objetos expresa con naturalidad las nociones de protección. Muchos de los sistemas operativos distribuidos en experimentación, incluyendo Argus [62], Amoeba [91], Mach [1], Arjuna [78], Clouds [24] y Emerald [9], están basados en objetos. Los modelos basados en objetos requieren de administradores de objetos que hagan el seguimiento de éstos en el sistema.

Una alternativa al sistema operativo realmente distribuido es proporcionar capas de aplicación que se ejecuten sobre la parte superior de sistemas operativos comunes y que aprovechen el paralelismo de la red. La Parallel Virtual Machine (PVM) es un paquete de *software* [7] que permite que una colección heterogénea de estaciones de trabajo UNIX funcionen como una computadora paralela para la solución de grandes problemas computacionales. PVM administra y vigila las tareas distribuidas en varias estaciones de trabajo de la red. En el capítulo 11 se desarrolla un despachador para una versión simplificada de PVM, mientras que en el 15 se explica una implantación del espacio de tuplas para una versión simplificada de Linda, un lenguaje comercial de programación distribuida.

1.3 Estándares UNIX

Las diferencias a nivel de sistema en los distintos sistemas operativos UNIX producidos por varios vendedores provocan confusión. Este libro se apega a tres estándares importantes —ANSI C, POSIX y Spec 1170. El lenguaje C ha sido estandarizado por el American National Standards Institute (ANSI) y el International Standards Organization (ISO) [2, 41].

A nivel de sistema, el Institute of Electrical and Electronics Engineers (IEEE) ha desarrollado un grupo de estándares denominados POSIX [52, 53, 55], que son las siglas de Portable

Operating System Interface. POSIX especifica la interfaz entre el sistema operativo y el usuario de una manera estándar, de modo que los programas de aplicaciones sean transportables a través de distintas plataformas. Los estándares POSIX también son conocidos como IEEE Std. 1003. La tabla 1.3 presenta un resumen que muestra los distintos miembros del grupo de estándares POSIX. Por ejemplo, el estándar que abarca al System Application Program Interface (API) es el IEEE Std. 1003.1 o POSIX.1.

Estándar	Fecha	Descripción
POSIX.1	1990	Interfaz del sistema para programas de aplicación (API) [Lenguaje C]
POSIX.1b	1993	Rectificación 1 del API : Extensión de tiempo real [Lenguaje C] (Antes conocida como POSIX.4)
POSIX.1c	1995	Rectificación 2 del API : Extensión de hilos de control
POSIX.2	1992	Intérprete de comandos y utilerías
POSIX.3	1991	Métodos de prueba para apego a POSIX
POSIX.3.1	1992	Métodos de prueba de apego a POSIX.1
POSIX.4	1993	Ahora llamado POSIX.1b
POSIX.5	1992	POSIX.1 [Lenguaje ADA]
POSIX.6	en desarrollo	Seguridad
POSIX.7	en desarrollo	Administración del sistema
POSIX.7.1	en desarrollo	Administración de impresión
POSIX.7.2	en desarrollo	Instalación y administración de software
POSIX.7.3	en desarrollo	Administración de usuarios/grupos
POSIX.8	en desarrollo	Acceso transparente a archivos
POSIX.9	1992	POSIX.1 [Lenguaje FORTRAN]
POSIX.12	en desarrollo	Servicios de red

Tabla 1.3: Algunos estándares POSIX de interés para programadores.

Una implantación que cumple con POSIX debe apoyar el estándar base POSIX.1. Muchos de los aspectos interesantes de POSIX.1 no son parte del estándar base sino que más bien están definidos como extensiones de éste. Una implantación soporta una extensión particular si el símbolo que corresponde está definido en el archivo de encabezado `unistd.h` de la puesta en práctica. En la tabla 1.4 aparece la lista de los símbolos y sus correspondientes extensiones POSIX.

POSIX está diseñado para englobar otros sistemas operativos además de UNIX. Incluso del lado de UNIX existen complicaciones. Existen dos grandes familias de ejecuciones de UNIX: System V y BSD. System V evolucionó del UNIX original de AT&T. BSD UNIX fue desarrollado en la University of California, en Berkeley, con la finalidad de proporcionar un sistema abierto con soporte complejo para red. (Véase Leffler *et al.* [59] para una perspectiva histórica). Parece que muchas versiones comerciales de UNIX están estandarizadas con res-

Símbolo	Extensión POSIX.1
_POSIX_ASYNCHRONOUS_IO	Entrada y salida asíncronas
_POSIX_FSYNC	Sincronización de archivos
_POSIX_JOB_CONTROL	Control de trabajos
_POSIX_MAPPED_FILES	Archivos mapeados en memoria
_POSIX_MEMLOCK	Bloqueo de la memoria del proceso
_POSIX_MEMLOCK_RANGE	Bloque de rango de memoria
_POSIX_MEMORY_PROTECTION	Protección de memoria
_POSIX_MESSAGE_PASSING	Paso de mensajes
_POSIX_PRIORITIZED_IO	Entrada y salida priorizada
_POSIX_PRIORITY_SCHEDULING	Planificación del procesamiento
_POSIX_REALTIME_SIGNALS	Señales de tiempo real
_POSIX_SAVED_IDS	El proceso ha guardado un ID especificado por el usuario
_POSIX_SEMAPHORES	Semáforos
_POSIX_SHARED_MEMORY_OBJECTS	Objetos de memoria compartida
_POSIX_SYNCHRONIZED_IO	Entrada y salida sincronizada
_POSIX_TIMERS	Temporizadores
_POSIX_VERSION	199309L para apego a implantaciones

Tabla 1.4: Constantes simbólicas de POSIX.1 al momento de la compilación. Si la constante está definida en `unistd.h`, entonces el sistema soporta la extensión POSIX.1 correspondiente.

pecto de la versión 4 del System V, aunque está surgiendo un estándar denominado Spec 1170 desarrollado por la X/Open Foundation [96, 97, 98, 99]. Este libro hace hincapié en POSIX, Spec 1170 y en la versión 4 de System V. Para aspectos tales como los servicios de red que no están cubiertos por el estándar POSIX, el libro sigue el Spec 1170. Para los que no están cubiertos por ningún estándar, el libro utiliza la versión 4 del System V.

1.4 Programación en UNIX

Una manera de aprender la concurrencia es implantando programas en un lenguaje diseñado específicamente para dar soporte a la concurrencia. En lugar de hacer esto, el presente libro utiliza C estándar y programación en UNIX para explorar las ideas. En esta sección se proporciona, de manera breve, varios principios para el desarrollo de programas sistema bajo UNIX. En el apéndice A se repasa algunos aspectos básicos de UNIX para programadores de sistemas.

Los programas solicitan servicios al sistema operativo, como los de entrada y salida, al invocar una *llamada al sistema* apropiada. La llamada al sistema es un punto de entrada directa al núcleo. El *núcleo* (o *kernel*) es una colección de módulos de *software* que se ejecutan en

forma *privilegiada* –lo que significa que tiene acceso pleno a los recursos del sistema. Las llamadas al sistema son una parte intrínseca del núcleo del sistema operativo. (En ocasiones los términos núcleo y sistema operativo) se emplean en forma alternada.

También están disponibles muchas funciones de propósito general de la biblioteca de C. La funciones de esta biblioteca pueden o no hacer uso de llamadas al sistema. No siempre es posible determinar, a partir de la definición de éstas, si una función es una función de la biblioteca o una llamada al sistema. Tradicionalmente las llamadas al sistema están descritas en la sección 2 del manual de UNIX, mientras que las de la biblioteca de funciones aparecen en la sección 3. La versión en línea del manual de UNIX recibe el nombre de páginas del manual (*man pages*). (El apéndice A.1 comenta sobre éstas.)

Cuando se utiliza una llamada al sistema o a la biblioteca, asegúrese de consultar las páginas del manual de UNIX para encontrar el encabezado o prototipo apropiado para la llamada. Estas páginas presentan un resumen que identifica los archivos de encabezado que deben ser incluidos en un programa cuando éste hace uso de una llamada al sistema o de una función de la biblioteca.

Ejemplo 1.1

El resumen de las páginas del manual para la llamada al sistema `open` *menciona tres archivos de encabezado.*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open (const char *path, int oflag, /* mode_t mode */
     ...);
```

Los archivos de encabezado del ejemplo 1.1 contienen el prototipo de `open` así como las definiciones de varias banderas que pueden ser empleadas como parámetro de la llamada. Asegúrese de incluir estos archivos de encabezado en cualquier programa que utilice `open`.

¿Qué sucede cuando una llamada al sistema o a la biblioteca encuentra un error? Usualmente la ejecución del programa no termina de manera abrupta, al menos no directamente. En lugar de esto la llamada devuelve un valor que indica una condición de error, y es responsabilidad del programa examinar el error y manejarlo. En general, el valor devuelto para el error es negativo si el tipo de regreso de la llamada es `int` o `NULL` si es de tipo apuntador.

Las páginas del manual establecen que `open` devuelve un descriptor de archivo entero si se tiene éxito. Si existe un error, `open` devuelve –1 y pone en la variable `errno` un código de `errno` apropiado. El sistema llama y muchas funciones Library (biblioteca) regresan a –1 para indicar un error. El valor de la variable `errno` no necesariamente cambia cuando se hace una llamada subsecuente al sistema o a la bliblioteca, a menos que ésta genere otro error. Las páginas del manual especifican nombres simbólicos para los distintos errores, y el programa puede comparar `errno` con estos valores para determinar el tipo de error que ha ocurrido. Para ello es necesario incluir el archivo de encabezado `errno.h` para tener acceso a los nombres simbólicos de los errores asociados con el valor de `errno`.

Ejemplo 1.2

El siguiente fragmento de código en C abre el archivo `my.file` *para lectura.*

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

if ((fd = open ("my.file", O_RDONLY)) == -1)
perror ("Unsuccessful open of my.file");
```

La función de la biblioteca de C `perror` muestra un mensaje con el error estándar seguido por el de la última llamada al sistema o la biblioteca que lo produjo. (El último mensaje de error tal vez no tenga nada que ver con la última llamada si la anterior no produjo un error.)

SINOPSIS

```
#include <stdio.h>

void perror (const char *s);
```

ISO C, POSIX.1, Spec 1170

Ejemplo 1.3

El ejemplo 1.2 puede producir el siguiente mensaje de salida si `my.file` *no existe.*

```
Unsuccesfull open of my.file: No such file or directory
```

La llamada a `open` puede fallar por varias razones, y las páginas del manual proporcionan una lista de muchos nombres de error simbólicos asociados con `open`. Uno de los nombres de error es `EAGAIN`, lo que indica que el archivo está bloqueado.

Ejemplo 1.4

El siguiente ciclo vuelve a intentar `open` *si la llamada no tiene éxito debido a que el archivo está bloqueado.*

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
int fd;

while (((fd = open ("my.file", O_RDONLY)) == -1)&&(errno == EAGAIN))
if (fd == -1)
perror ("Unsuccessful open of my.file");
```

Recuérdese que es necesario probar `errno` sólo si la llamada devuelve un error. La prueba del ejemplo 1.4 funciona debido a que C garantiza que el segundo operador de `&&` no sea evaluado a menos que el primero sea verdadero.

La función de C `strerror` devuelve un apuntador a un mensaje de error.

SINOPSIS

```
#include <string.h>

char *strerror (int errnum);
```

ISO C, POSIX.1, Spec 1170

La variable `errnum` contiene el valor de `errno` asociado con el mensaje de error. Para dar formato al mensaje que contiene los valores de las variables utilícese `strerror` en lugar de `perror`.

Ejemplo 1.5

El siguiente fragmento de código en C utiliza `strerror` *para imprimir un mensaje de error que depende del sistema.*

```
#include <string.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
int fd;

if ((fd = open(argv[1] O_RDONLY) == -1)
    fprintf(stderr, "Could not open file %s: %s\n"
       argv[1], strerror(errno));
```

El nombre del archivo del ejemplo 1.5 se toma del primer argumento de la línea comando, `argv[1]`. El mensaje de error incluye el nombre del archivo así como el mensaje de error del sistema.

Muchas llamadas al sistema UNIX o a las funciones de la biblioteca de C proporcionan buenos modelos para la implantación de funciones. Las siguientes son recomendaciones que hay que seguir para hacer lo anterior:

- Haga uso de los valores de regreso (return values) para comunicar información y facilitar el seguimiento de errores por parte del programa que hace la llamada.
- No salga de las funciones. En lugar de hacerlo, devuelva un valor de error que permita al programa que hizo la llamada manejar el error de manera flexible.
- Construya funciones generales pero utilizables.(En ocasiones, éstas son metas que se contraponen).
- No haga hipótesis innecesarias sobre el tamaño de los *buffers*. (A menudo, lo anterior es difícil de establecer).

- Cuando sea necesario utilizar límites, emplee los definidos por el sistema más que constantes arbitrarias.
- No "reinvente la rueda"—cuando sea posible haga uso de las funciones de la biblioteca.
- No modifique los valores de los parámetros de entrada, a menos que tenga sentido hacerlo.
- No utilice variables estáticas o la asignación dinámica de memoria si la asignación automática trabaja bien.
- Analice todas las llamadas a la familia de funciones `malloc` para asegurar que el programa libere toda la memoria asignada a él.
- Considere la forma en que la función será llamada, recursivamente, por un manejador de señal o por un hilo. Las funciones reentrantes no pueden automodificarse, de modo que pueden existir varias invocaciones activas de ellas al mismo tiempo sin que interfieran unas a otras. En contraste, las funciones con variables locales estáticas o externas no son reentrantes y tal vez no se comporten de la manera deseada cuando sean llamadas de manera recursiva. (En este caso, `errno` causa un gran problema).
- Analice las consecuencias de las interrupciones generadas por señales.
- Considere de manera cuidadosa la forma en que terminará la ejecución del programa.

Como ilustración de estos principios considérese el problema de construir arreglos de argumentos similares a los empleados para pasar a los programas en C, los argumentos de la línea comando. Los arreglos de argumentos son arreglos de apuntadores a cadenas de caracteres.

En UNIX una línea comando está formada por unidades sintácticas (los argumentos) que están separadas por espacios en blanco, tabuladores o diagonales invertidas (\), justo antes de un carácter de línea nueva. Cada unidad sintáctica es una cadena de caracteres. Cuando un usuario introduce un comando que corresponde a un programa ejecutable en C, el intérprete de comandos (*shell*) analiza la línea comando y pasa el resultado al programa como un arreglo de argumentos. (Más adelante aparecen ejercicios y proyectos que utilizan arreglos de argumentos cuando es necesario crear procesos que lleven a cabo varias tareas).

Ejemplo 1.6

La siguiente línea comando contiene cuatro unidades sintácticas: `mine, -c, 10` *y* `2.0.`

```
mine -c 10 2.0
```

La primera unidad sintáctica de la línea comando es el nombre de un comando o un archivo ejecutable. La figura 1.1 muestra el arreglo de argumentos de la línea comando del ejemplo 1.6.

Ejemplo 1.7

El programa principal `mine` *al que hace referencia el ejemplo 1.6 puede comenzar con la siguiente línea.*

```
int main (int argc, char *argv[])
```

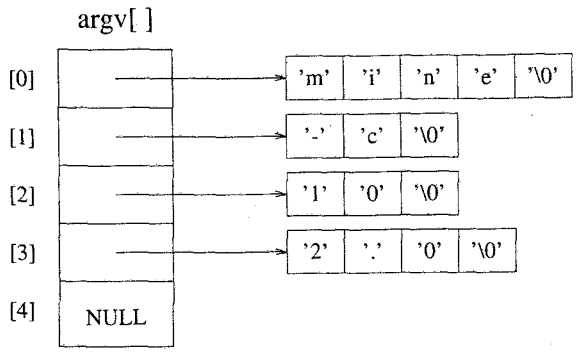


Figura 1.1 : Arreglo `argv` para la llamada `mine -c 10 2.0`.

En el ejemplo 1.7, el parámetro `argc` contiene el número de unidades sintácticas de la línea comando (cuatro para el ejemplo 1.6), y `argv` es un arreglo de apuntadores a éstas. Los arreglos de argumentos son útiles para manejar un número variable de argumentos en las llamadas a `execvp`. (Véase la sección 2.7 donde se muestra un ejemplo de su aplicación).

Ejemplo 1.8

El siguiente prototipo para la función `makeargv` *indica que ésta crea un arreglo de argumentos a partir de una cadena de fichas (tokens).*

```
char **makeargv(char *s) ;
```

La función `makeargv` del ejemplo 1.8 tiene como parámetro de entrada una cadena de caracteres y devuelve un apuntador a un arreglo `argv`. Si la llamada no tiene éxito, entonces la función devuelve el apuntador `NULL`.

Ejemplo 1.9

El siguiente fragmento de código ilustra la forma en que puede invocarse la función `makeargv` *del ejemplo 1.8.*

```
#include <stdio.h>
int i;
char **myargv;
char mytest[] = "This is a test";

if (myargv = makeargv(mytest)) == NULL)
    fprintf(stderr, "Could not construct an argument array\n");
else
    for (i = 0; myargv[i] != NULL; i++)
        printf("%i: %s\n", i myargv [i]);
```

Ejemplo 1.10

La siguiente alternativa de prototipo indica que `makeargv` *debe pasar el arreglo de argumentos como un parámetro. La versión alternativa de* `makeargv` *devuelve un entero que indica el número de unidades sintácticas presentes en la cadena de caracteres proporcionada como entrada. En este caso, los errores están señalados por valores negativos diferentes.*

```
int makeargv(char *s, char ***argvp);
```

Ejemplo 1.11

Este segmento de código hace una llamada a la función `makeargv` *definida en el ejemplo 1.10.*

```
#include <stdio.h>
int i;
char **myargv;
char mytest[] = "This is a test";
int numtokens;

if ((numtokens = makeargv(mytest, &myargv)) < 0)
   fprintf(stderr, "Could not construct an argument array\n");
else
   for (i = 0; i < numtokens; i++)
      printf("%i: %s\n", i myargv [i]);
```

El ejemplo 1.11 ilustra más de un nivel de indirección (*) cuando se pasa la dirección de `myargv` debido a que C utiliza un mecanismo de paso de parámetros por valor. Una versión más general de `makeargv` permite un parámetro adicional que representa el conjunto de delimitadores que deben emplearse para analizar sintácticamente la cadena de caracteres.

Ejemplo 1.12

Este prototipo muestra una función `makeargv` *que tiene como parámetro un conjunto de delimitadores.*

```
int makeargv(char *s, char *delims, char***argvp) ;
```

El programa 1.1 llama a la función `makeargv` del ejemplo 1.12 para crear un arreglo de argumentos a partir de una cadena de caracteres pasados a través de la línea comando. El programa verifica que exista sólo un argumento en la línea comando, de lo contrario imprime un mensaje. Si `makeargv` encuentra un error, entonces su ejecución no termina, sino que devuelve un error a la función que la llamó. Con esto el programa principal llama a `exit(1)` si es que falla, o a `exit(0)` si termina de manera exitosa. La llamada a `makeargv` utiliza como delimitadores caracteres en blanco y tabuladores. El intérprete de comandos emplea los mismos delimitadores, pero el usuario debe tener cuidado de encerrar la cadena de caracteres entre comillas dobles.

Programa 1.1 : El programa `argtest` toma como argumento una línea comando, el cual es una cadena de caracteres, y llama a `makeargv` para crear un arreglo de argumentos.

```
#include <stdio.h>
#include <stdlib.h>
int myargvi(char *s, char *delimiters, char ***argvp);

void main(int argc, char *argv[])
{
   char **myargv;
   char delim[] = " \t";
   int i;
   int numtokens;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s string\n", argv[0]);
      exit(1);
   }
   if ((numtokens = makeargv(argv[1], delim, &myargv)) < 0) {
      fprintf(stderr,
         "Could not construct an argument array for %s\n", argv[1]);
      exit(1);
   } else {
      printf("The argument array contains:\n");
      for (i = 0; i < numtokens; i++)
          printf("[%d]:%s\n", i myargv[i]);
   }
   exit(0);
}
```

Programa 1.1

Ejemplo 1.13

Si el archivo ejecutable del programa 1.1 se llama `argtest`*, entonces el siguiente comando crea e imprime un arreglo de argumentos para* `This is a test`.

```
argtest "This is a test"
```

La implantación de `makeargv`, aquí presentada, utiliza la función de la biblioteca de C `strtok` para separar la cadena de caracteres en fichas (token).

SINOPSIS

```
#include <string.h>

char *strtok (char *s1, const char *s2);
```

ISO C, POSIX.1, Spec 1170

En la primera llamada a `strtok` se emplea la dirección inicial de la cadena de caracteres para rastrear a `s1`, y un `NULL` para buscar a `s1` en las demás llamadas. Las llamadas subsecuentes devuelven el inicio de la siguiente ficha y ponen un `'\0'` al final de ésta. La cadena `s2` contiene los delimitadores permitidos. La función `strtok` devuelve `NULL` cuando llega al final de `s1`.

Es importante no imponer ninguna limitación a priori innecesaria sobre el tamaño del arreglo mediante el empleo de *buffers* de tamaños preestablecidos. Aunque la constante `MAX_CANON` definida por el sistema proporciona un tamaño de *buffer* aceptable para el manejo de los argumentos de la línea comando, la función `makeargv` puede emplearse para hacer una lista de las variables de ambiente o por aplicaciones que toman su entrada de un archivo. Esta implantación de `makeargv` asigna todos los *buffers* de manera dinámica mediante llamadas a `calloc`. Para preservar la cadena de entrada `s`, `makeargv` no aplica `strtok` directamente a `s`, sino que crea un área de trabajo del mismo tamaño a la que apunta `t`, copiando a `s` en ella. La estrategia global es

- Asignar el espacio para las unidades sintácticas en lugar de modificar a `s`. Más que hacer una llamada a `calloc` para cada unidad, lo que se hace es asignar un nueva cadena `t` que tiene el mismo tamaño que la cadena `s` original, tal como se muestra en la figura 1.2.
- Hacer un análisis de la cadena `t` utilizando `strtok` para contar las fichas.
- Utilizar el conteo (`numtokens`) para crear un arreglo `argv`.
- Emplear `strtok` para obtener los apuntadores a cada una de las fichas, modificando `t` en el proceso. (La figura 1.3 muestra el método para detectar las fichas).

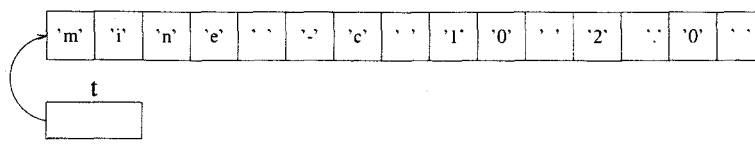


Figura 1.2 : La función `makeargv` hace una copia de trabajo de la cadena `s`, de modo que no modifique dicho parámetro de entrada.

El programa 1.2 ilustra una implantación de `makeargv`. Puesto que `strtok` permite a quien la llama especificar los delimitadores que debe utilizar para separar las fichas, la implantación incluye la cadena `delimitadores` como parámetro. El programa comienza utilizando a `strspn` para hacer caso omiso de los delimitadores. Esto garantiza que `**argvp`, el cual apunta a la primera ficha, también apunte al inicio del *buffer* temporal, el cual se llama `t` en el programa. Si ocurre un error, este *buffer* se libera de manera explícita. De lo contrario, puede liberarlo el programa que invoca la función. Es probable que la función `free` no sea importante en muchos programas, pero si `makeargv` se llama con frecuencia desde un intéprete de comandos o en un programa de comunicación de ejecución constante, entonces es probable que se acumule el espacio no liberado por varias llamadas sin éxito a `makeargv`. Cuando se emplea `calloc` o alguna otra función similar, es necesario analizar cuándo liberar la memoria si se presenta un error o al término de la función.

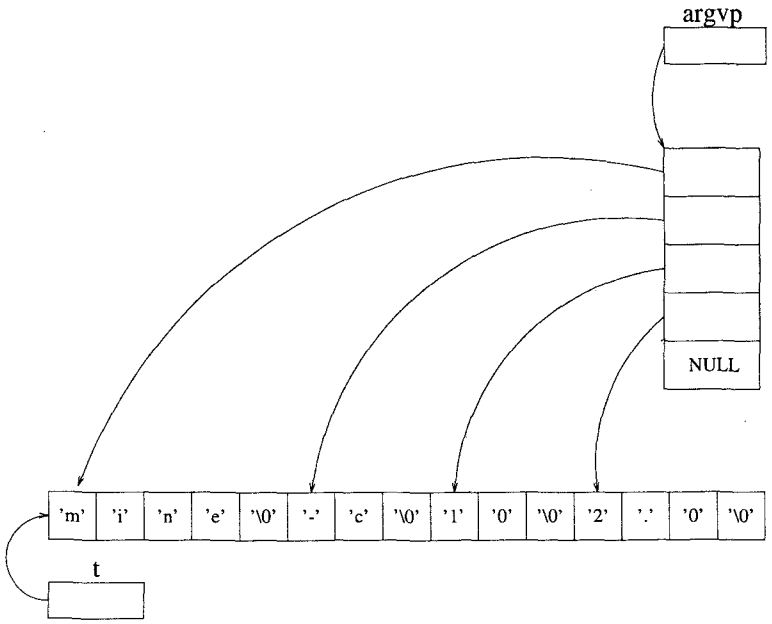


Figura 1.3 : Uso de `strtok` para localizar cadenas en `makeargv`.

Programa 1.2 : Implantación de `makeargv`

```
#include <string.h>
#include <stdlib.h>
/*
 *Se construye el arreglo argv (*arvp) para las fichas en s que
  están separadas por
 *delimitadores. La función devuelve -1 si hay un error o, de lo
  contrario, el número de unidades sintácticas.
 */
int makeargv(char *s, char *delimiters, char ***argvp);
{
        char *t;
        char *snew;
        int numtokens;
        int i;
     /* snew es el inicio real de la cadena después de omitir los
        delimitadores */
   snew = s + strspn (s, delimiters);
                /* se crea el espacio para una copia de snew en t */
        if ((t = calloc(strlen(snew) + 1, sizeof(char))) == NULL) {
            *argvp = NULL;
            numtokens = -1;
```

```
} else {                /*conteo del número de fichas en snew */
    strcpy(t, snew);
    if (strtok(t, delimiters) == NULL)
       numtokens = 0;
    else
      for (numtokens = 1; strtok(NULL, delimiters) != NULL;
           numtokens++)
           ;
             /* se crea un arreglo de argumentos que contiene
                apuntadores (ptr) a las fichas */
if ((*argvp = calloc(numtokens + 1, sizeof(char *))) == NULL) {
    free(t);
    numtokens = -1;
} else {   /* en el arreglo se insertan apuntadores a las fichas */
    if (numtokens > 0) {
        strcpy(t, snew);
        **argvp = strtok(t, delimiters);
        for (i = 1; i < numtokens + 1; i++)
            *((*argvp) + i) = strtok(NULL, delimiters);
    } else {
      **argvp = NULL;
      free(t);
    }
  }
}
return numtokens;
}
```

Programa 1.2

1.5 Cómo hacer que las funciones sean seguras

La función `strtok` es un buen ejemplo de una función que no debe emularse, ya que no es reentrante. Una llamada a `strtok` puede modificar el comportamiento de otras llamadas subsecuentes debido a que la función mantiene un registro de su posición en la cadena de caracteres entre llamadas sucesivas. El uso seguro de la función `strtok` en un ambiente concurrente, ya sea con señales o hilos de control, no es directo.

Supóngase que un programa hace uso de `strtok` en un manejador de señal. Si la señal aparece entre llamadas sucesivas a `strtok` hechas en cualquier parte del programa, entonces `strtok`, se confundirá. La rutina utilizará la posición que tenía en la última llamada hecha por el `strtok` de señal, más que por la efectuada en la parte principal del programa.

Recuérdese que las señales pueden ser asíncronas, esto es, no es posible determinar con anticipación el momento en el que ocurrirán. Si la señal es atrapada antes de la primera o después de la última llamada a `strtok` en la parte principal del programa, todo marcha bien. Sin embargo, si la señal es atrapada entre llamadas a `strtok`, entonces el programa puede fallar. El mismo programa no tiene ningún control sobre este comportamiento y tal vez funcione el

99.99 por ciento de las veces. Cuando se emplean hilos de control aparece un problema similar.

Existen dos aspectos por considerar para lograr que las funciones sean seguras. Una función se conoce como *segura con respecto de hilos de control* si puede ser invocada con seguridad en forma concurrente por varios hilos de control. Una función es segura con respecto de señales asíncronas si un manejador de señal puede llamarla sin ninguna restricción. Estos términos reemplazan la vieja noción de *función reentrante*.

El estándar POSIX.1c [54] incluye hilos de control así como las modificaciones necesarias al estándar base para dar apoyo a la ejecución correcta de éste. La meta es hacer que muchas de las funciones definidas por POSIX sean seguras con respecto de los hilos de control. La seguridad con respecto de señales asíncronas es más difícil, y la tabla 5.3 proporciona una lista completa de las funciones que están garantizadas para este tipo de seguridad, de acuerdo con el estándar POSIX.1 adoptado [52]. El apego a los estándares POSIX no es completo, incluso en sistemas operativos que aseguran dicho apego. Consúltense el manual para determinar la seguridad de una función dada en una versión determinada de un sistema operativo.

En la revisión del estándar POSIX.1 todas las funciones, con excepción de unas cuantas, son seguras en cuanto a los hilos de control. Algunas funciones (como `strtok`) están más allá de cualquier esperanza debido a que su definición garantiza que son no reentrantes. Para cada una de ellas, POSIX define una función sustituto que tiene como sufijo `_r`. La versión segura con respecto de hilos de control de `strtok` es `strtok_r`.

RESUMEN

```
#include <string.h>

char *strtok_r(char *s, const char *sep, char **lasts);
```

POSIX.1c

Esta función tiene un comportamiento similar a `strtok`, con la excepción de que acepta el parámetro `lasts`. Este parámetro es apuntador proporcionado por el usuario que apunta a una localidad que será utilizada para conservar la posición en la cadena necesaria para la siguiente llamada a `strtok_r`.

Es posible que todas las versiones no reentrantes de otras funciones similares eventualmente sean eliminadas del estándar. Evítese el uso de éstas hasta donde sea posible. Desafortunadamente, muchas de las llamadas estándares al sistema así como las funciones de la biblioteca de C son no reentrantes por definición, ya que ellas devuelven información sobre errores en una variable externa, `errno`. El comité de estándares POSIX decidió que todas las funciones nuevas no hagan uso de `errno` y que en su lugar devuelvan directamente un número de error como valor de retorno de la función. Sin embargo, sigue existiendo el problema de qué hacer con las funciones que ya existen. Considérese `read`, la cual devuelve el número de octetos leídos o `-1` si se presenta un error. Si `read` devuelve `-1`, esto hace que `errno` indique el tipo de error.

RESUMEN

```
#include <unistd.h>

ssize_t read(int filedes, void *buf, size_t nbyte);
```

POSIX.1, Spec 1170

Las siguientes son varias alternativas para el manejo de errores, algunas de las cuales fueron consideradas por el comité de estándares.

- En lugar de `read`, utilice lo siguiente:

```
in read_r(int filedes, void *buf, size_t nbyte,
          ssize_t *bytesread);
```

  la cual devuelve un número de error ó 0 si éstos no existen. El último parámetro proporciona un lugar donde almacenar el número de octetos leídos.
- En lugar de `read` utilice lo siguiente:

```
ssize_t read(int filedes, void *buf, size_t nbyte,
             int *status);
```

  donde se utiliza el último parámetro para guardar un número de error.
- Implantar `errno` como una variable local de cada hilo.
- Implantar `errno` como un servicio que puede tener acceso a un número de error por hilo. Esto significa que `errno` podría ser un macro que invoque a una función para obtener el número de error del hilo que está bajo ejecución.
- Modificar `read` de modo que genere una excepción del lenguaje (una señal) si ocurre un error.

Todos estos enfoques plantean problemas serios. Los dos primeros y el último requieren que los programas se escriban otra vez para hacer uso de la nueva función `read`. La tercera opción preserva la compatibilidad, pero requiere de un soporte especial del enlazador (*linker*), el compilador y el sistema de memoria virtual. La cuarta opción preserva la compatibilidad a nivel de código fuente, puesto que el macro puede definirse en `errno.h`, el cual es incluido cada vez que se emplea `errno`. Desafortunadamente, los programas de aplicación que asignan de manera explícita un valor a `errno` tendrán que volver a ser escritos si `errno` es un macro. Las opciones tercera y cuarta hacen la función segura con respecto de hilos de control, pero siguen causando problemas si la función es llamada desde el interior de un manejador de señal. La quinta opción requiere que el programa instale manejadores de señal o que dé por terminada su ejecución.

1.6 Ejercicio: arreglos de argumento

- Hacer una prueba de la función `makeargv` definida en el programa 1.2 mediante el empleo de la rutina principal del programa 1.1. Ponga la función `makeargv` en un archivo aparte, cuyo nombre sea `argvlib.c`. Utilice un archivo de instrucciones (`makefile`) para la construcción de ejecutables junto con la utilería `make` para compilar los programas. (Consulte el apéndice A.3, el cual contiene una introducción breve a `make`.) Pruebe la función con varias entradas.
- Escribir una función `freeargv` para el archivo `argvlib.c` que tenga el prototipo siguiente:

```
void freeargv(char ***argvp) ;
```

La función `freeargv` libera el arreglo de argumentos al que apunta `*argvp` y hace que `*argvp` sea igual a `NULL`.

- Escribir una página de manual completa para la función `makeargv`. (Consulte el apéndice A.1 para una presentación breve de este tema).
- Escribir una versión nueva de `strtok`, llamado `estrtok`, que trate la ocurrencia de delimitadores consecutivos como una ficha vacía y no como la ausencia de ésta. El prototipo para `estrtok` es

```
char *estrtok(char *s1, const char *s2);
```

El intérprete de comandos puede utilizar la función `estrtok` para analizar la variable de ambiente `PATH`. Esta variable consta de prefijos de ruta de acceso separados por dos puntos (`:`). El significado asociado con un prefijo vacío es el directorio de trabajo en uso. Dos puntos solos equivalen a un prefijo de ruta de acceso vacío. Cuando se ejecuta un comando que no comienza con un `'/'`, el intérprete buscará el archivo ejecutable añadiendo el nombre de éste a cada prefijo de ruta de acceso, hasta que encuentre el archivo. Por ejemplo, si el intérprete ejecuta el comando `ls` y la variable `PATH` tiene el valor `/bin :/usr/bin :/usr/local/bin : :`, el intérprete intentará localizar el ejecutable buscando sucesivamente en `/bin/ls`, `/usr/bin/ls`, `/usr/local/bin/ls` y `./ls`. La búsqueda termina cuando se encuentra el archivo o cuando se agota la lista de posibilidades.

- Volver a escribir la función `makeargv` del programa 1.2 de modo que ahora ésta tenga como prototipo

```
in makeargv(char *s, char *delimiters, int flags,
            char ***argvp);
```

La variable `flags` especifica opciones de conversión. Para `makeargv`, defina las opciones siguientes en un archivo de inclusión

```
#define EMPTYTOKENS 1 /* Token for consecutive delims */
#define NOTOKENS 0 /* No Token for consecutive delims */
```

Cuando `flags` es igual a `EMPTYTOKENS`, entonces una ficha vacía está rodeada por dos delimitadores sucesivos (por ejemplo, `::` en `PATH`). Cuando `flags` tiene el valor `NOTOKENS`, se hace caso omiso de la presencia de delimitadores sucesivos, tal como lo hace la función estándar `strtok`.

- La implantación estándar de `strtok` mantiene un apuntador estático al inicio de la siguiente parte de la cadena. Los apuntadores estáticos presentan problemas si existen varios hilos de control de ejecución dentro del mismo programa. Lea la página del manual de su sistema que corresponde a `strtok` y vea si se proporciona alguna alternativa. Si es así, utilícela. De lo contrario escriba una función `strtok_r`, como está descrita en la sección 1.5. El prototipo para esta función `strtok_r` es

```
char *estrtok_r(char *s1, const char *s2, char **lasts);
```

- Escriba una forma reentrante de `strtok` denominada `strtok_r`.

1.7 Lecturas adicionales

Muchos libros que tratan los sistemas operativos en general, contienen una historia y perspectiva de éstos. Las introducciones recomendadas incluyen el capítulo 1 de *Modern Operating Systems* de Tanenbaum [92] o los capítulos del 1 al 3 de *Operating Systems Concepts* de Silberschatz y Galvin [80]. Los capítulos 1 y 2 de *Distributed Systems: Concepts and Design* de Coulouris, Dollimore y Kindberg comentan aspectos de diseño para sistemas distribuidos [21]. *Distributed Operating Systems* de Tanenbaum [93] también contiene un buen panorama de varios aspectos de los sistemas distribuidos, pero proporciona menos detalles sobre sistemas distribuidos específicos que [21].

Los requisitos previos en programación incluyen un conocimiento general de C y UNIX. The *C Programming Language*, segunda edición, por Kernighan y Ritchie [58] es una referencia estándar sobre el lenguaje C. *C: A Reference Manual*, cuarta edición, por Harbison y Steele [37] es una referencia actualizada sobre el lenguaje C. *UNIX in a Nutshell: A Desktop Quick Reference for System V* publicado por O'Reilly & Associates es una buena referencia para el usuario de UNIX [68]. *A Practical Guide to the UNIX System*, tercera edición, por Mark Sobell [82] proporciona un panorama de UNIX y sus utilerías desde la perspectiva del usuario.

Advanced Programming in the UNIX Environment de Stevens [86]es una referencia técnica importante sobre la interfaz UNIX que puede emplearse en conjunción con este libro. Los programadores de sistemas serios deben adquirir el *POSIX Std. 1003.1b System Application Program Interface (API) Amendment 1: Realtime Extension* [53]. El estándar es sorprendentemente completo y fácil de leer. Las secciones al final del mismo proporcionan muchos detalles sobre aspectos importantes. Finalmente el *Standard C Library* de Plauger es un examen interesante, y muy detallado, de la implantación de la bliblioteca de funciones de C [70]. El texto proporciona un modelo para escribir buen código en C a nivel de sistemas.

Capítulo 2

Programas y procesos

Un *proceso* es la entidad activa básica en muchos modelos de sistemas operativos. Una definición bastante común de proceso es aquella que lo describe como una instancia de un programa cuya ejecución aún no ha terminado. En este capítulo se estudia las diferencias entre programas y procesos, y cómo los primeros se transforman en los últimos. Además de abarcar el modelo y la creación de procesos, los procesos demonios y las secciones críticas, el capítulo aborda aspectos sobre la estructura de los programas y el diseño orientado a objetos.

Para escribir un programa fuente en C, el programador crea archivos en disco que contienen instrucciones en C, organizadas en funciones. Cada archivo fuente también puede contener declaraciones de variables y funciones; definiciones de tipos (por ejemplo, `typedef`), y comandos para el preprocesador (`#ifdef, #include, #define`). El programa fuente contiene sólo una función `main`. Tradicionalmente, los archivos fuentes de C tienen nombres con una extensión `.c`. Los archivos de encabezado por lo común contienen sólo definiciones de tipo, constantes definidas y declaraciones de funciones, además de una extensión `.h`.

El compilador de C traduce cada archivo fuente en un archivo objeto. Después, el compilador reúne todos los archivos objeto para crear un *módulo ejecutable*. Un *programa* es un archivo que contiene un módulo ejecutable. Cuando se ejecuta el programa, el sistema operativo copia el módulo ejecutable en una *imagen de programa* en la memoria principal. Un *proceso* es una instancia de un programa que está en ejecución. Cada instancia tiene su propio espacio de direcciones y estado de ejecución.

¿En qué momento un programa se convierte en un proceso ? El sistema operativo lee el programa en la memoria. La asignación de memoria para la imagen del programa no es suficiente para que éste se convierta en un proceso. El proceso debe tener un ID (el *ID del proceso*) de modo que el sistema operativo sea capaz de distinguir entre varios procesos. El *estado del*

proceso indica el estado de la ejecución de un proceso. El sistema operativo mantiene una lista de los ID de los procesos así como de los estados correspondientes, y utiliza esta información para asignar y administrar los recursos del sistema. El sistema operativo también mantiene una lista de la memoria ocupada por los procesos y de la que está disponible para asignación (el *administrador de memoria*).

Cuando el sistema operativo agrega la información apropiada en las estructuras de datos del núcleo (*kernel*) y asigna los recursos necesarios para ejecutar el código del programa, éste se convierte entonces en un proceso. Algunos llaman a los procesos definidos de esta manera *procesos pesados* debido al trabajo que se necesita para ponerlos en marcha, en contraste con los *procesos ligeros* (también llamados hilos de control o *threads*) estudiados en los capítulos 9 y 10.

Las siguientes dos secciones estudian la estructura y organización de programas. Las demás secciones del capítulo abordan otros aspectos de la administración de procesos incluyendo la identificación, creación, herencia y terminación.

2.1 Estructura de un programa ejecutable

Los programas en C pueden contener variables estáticas y automáticas. La clase de almacenamiento `static` se refiere a variables que, una vez asignadas, persisten durante toda la ejecución del programa. Las variables externas, aquellas que están definidas fuera de cualquier función, siempre son estáticas.

La clase de almacenamiento *automatic* se refiere a variables que inician su existencia cuando se ejecuta el bloque donde están declaradas, y que son descartadas cuando éste termina. Las variables declaradas dentro de una función son automáticas, a menos que se declaren explícitamente como estáticas. En general, las variables automáticas son asignadas en la pila del programa, de modo que es posible que existan muchas instancias de la misma variable (como en la recursión).

Desafortunadamente, la palabra resrvada `static` tiene dos significados en C. En algunos casos modifica la clase de almacenamiento, como se modificó anteriormente y en otras modifica la *clase de enlace*. Las variables y funciones pueden tener un *enlace* ya sea *interno* o *externo*. La clase de enlace determina si puede tenerse acceso a las funciones y variables desde otros archivos vinculados entre sí.

De manera preestablecida, se puede hacer referencia en otros archivos a las variables declaradas fuera de cualquier función y a todos los identificadores de nombre de función. (Esto es, el enlace preestablecido es el externo.) La palabra reservada `static` indica enlace interno para estos identificadores. En este caso, `static` modifica la clase de enlace, dado que las variables que están declaradas fuera de una función siempre tienen una clase de almacenamiento `static`. La referencia a las variables externas e identificadores de función que son modificados por la palabra reservada `static` sólo podrá hacerse dentro del archivo en que se encuentran definidas.

Las variables declaradas dentro de una función siempre tienen un enlace interno puesto que sólo puede hacerse referencia a ellas dentro de dicha función. De manera preestablecida, las variables declaradas dentro de una función tienen una clase de almacenamiento automática. La palabra reservada `static` especifica una clase de almacenamiento estática. Si la declara-

ción de una variable tiene un modificador `static`, entonces ésta tiene una clase de almacenamiento estática. Una vez asignada, la variable persiste hasta que el programa termine. La tabla 2.1 resume esta información.

Objeto	Dónde está declarado	¿Se aplica `static` modifica	Clase de `static`?	Clase de almacenamiento	Clase de enlace
variable	dentro de una función	clase de almacenamiento	sí	estático	interno
variable	dentro de una función	clase de almacenamiento	no	automático	interno
variable	fuera de una función	clase de enlace	sí	estático	interno
variable	fuera de una función	clase de enlace	no	estático	externo
función	fuera de una función	clase de enlace	sí	estático	interno
función	fuera de una función	clase de enlace	no	estático	externo

Tabla 2.1 : Efecto del uso del modificador `static` en un programa en C.

La figura 2.1 ilustra la estructura de una imagen de programa en memoria [86]. En ella parece que el programa ocupa un bloque contiguo de memoria. En la práctica, el sistema operativo asocia el espacio de direcciones lógicas contiguas del programa ejecutable a la memoria física. Una técnica de asociación común es dividir el programa en partes iguales denominadas *páginas*. Cada página puede estar localizada en cualquier parte de la memoria, y cada vez que se hace una referencia a la memoria se emprende una búsqueda. Esta transformación permite tener un espacio de direcciones lógicas muy grande para la pila y el área libre de memoria (*heap*) sin utilizar en realidad la memoria física a menos que sea necesario. La existencia de esta transformación es ocultada por el sistema operativo, de modo que el programador puede ver el programa como lógicamente contiguo.

La imagen de programa tiene varias secciones distintas. El texto o código del programa ocupa las direcciones bajas de la memoria. Las variables estáticas y sin iniciar tienen sus propias secciones en la imagen. Otras incluyen el *heap*, la pila y el ambiente.

Por lo común, las variables automáticas son parte del registro de activación de la llamada a la función donde éstas son creadas. Un *registro de activación* (activation record) es un bloque de memoria asignado en el tope de la pila que sirve para guardar el contexto de ejecución de una función cuando ésta es llamada. El registro contiene la dirección de regreso, los parámetros (cuyos valores son copiados de los correspondientes argumentos de C), información sobre el estado y una copia de los valores contenidos en los registros de la CPU al momento en que se hace la llamada, de modo que éstos puedan ser recuperados después del regreso de la función. El registro de activación también contiene las variables automáticas asignadas dentro de la función. El formato particular de un registro de este tipo depende del *hardware* y el compilador. Cada llamada a una función crea un registro de activación nuevo en la pila. El registro se elimina de la pila cuando la función regresa, siguiendo el orden el último llamado es el primero en regresar para llamadas anidadas a funciones.

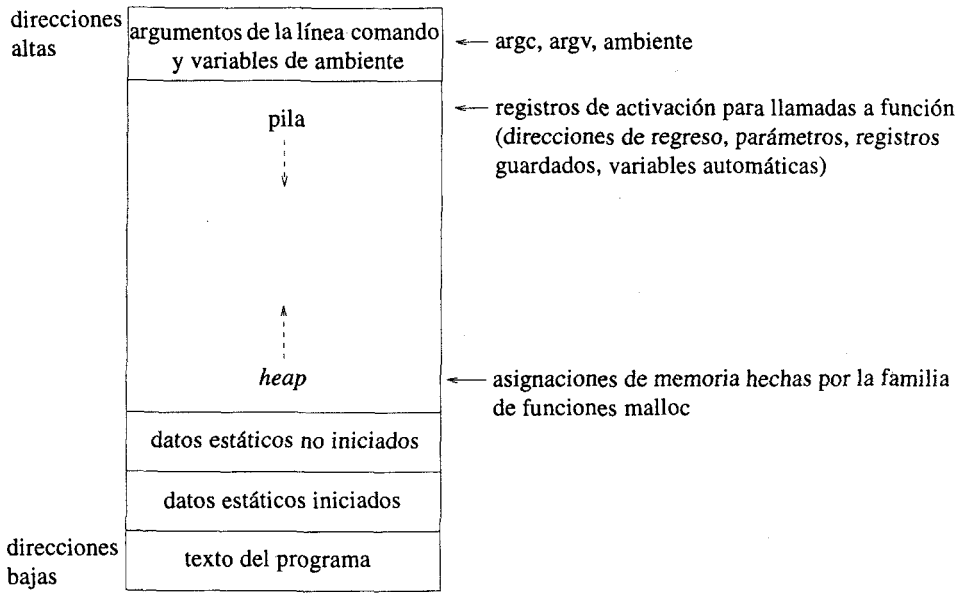


Figura 2.1: Estructura de un programa en la memoria principal

Además de las variables estáticas y automáticas, la imagen de un programa contiene espacio para `argc` y `argv`, así como para las asignaciones hechas por `malloc`. La familia de funciones `malloc` asigna espacio de almacenamiento de un área denominada *heap*. El espacio asignado en el *heap* persiste hasta que es liberado explícitamente o al terminar la ejecución del programa. Si una función llama a `malloc`, el almacenamiento asignado persiste después del regreso de la función. El programa no puede tener acceso a él después del regreso de la función, a menos que se tenga un apuntador accesible a la memoria después del regreso.

Las variables estáticas que no están iniciadas de manera explícita en sus declaraciones tienen, al momento de dar inicio a la ejecución, el valor 0. Nótese que las variables estáticas iniciadas y no iniciadas ocupan secciones diferentes en la imagen de un programa. En general, las variables estáticas iniciadas son parte del módulo ejecutable que reside en el disco, lo que no sucede con las que no están iniciadas. Es evidente que las variables automáticas no son parte del módulo ejecutable, pero son asignadas en la pila durante la ejecución. Los valores iniciales de las variables automáticas son indeterminados, a menos que el programa las inicie explícitamente.

Ejercicio 2.1

Utilice `ls -l` para comparar los tamaños de los módulos ejecutables de los dos siguientes programas en C.

Versión 1:

```
int myarray[50000] = {1, 2, 3, 4};
void main(int argc, char *argv[])
{
   myarray[0] = 3;
}
```

Versión 2:

```
int myarray[50000];
void main(int argc, char *argv[])
{
    myarray[0] = 3;
}
```

Respuesta:

El tamaño del módulo ejecutable de la versión 1 debe ser aproximadamente 200 000 bytes mayor que el de la versión 2 debido a que `myarray` de la versión 1 es un dato estático iniciado y, por tanto, parte del módulo ejecutable. En la versión 2, el arreglo `myarray` no será asignado sino hasta que el programa sea colocado en la memoria; en ese momento, todos los elementos del arreglo tendrán un valor inicial igual a 0.

Las variables estáticas pueden hacer que el uso de un programa no sea seguro en cuanto a la ejecución con hilos de control. Por ejemplo, la función `readdir` de la biblioteca de C, y las relacionadas con ella descritas en la sección 3.1.1, utilizan variables estáticas para guardar los valores de regreso. Esta estrategia también la utilizan los fragmentos de adaptación (*stubs*) para clasificar y desclasificar argumentos para llamadas a procedimientos remotos, como se describe en los capítulos 14 y 15. Ninguna de estas situaciones es segura en cuanto a la ejecución con hilos de control *threads*. Las variables estáticas externas también hacen que el código sea más difícil de depurar debido a las interacciones inesperadas entre las funciones que las utilizan. Por estas razones, evítese el empleo de variables estáticas salvo en condiciones controladas.

2.2 Objetos estáticos

Una situación en la que las variables estáticas son ampliamente utilizadas es en la implantación, en C, de una estructura de datos como un objeto. La estructura de datos junto con todas las funciones que tienen acceso a ella se encuentran en un solo archivo fuente, y la definición de estas estructuras aparece fuera de cualquier función. La estructura de datos tiene un atributo `static`, lo que le da un enlace interno: es privada con respecto del archivo fuente donde se encuentra. Todas las referencias a la estructura de datos fuera del archivo serán hechas a través de las funciones de acceso (métodos en la terminología de C++), definidas en dicho archivo. Los detalles internos sobre la estructura de datos deben ser invisibles para el mundo externo,

de modo que un cambio en la implantación interna no requiera un cambio en las referencias externas. Un programador del objeto puede crearlo con acceso seguro con hilos de control (*acces threadsafe*) colocando mecanismos de bloqueo en las funciones de acceso que están dentro de dicho archivo, sin afectar a quienes las invocan externamente.

En esta sección se desarrolla la implantación de un objeto lista con el tipo de estructuras estáticas antes descrito. El objeto lista será utilizado posteriormente por una aplicación que ejecuta comandos y mantiene una lista de los que ha ejecutado. La figura 2.2 muestra un diagrama de la lista representada como un objeto. El objeto tiene tres funciones de acceso: `add_data`, `get_data`, y `rewind_list`. Las estructuras de datos para el objeto y el código de las funciones de acceso se encuentran en un solo archivo. Este objeto lista será utilizado en varios proyectos, incluyendo el programa `newsbiff` desarrollado en la sección 2.15.

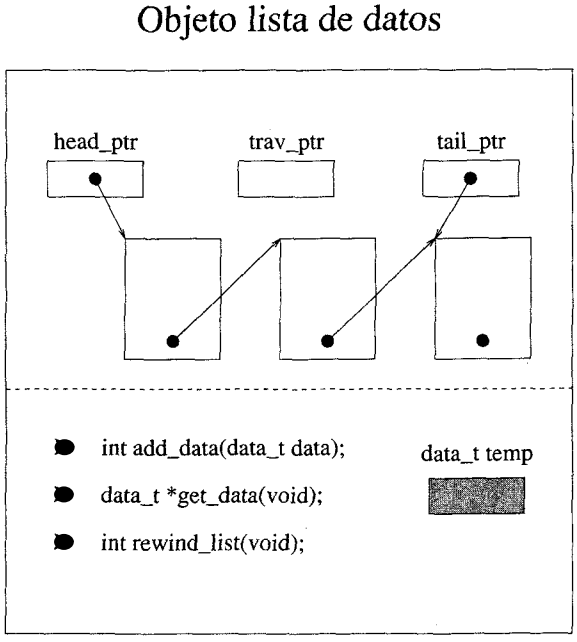


Figura 2.2: Un objeto lista con tres funciones de acceso. La variable `temp` guarda el valor devuelto para `get_data`.

En una representación del objeto, las funciones que lo invocan externamente no tienen acceso a la representación interna de éste (esto es, no saben si el objeto es un lista enlazada más que un arreglo o cualquier otra representación de la estructura de datos abstracta). El programa 2.1 muestra el archivo de encabezado `list.h`, el cual debe incluir cualquier programa que desee tener acceso a la lista. El programa reconoce la estructura de datos, pero no la lista que hay detrás de ella.

Los datos definidos en `data_t` consisten en `time_t` un valor (`time`) y (`string`) una cadena de caracteres de longitud indeterminada. El programa 2.2 muestra la implantación del

objeto lista de la figura 2.2. La implantación es complicada por la suposición de que no existe un límite superior para la longitud de `string`.

Programa 2.1: Archivo de encabezado `list.h`.

```
/*                  list.h                              */
#include <sys/types.h>
#include <time.h>

typedef struct data_struct {
     time_t time;
     char *string;
} data_t;

int add_data(data_t data);
data_t *get_data(void);
int rewind_list(void);
```

Programa 2.1

Programa 2.2: Implantación de un objeto lista.

```
#include <stdlib.h>
#include <string.h>
#include "list.h"

typedef struct list_struct {
     data_t item;
     struct list_struct *next;
} list_t;

static list_t *head_ptr = NULL;
static list_t *tail_ptr = NULL;
static list_t **trav_ptr = &head_ptr;
static data_t temp;

/* Asignación de un nodo para guardar los datos y añadirlo al final de la lista.
 * Regresa al valor 0 si hay éxito, de lo contrario -1.
 */
int add_data(data_t data)
{
   list_t *newnode;

   if ((newnode = (list_t *)(malloc(sizeof(list_t) +
                           strlen(data.string) + 1))) == NULL)
      return -1;
```

```
    newnode->item.time = data.time;
    newnode->item.string = (char *)(newnode + sizeof(list_t));
    strcpy (newnode->item.string, data.string);
    newnode->next = NULL;
    if (head_ptr == NULL)
        head_ptr = newnode;
    else tail_ptr->next = newnode;
    tail_ptr = newnode;
    return 0;
}

  /* Regresa un apuntador en temp, el cual tiene una copia de los datos
     contenidos
   * en el nodo al que apunta *trav_ptr. Si es el final de la lista
   * entonces se devuelve NULL. En caso contrario, se actualiza trav_ptr.
   */
date_t *get_data(void)
{
    list_t *t;

    t = *trav_ptr;
    if (t == NULL)
        return NULL;
   if (temp.string != NULL)
        free (temp.string);
   if ( (temp.string =
       (char *) malloc(string(t->item.string) + 1)) == NULL)
        return NULL;
    temp.time = t->item.time;
    strcpy(temp.string, t->item.string);
    trav_ptr = &(t->next);
    return &temp;
}

  /* Se pone en trav_ptr la dirección de head_ptr.
   * Si head_ptr es NULL se devuelve -1, lo que indica que la lista está vacía.
   * En caso contrario, se devuelve 0.
   */
int rewind_list(void)
{
    trav_ptr = &head_ptr;
    if (head_ptr == NULL)
        return -1;
    else
        return 0;
    }
```

Programa 2.2

La función `add_data` añade un nodo que contiene datos a la representación de la estructura interna lista. `add_data` devuelve 0 si tiene éxito, y `-1` en caso contrario. La función `malloc` asigna espacio en un bloque contiguo para `list_t` y la cadena de caracteres correspondiente. La única manera en que `add_data` puede fallar es si `malloc` lo hace. En este caso, `add_data` devuelve `-1` y `errno` tiene el valor dado por `malloc` debido a que no hay llamadas intermedias.

La función `get_data` devuelve un apuntador a una localidad temporal (`temp` de la figura 2.2) que contiene los datos del siguiente ítem de la lista. Si se hacen llamadas sucesivas a `get_data`, entonces se sobreescriben los datos. Si se llega al final de la lista, `get_data` devuelve un apuntador `NULL`. El objeto lista mantiene una variable interna `trav_ptr` que contiene la dirección de un apuntador al siguiente ítem que será accesado por `get_data`. Cada llamada a `get_data` mueve a `trav_ptr` de modo que apunte hacia el siguiente ítem.

Ejercicio 2.2

¿Por qué `trav_ptr` es de tipo `list_t **` más que de tipo `list_t *`?

Respuesta:

Suponga que `trav_ptr` es de tipo `list_t *`. Cuando `get_data` procesa el último nodo de la lista, esto hará que `trav_ptr` sea `NULL`. Si el programa llama a `add_data` antes de invocar a `get_data` de nuevo, entonces `trav_ptr` perderá el ítem que acaba de añadirse.

La función `rewind_list` vuelve a poner en `trav_ptr` la dirección de `head_ptr`. La función devuelve un 0 si la lista no está vacía, y `-1` si se encuentra vacía.

El programa 2.3 es una aplicación que ejecuta comandos y mantiene una historia interna utilizando el objeto lista del programa 2.2. El programa recibe como entradas comandos que provienen del dispositivo de entrada estándar y mantiene un registro interno de éstos y del instante en que fueron ejecutados. El programa toma un argumento de línea comando opcional, history, el cual, si está presente, hace que se imprima la historia de comandos ejecutados hasta el momento en que el programa lee la cadena `"history"` de la entrada estándar. El programa emplea `fgets` en lugar de `gets` para evitar que el buffer sea rebasado por la entrada.

El programa 2.3 llama a `runproc` para ejecutar el comando, y a `showhistory` para presentar la historia de comandos ejecutados. `MAX_CANON` es una constante definida por POSIX que especifica el número máximo de bytes en la línea de entrada de la terminal. Si `MAX_CANON` no está definida en `limits.h`, entonces la longitud máxima de la línea depende del dispositivo en particular y el programa fija el valor en 8192 bytes.

Para la ejecución del programa, la función `runproc` llama a la función `system`.

SINOPSIS

```
#include <stdlib.h>

int system(const char *string);
```

ISO C, Spec 1170

La función `system` ejecuta un `fork`, y el `exec` hijo ejecuta un shell para realizar el comando dado. La función `system` devuelve un `-1` si `fork` o `exec` fallan.

El programa 2.4 muestra el archivo fuente que contiene las funciones `runproc` y `showhistory`. Éstas tienen acceso a la historia del objeto. Cuando `runproc` ejecuta con éxito un comando, añade un nodo a la lista que guarda la historia llamando a la función `add_data`. La función `showhistory` muestra el contenido de cada nodo de la lista llamando a la función `get_data`.

Programa 2.3: Programa principal contenido en el archivo `keeplog.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>

#ifndef MAX_CANON
#define MAX_CANON 8192
#endif

void showhistory(FILE *f);
int runproc(char *cmd);

void main(int argc, char *argv[])
{
    char cmd[MAX_CANON];
    int history = 1;

    if (argc == 1)
        history = 0;
    else if ((argc > 2) || strcmp(argv[1],"history")) {
        fprintf(stderr, "Uso: %s [history]\n", argv[0]);
        exit(1);
    }
    while(fgets(cmd, MAX_CANON, stdin) != NULL) {
        if (*(cmd + strlen(cmd) -1) == '\n')
            *(cmd + strlen(cmd) -1) = 0;
        if (history && !strcmp(cmd, "history"))
            showhistory(stdout);
        else if (runproc(cmd))
            break;
    }
    printf("\n\n>>>>>>La lista de comandos ejecutados es: \n");
    showhistory(stdout);
    exit(0);
}
```

Programa 2.3

Programa 2.4: Archivo `keeploglib.c`.

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

  /* Se ejecuta cmd y se guardan cmd y la hora de ejecución en la lista de
     historial. */
int runproc(char *cmd)
{
   data_t execute;

   time(&(execute.time)=;
   execute.string = cmd;
   if (system(cmd) == -1)
      return -1;
   return add_data(execute);
}
/*Se imprime la lista de historial contenida en el archivo f. */
void showhistory(FILE *f)
{
   data_t *infop;

   rewind_list();
   while ((infop = get data ()) != NULL)
      fprintf(f, "Comando: %s\nHora: %s\n", infop->string,
   return;
}
```

Programa 2.4

2.3 El proceso ID

UNIX identifica los procesos mediante un entero único denominado *ID del proceso*. El proceso que ejecuta la solicitud para la creación de un proceso recibe el nombre de *padre* del proceso, y el proceso creado se conoce como *hijo*. El ID del proceso padre identifica al padre del proceso. Para determinar estos procesos utilice las funciones `getpid` y `getppid`.

SINOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

POSIX.1, Spec 1170

UNIX asocia cada proceso con un usuario particular conocido como *propietario* del proceso. El propietario tiene ciertos privilegios con respecto de los procesos. Cada usuario tiene un número de identificación único que se conoce como *ID del usuario*. Un proceso puede determinar el ID de usuario de su propietario con una llamada a `getuid`. El propietario es el usuario que ejecuta el programa. El proceso también tiene un *ID de usuario efectivo (effective user)*, que determina los privilegios que el proceso tiene para el acceso de recursos tales como archivos. El ID de usuario efectivo puede cambiar durante la ejecución de un proceso. El proceso puede determinar el ID de su usuario efectivo llamando a la función `geteuid`.

SINOPSIS

```
#include <sys/types.h>
#include <unistd.h>

uid_t getuid(void);
uid_t geteuid(void);
```

POSIX.1, Spec 1170

Ejemplo 2.1

El siguiente programa imprime los ID del proceso, del padre del proceso y del propietario.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main(void)
{
   printf("ID del proceso: %ld\n", (long)getpid());
   printf("ID del padre del proceso: %ld\n", (long)getppid());
   printf("ID del usuario propietario: %ld\n", (long)getuid());
}
```

Bajo POSIX, `getpid` devuelve un valor de tipo `pid_t` que puede ser un `int` o un `long`. Para evitar errores, haga que el valor devuelto por `getpid` sea de tipo `long`.

2.4 Estado de un proceso

El *estado* de un proceso indica la condición en que éste se encuentra en un momento determinado. Muchos sistemas operativos permiten alguna forma de los estados que aparecen en la tabla 2.2. El *diagrama de estados* es una representación gráfica de los estados permitidos de un proceso, así como de las transiciones permitidas entre éstos. La figura 2.3 muestra uno de estos diagramas. Los nodos de la gráfica representan los estados posibles, y las aristas las transiciones posibles. Un arco dirigido del estado **A** hacia el **B** significa que el proceso puede ir directamente del estado **A** al estado **B**. Las etiquetas sobre los arcos indican las condiciones que hacen que se presenten las transiciones entre los estados.

Estado	Significado
nuevo *(new)*	creación del proceso
ejecuta *(running)*	ejecución de las instrucciones del proceso
bloqueado *(blocked)*	proceso en espera de un evento, como una operación de E/S
listo *(ready)*	proceso en espera de ser asignado al procesador
hecho *(done)*	proceso terminado y recuperación de sus recursos

Tabla 2.2: Estados comunes de proceso.

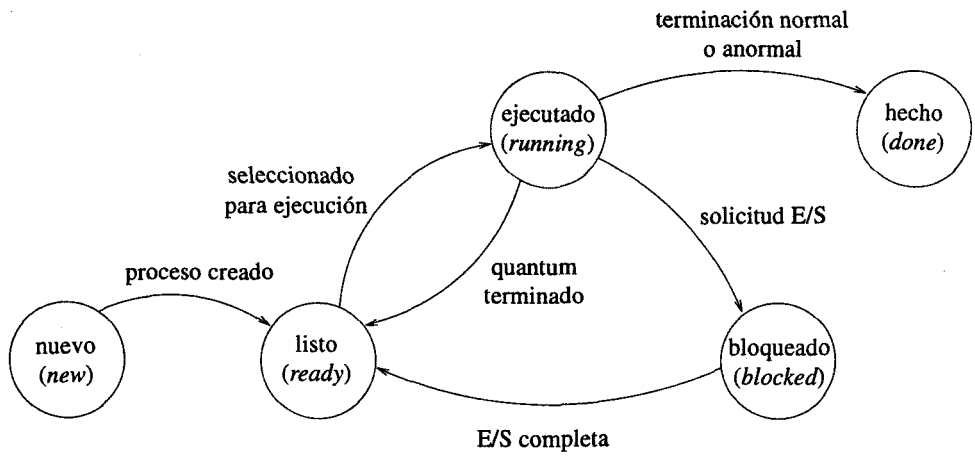


Figura 2.3: Diagrama de estado para un sistema operativo sencillo.

Se dice que un programa se encuentra en el estado *nuevo (new)* mientras experimenta la transformación que lo convertirá en un proceso activo. Cuando la transformación termina, el sistema operativo coloca el proceso en una cola de procesos que están listos para ser ejecutados. Entonces, el proceso se encuentra en el estado *listo (ready)*. En algún momento dado, el administrador de procesos lo seleccionará para ejecutarlo. El proceso se encontrará en el estado *ejecución (running)* cuando esté siendo ejecutado por el CPU.

Un proceso se encuentra en el estado bloqueado *(blocked)* si está en espera de un evento y no es considerado como candidato para ejecución por el administrador de procesos. Un proceso puede moverse voluntariamente al estado *bloqueado* haciendo una llamada a una función como `sleep`. Lo más común es que el proceso se mueva al estado *bloqueado* cuando lleva a cabo una solicitud de E/S. La entrada y la salida pueden ser miles de veces más lentas que las instrucciones ordinarias. La E/S es manejada por el sistema operativo, y un proceso lleva a cabo una operación E/S solicitando el servicio a través de una *llamada al sistema*. El sistema operativo vuelve a tomar el control y puede mover el proceso al estado bloqueado hasta que la operación termine.

El acto de quitar un proceso del estado ejecución y reemplazarlo con otro se conoce como *conmutación de contexto (context switch)*. El *contexto* de un proceso es la información sobre el proceso y su ambiente necesaria para continuarlo después de una conmutación de contexto. Es evidente que el ejecutable, la pila, los registros y el contador del programa son parte del contexto, al igual que la memoria utilizada por las variables asignadas estática y dinámicamente. Para poder continuar el proceso de manera transparente, el sistema operativo también mantiene el estado del proceso, el estado de la E/S del programa, la identificación del usuario y el proceso, los privilegios, información sobre la planificación y administración, e información sobre la administración de la memoria. Otra información que también es parte del contexto es la de si un proceso está en espera de un evento o ha atrapado una señal. El contexto también contiene información sobre otros recursos, como los bloqueos mantenidos por el proceso.

La utilería `ps` muestra información sobre procesos.

SINOPSIS

```
ps [-aA] [-G grouplist] [-o format]...[-p proclist]
   [-t termlist] [-U userlist]
```

POSIX.2

De manera preestablecida, `ps` presenta información sobre los procesos asociados con el usuario. La opción `-a` muestra información de los procesos asociados con terminales, mientras que la opción `-A` visualiza información de todos los procesos. La opción `-o` especifica el formato de la salida.

La especificación Spec *1170 difiere un poco de la proporcionada por* POSIX.

SINOPSIS

```
ps [-aA] [-defl] [-G grouplist] [-o format]...[-p proclist]
   [-t termlist] [-U userlist] [-g grouplist] [-n namelist]
   [-u userlist]
```

Spec 1170

Muchas de las implantaciones de `ps` que hacen los vendedores no cumplen exactamente con la especificación POSIX o Spec 1170. Por ejemplo, Sun Solaris 2 emplea `-e` en lugar de `-A` para la opción que permite obtener información sobre todos los procesos. La versión de Sun sin ningún argumento presenta información sobre los procesos asociados con la terminal de control más que con el usuario. Las terminales de control serán estudiadas en la sección 7.5.

Ejemplo 2.2

La ejecución de `ps -l` *bajo Sun Solaris 2 produce la salida siguiente:*

```
F S   UID   PID  PPID  C PRI NI     ADDR     SZ    WCHAN TTY      TIME COMD
8 S   512  4509  4502 80  40 20 fc579000    205 fc5791c8 pts/13   0:00 csh
8 O   512  4627  4509 13  60 20 fc5b1800    151          pts/13   0:00 ps
```

La forma larga del `ps` de Sun del ejemplo 2.2 muestra mucha información interesante sobre los procesos asociados con la terminal de control activa (la cual tiene el nombre de dispositivo `pts/13`). La tabla 2.3 contiene un resumen con el significado de los distintos campos.

Encabezado	Significado
F	banderas asociadas con el proceso
S	estado del proceso
UID	ID del usuario propietario del proceso
PID	ID del proceso
PPID	ID del padre del proceso
C	utilización del procesador empleada para la administración de procesos
PRI	prioridad del proceso
NI	valor amabilidad (*nice value*)
ADDR	dirección en memoria del proceso
SZ	tamaño de la imagen del proceso
WCHAN	dirección del evento si el proceso está suspendido
TTY	terminal de control
TIME	tiempo acumulado de ejecución
COMMAND	nombre del comando

Tabla 2.3: Campos notificados por la forma larga del comando `ps` de Sun Solaris

2.5 Creación de procesos y el `fork` de UNIX

UNIX crea los procesos a través de una llamada `fork` al sistema, copiando la imagen en memoria que tiene el proceso padre. El nuevo proceso recibe una copia del espacio de direcciones del padre. Los dos procesos continúan su ejecución en la instrucción que está después del `fork`.

SINOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

POSIX.1, Spec 1170

La creación de dos procesos totalmente idénticos no es algo muy útil. El valor devuelto por `fork` es la característica distintiva importante que permite que el padre y el hijo ejecuten código distinto. El `fork` devuelve 0 al hijo y el ID del hijo, al padre.

Ejemplo 2.3

En el siguiente fragmento de código tanto el padre como el hijo ejecutan la instrucción de asignación `x = 1` *después del regreso de* `fork`. *Existe un proceso y una sola variable* `x` *antes de la ejecución de* `fork`.

```
#include <sys/types.h>
#include <unistd.h>

x = 0;
fork();
x = 1;
```

Después del `fork` del ejemplo 2.3, existen dos procesos independientes. Cada uno tiene su propia copia de la variable x. Puesto que los procesos padre e hijo son ejecutados de manera independiente, no ejecutan el código que esté bloqueado ni tampoco modifican la misma localidad de memoria. No es posible distinguir los procesos padre e hijo, ya que no se examinó el valor devuelto por `fork`.

Ejemplo 2.4

Después de la llamada a `fork` *en el siguiente fragmento de código, los procesos padre e hijo imprimen sus ID de proceso.*

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

if ((childpid = fork()) == 0) {
    fprintf(stderr, "Soy el hijo, ID = %ld\n", (long)getpid());
        /* el código del hijo va aquí */
} else if (childpid > 0) {
    fprintf(stderr, "Soy el padre, ID = %ld\n", (long)getpid());
        /* el código del padre va aquí */
}
```

El valor de la variable `childpid` en los procesos originales del ejemplo 2.4 es distinto de cero, así que con esto se ejecuta la segunda instrucción `fprintf`. El valor de `childpid` en el proceso hijo es cero, lo que ejecuta la primera instrucción `fprintf`. La salida de `fprintf` puede aparecer en cualquier orden.

Ejemplo 2.5

El siguiente fragmento de código crea una cadena de `n` *procesos.*

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
```

```
int i;
int n;
pid_t childpid;
for (i = 1; i < n; ++i)
   if (childpid = fork())
      break;
fprintf(stderr, "Éste es el proceso %ld con padre %ld\n",
      (long)getpid(), (long)getppid());
```

Con cada ejecución de `fork` en el ejemplo 2.5, el valor de `childpid` en el proceso padre es distinto de cero, lo que termina el ciclo. El valor de `childpid` en el proceso hijo es cero, con lo que éste se convierte en un padre en la siguiente iteración del ciclo. En caso de que haya un error en la llamada a `fork`, el valor de regreso es -1 y el proceso que hizo la llamada es el que termina el ciclo. Los ejercicios de la sección 2.12 se basan en este ejemplo.

La figura 2.4 muestra una representación gráfica de la cadena de procesos generada por el ejemplo 2.4 cuando `n` es 4. Cada círculo representa un proceso y está etiquetado por el valor de `i` que tiene el proceso correspondiente cuando abandona el lazo. Las aristas representan la relación "is a parente A" $\rightarrow B$ significa que A es el padre del proceso B.

Figura 2.4: Cadena de procesos generada por el fragmento de código del ejemplo 2.5 cuando `n` es 4.

Ejemplo 2.6

El siguiente fragmento de código crea un abanico de procesos.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
```

```
int i;
int n;
pid_t childpid;

for (i = 1; i < n; ++i)
   if (childpid = fork()) <= 0)
      break;
fprintf(stderr, "Éste es el proceso %ld con padre %ld\n",
      (long)getpid(), (long)getppid());
```

La figura 2.5 muestra el abanico de procesos generado por el ejemplo 2.6 cuando `n` es 4. Los procesos de la figura están señalados con los valores de `i` que éstos tienen cuando abandonan el ciclo del ejemplo 2.6. El proceso original crea `n - 1` hijos. Los ejercicios de la sección 2.13 se basan en este ejemplo.

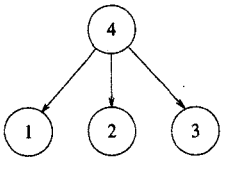


Figura 2.5: Abanico de procesos generado por el código del ejemplo 2.6 cuando `n` es 4.

Ejemplo 2.7

El siguiente código produce un árbol de procesos.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int i;
int n;
pid_t childpid;

for (i = 1; i < n; i++)
   if (childpid = fork()) == -1)
      break;
fprintf(stderr, "Éste es el proceso %ld con padre %ld\n",
      (long)getpid(), (long)getppid());
```

La figura 2.6 muestra el árbol de procesos generado por el ejemplo 2.7 cuando `n` es 4. Cada proceso está representado por un círculo y tiene como etiqueta el valor de `i` al momento en que fue creado.

El proceso original tiene la etiqueta 0. Las letras minúsculas distinguen procesos que fueron creados con el mismo valor de `i`. Si bien este código parece ser similar al del ejemplo 2.5, no hace distinción entre el padre y el hijo en el `fork`. Tanto el padre como el hijo crean hijos en la siguiente iteración del ciclo, esto explica la explosión poblacional de procesos.

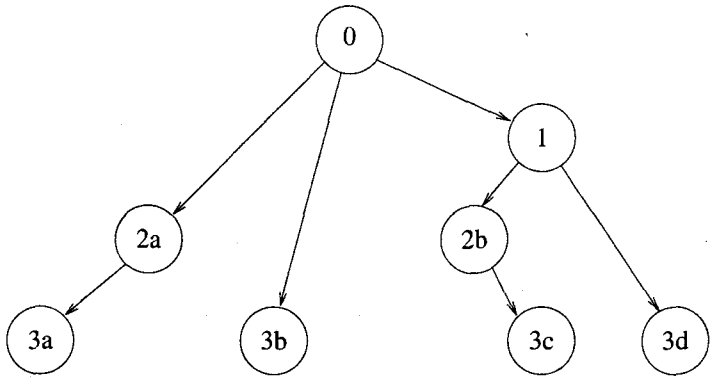


Figura 2.6: Arbol de procesos producido por un ciclo `fork`.

`Fork` crea procesos nuevos haciendo una copia de la imagen del padre en la memoria. El hijo *hereda* la mayor parte de los atributos del padre, incluyendo el ambiente y los privilegios. El hijo también hereda algunos de los recursos del padre, tales como los archivos y dispositivos abiertos. Las implicaciones de la herencia son más complicadas de lo que en principio parecen ser. En la sección 3.3 y el capítulo 4 se explora los aspectos de la herencia de archivos.

No todos los atributos o recursos del padre son heredados por el hijo. Este último tiene un ID de proceso nuevo y, claro está, un ID de padre diferente. Los tiempos del hijo para el uso del CPU son iniciados a 0. El hijo no obtiene los bloqueos que el padre mantiene. Si el padre ha puesto una alarma, el hijo no recibe notificación alguna del momento en que ésta expira. El hijo comienza sin señales pendientes, aunque el padre las tenga en el momento en que se ejecuta el `fork`.

Aunque el hijo hereda la prioridad del padre y los atributos de la administración de procesos, tiene que competir con otros procesos, como entidad aparte, por el tiempo del procesador. Un usuario que utiliza un sistema de tiempo compartido muy concurrido puede obtener un tiempo de acceso mayor mediante la creación de muchos procesos. Por el contrario, el sistema operativo VAX VMS permite la creación de un tipo de procesos en los que todos aquellos que son creados por un solo usuario comparten el tiempo de CPU asignado al usuario. El administrador de un sistema UNIX académico con gran demanda puede restringir la creación de procesos para impedir que un usuario cree procesos con la finalidad de obtener un segmento mayor de los recursos.

2.6 Llamada `wait` al sistema

¿Qué sucede con el proceso padre después de que éste crea un hijo? Tanto el padre como el hijo continúan la ejecución desde el punto donde se hace la llamada a `fork`. Si un padre desea esperar hasta que el hijo termine, entonces debe ejecutar una llamada a `wait` o a `waitpid`.

SINOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

POSIX.1, Spec 1170

La llamada al sistema `wait` detiene al proceso que llama hasta que un hijo de éste termine o se detenga, o hasta que el proceso que la invocó reciba una señal. `Wait` regresa de inmediato si el proceso no tiene hijos o si el hijo termina o se detiene y aún no se ha solicitado la espera. Si `wait` regresa debido a la terminación de un hijo, el valor devuelto es positivo e igual al ID de proceso de dicho hijo. De lo contrario, `wait` devuelve `-1` y pone un valor en `errno`. Un `errno` igual a `ECHILD` indica que no existen procesos hijos a los cuales esperar, mientras que un `errno` igual a `EINTR` señala que la llamada fue interrumpida por una señal. `stat_loc` es un apuntador a una variable entera. Si quien hace la llamada pasa cualquier cosa distinta a `NULL`, `wait` guarda el estado devuelto por el hijo. POSIX especifica los macros `WIFEXITED`, `WEXITSTUS`, `WIFSIGNALED`, `WTERMSIG`, `WIFSTOPPED` y `WSTOPSIG` para analizar el estado devuelto por el hijo y que permanece guardado en `*stat_loc`. El hijo regresa su estado llamando a `exit`, `_exit` o `return`.

Ejemplo 2.8

En el código siguiente el padre determina el estado de la salida de un hijo.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

pid_t child
int status;

while(((child = wait(&status)) == -1) && (errno == EINTR))
   ;
if (child == -1)
   perror("No fue posible esperar al hijo");
else if (!status)
   printf("El hijo %ld terminó normalmente, el estado devuelto es cero\n",
          (long)child);
else if (WIFEXITED(status))
   printf("El hijo %ld terminó normalmente, el estado devuelto es %d\n",
          (long)child), (WEXITSTATUS(status));
```

```
else if (WIFSIGNALED(status))
   printf("El hijo %ld terminó debido a una señal no atrapada\n",
          (long)child);
```

El programa 2.5 ilustra la llamada `wait` al sistema. El proceso sólo tiene un hijo; así que si el valor devuelto no es el ID de proceso de dicho hijo entonces `wait` regresó tal vez por causa de una señal.

Programa 2.5 : Programa simple que ilustra el empleo de `wait`.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

void main (void)
{
   pid_t childpid;
   int status;

   if ((childpid = fork()) == -1) {
      perror("fork falló");
      exit(1);
   } else if (childpid == 0)
      fprintf(stderr,
         "Soy el hijo con pid = %ld\n";(long)getpid());
   else if (wait(&status) != childpid)
      fprintf(stderr,"Una señal debió interrumpir la espera\n");
   else
      fprintf(stderr,
         "Soy el padre con pid = %ld e hijo con pid = %ld\n",
        (long)getpid(), (long)childpid);
   exit(0);
}
```

Programa 2.5

Ejercicio 2.3

¿Cuáles son las formas posibles de la salida generada por el programa 2.5?

Respuesta:

Existen varias posibilidades.

- Si `fork` falla (lo que es poco probable a menos que el programa genere un árbol sin fin de procesos), aparece entonces el mensaje `"fork falló"`. De lo contrario, si no existen señales, debe aparecer algo parecido a lo siguiente:

```
Soy el hijo con pid = 3427
Soy el padre con pid = 3426 e hijo pid = 3427
```

- Si una señal llega después de que el hijo ejecute `fprintf` pero antes del `exit`, entonces aparece lo siguiente:

```
Soy el hijo con pid = 3427
Una señal debió interrumpir la espera
```

- Si una señal llega después de que el proceso hijo termine y `wait` regrese, se imprime lo siguiente:

```
Soy el hijo con pid = 3427
Soy el padre con pid = 3426 e hijo pid = 3427
```

- Si la señal llega después de la terminación del hijo, pero antes de que `wait` regrese, entonces cualquiera de los dos resultados es posible, lo que depende del momento en que llegue la señal.
- Si la señal llega antes que el hijo ejecute el `fprintf` y si el padre ejecuta primero su instrucción `fprintf`, entonces aparece lo siguiente:

```
Una señal debió interrumpir la espera
Soy el hijo con pid = 3427
```

- Finalmente, si la señal llega antes de que el hijo ejecute el `fprintf` pero el hijo logra ejecutar éste, entonces se imprime lo siguiente:

```
Soy el hijo con pid = 3427
Una señal debió interrumpir la espera
```

Para que el hijo del programa 2.5 siempre imprima su mensaje primero, el padre debe esperar repetidamente hasta que el hijo termine, antes de imprimir su propio mensaje.

Ejemplo 2.9

El siguiente fragmento de código reinicia el `wait` *hasta que termine un proceso hijo en particular.*

```
#include <sys/types.h>
#include <sys/wait.h>
int status;
pid_t childpid;

while(childpid != wait(&status))
```

El `wait` del ejemplo 2.9 puede fallar en regresar el `childpid` si encuentra un error. La manera de distinguir entre una falla debida a una señal de otros errores, es analizar `errno`. Cuando `wait` falla, devuelve `-1` y establece el valor de `errno`. Un valor de `errno` igual a `EINTR` indica interrupción por una señal.

Ejemplo 2.10

El siguiente segmento de código ejecuta el ciclo hasta que el hijo con ID de proceso `chilpid` *termine o se presente algún error.*

```
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>

int status;
pid_t childpid;

while(childpid != wait(&status))
   if ((childpid == -1) && (errno != EINTR))
      break;
```

La llamada `waitpid` al sistema proporciona métodos más flexibles para esperar a los hijos. Un proceso puede esperar un hijo en particular sin tener que esperar a todos sus hijos. Esta característica es útil para hacer el seguimiento de cierto hijo sin interferencia alguna por parte de los demás hijos cuya ejecución haya terminado. `Waitpid` también tiene una forma que no bloquea, de modo que un proceso pueda verificar de manera periódica las condiciones de no espera de los hijos sin quedarse suspendido indefinidamente.

La llamada a la función `waitpid` tiene tres parámetros: un `pid`, un apuntador que señala hacia una localidad donde guardar el estado, y las banderas de opción. Si `pid` es `-1`, `waitpid` espera a cualquier proceso. Si `pid` es positivo, entonces `waitpid` espera al hijo cuyo ID de proceso es `pid`. La opción `WNOHANG` hace que `waitpid` regrese, incluso si el estado del hijo no está disponible de inmediato. Para una especificación completa de todos los parámetros de `waitpid`, consulte la página del manual correspondiente.

Ejemplo 2.11

El siguiente fragmento de código espera a cualquier hijo, evitando el bloqueo si no hay hijos cuyo estado se encuentre disponible de inmediato. El procedimiento vuelve a iniciarse si `waitpid` *es interrumpida por una señal.*

```
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
int status;
pid_t waitreturnpid;

while(waitreturnpid = waitpid(-1, &status, WNOHANG))
   if ((waitreturnpid == -1) && (errno != EINTR))
      break;
```

Cuando `waitpid` devuelve 0, significa que existen hijos a los que todavía hay que esperar, pero ninguno de ellos está listo. ¿Qué sucede con un proceso cuyo padre no lo espera? En la terminología de UNIX, éste se convierte en un *zombie*. Los zombies permanecen en el sistema hasta que alguien los espere. Si un padre termina y no espera a uno de sus hijos, el hijo se convierte en un *huérfano* y es adoptado por el proceso `init` del sistema, el cual tiene un ID

de proceso igual a 1. El proceso `init` espera periódicamente a los hijos de modo que, en algún momento, los *zombies* huérfanos desaparecen del sistema.

Ejercicio 2.4

El siguiente segmento de código crea un abanico de procesos. Todos los procesos generados por llamadas a `fork` son hijos del proceso original. ¿Cuál es el orden de los mensajes de salida?

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

int i;
int n;
pid_t childpid;
int status;

for (i = 1; i < n; ++i)
   if (childpid = fork()) <= 0)
      break;
for( ; ; ) {
   childpid = wait(&status);
   if ((childpid == -1) && (errno != EINTR))
      break;
}
fprintf(stderr, "Soy el proceso %ld, mi padre es %ld\n",
                (long)getpid(), (long)getppid());
```

Respuesta:

Dado que ninguno de los hijos generados por `fork` son padres, el `wait` de éstos devuelve `-1` y hace que `errno` sea `ECHILD`. Ellos no son bloqueados por el segundo ciclo `for`. Sus mensajes de identificación pueden aparecer en cualquier orden. El mensaje del proceso original aparecerá al final, después de haber esperado a todos sus hijos.

Ejercicio 2.5

El siguiente fragmento de código crea una cadena de procesos. Sólo uno de los procesos generados por `fork` es un hijo del proceso original. ¿Cuál es el orden en que aparecen los mensajes?

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

int i;
int n;
pid_t childpid;
```

```
int status;
pid_t waitreturn;

for (i = 1; i < n; ++i)
   if (childpid = fork())
      break;
while(childpid != waitreturn = wait(&status)))
   if ((waitreturn == -1) && (errno != EINTR))
      break;
fprintf(stderr, "Soy el proceso %ld, mi padre es %ld\n",
                (long)getpid(), (long)getppid());
```

Respuesta:

Cada hijo generado por `fork` espera a que su hijo termine antes de escribir su propio mensaje de salida. Los mensajes aparecen en orden inverso al de la creación de los procesos.

2.7 Llamada exec al sistema

La llamada `fork` al sistema crea una copia del proceso que la llama. Muchas aplicaciones requieren que el proceso hijo ejecute un código diferente del de su padre. La familia `exec` de llamadas al sistema proporciona una característica que permite traslapar al proceso que llama con un módulo ejecutable nuevo. La manera tradicional de utilizar la combinación `fork-exec` es dejar que el hijo ejecute el `exec` para el nuevo programa mientras el padre continúa con la ejecución del código original.

Las seis variaciones de la llamada `exec` al sistema se distinguen por la forma en que son pasados los argumentos de la línea comando y el ambiente, y por si es necesario proporcionar la ruta de acceso y el nombre del archivo ejecutable. Las llamadas `execl` (`execl`, `execlp` y `execle`) pasan los argumentos de la línea comando como una lista y son útiles si se conoce el número de argumentos de línea comando se conoce al momento de la compilación. Las llamadas `execv` (`execv`, `execvp` y `execve`) pasan los argumentos de la línea comando en un arreglo de argumentos.

El código del programa 2.6 llama al comando `ls` con un argumento de línea comando igual a `-l`. El programa supone que `ls` está localizado en el directorio `/usr/bin`.

Programa 2.6: Programa que crea un proceso para ejecutar `ls -l`.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void main (void)
{
   pid_t childpid;
```

```
    int status;

    if ((childpid = fork()) == -1) {
        perror("Error al ejecutar fork");
        exit(1);
    } else if (childpid == 0) {                    /* código del hijo */
        if (execl("/usr/bin/ls", "ls", "-1", NULL) < 0) {
            perror("Falla en la ejecución de ls");
            exit(1);
        }
    } else if (childpid ! wait(&status))       /* código del padre */
        perror("Se presentó una señal antes de la terminación del hijo");
    exit(0);
}
```

Programa 2.6

SINOPSIS

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ...,
          const char *argn, char * /*NULL*/);
int execle (const char *path,char *const arg0[], ... ,
          const char *argn, char * /*NULL*/,
          char *const envp[]);
int execlp (const char *file, const char *arg0, ...,
          const char *argn, char * /*NULL*/);
```

POSIX.1, Spec 1170

El parámetro `path` de `execl` es la ruta de acceso y el nombre del programa, especificado ya sea como un nombre con la ruta completa o relativa al directorio de trabajo. Después aparecen los argumentos de la línea comando seguidos de un apuntador `NULL`. La lista de parámetros de cadena de caracteres corresponde al arreglo `argv` para el comando `exec`. Puesto que `argv[0]` es el nombre del programa, éste es el segundo argumento de `execl`.

Una forma alternativa es `execlp`, la cual tiene los mismos parámetros que `execl` pero utiliza la variable de ambiente `PATH` para buscar el ejecutable. De manera similar, cuando el usuario introduce un comando, el shell intenta localizar el archivo ejecutable en uno de los directorios especificados por la variable `PATH`.

Una tercera forma de `execl` es `execle`, la cual es similar a `execl` con la excepción de que toma un parámetro adicional que representa el nuevo ambiente del programa por ejecutar. En el casop de las otras formas de `execl`, el nuevo programa hereda el ambiente del padre.

`Execv` toma exactamente dos parámetros, un nombre y ruta de acceso para el ejecutable y un arreglo de argumentos. (En este caso, resulta útil la función `makeargv` del programa 1.2.) `Execvp` construye un nombre y ruta de acceso completo a partir del parámetro `file` al utilizar los prefijos de ruta de acceso encontrados en la variable de ambiente `PATH`. La forma `execve`

requiere un tercer parámetro, el arreglo de argumento `envp`, el cual especifica el ambiente para el proceso creado.

SINOPSIS

```
#include <unistd.h>

int execv(const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
int execve (const char *path, char *const argv[],
      char *const envp[]);
```

POSIX.1, Spec 1170

El programa 2.7 utiliza a `execvp` para ejecutar el comando pasado por la línea de comando. Suponga que el ejecutable del programa 2.7 es `myexec`. El proceso original crea un hijo y luego espera a que éste termine. El proceso hijo llama a `execvp` con el arreglo de argumento formado por los argumentos de la línea comando del programa original.

Ejemplo 2.12

La línea siguiente comando hace que `myexec` *cree un nuevo proceso para ejecutar el comando* `ls -l`.

```
myexec ls -l
```

El arreglo `argv` original producido en el ejemplo 2.12 contiene apuntadores a tres fichas (*tockens*): `myexec`, `ls` y `-l`. El arreglo de argumento para la función `execvp` comienza en `&argv[1]` y contiene apuntadores a los dos componentes léxicos `ls` y `-l`. El padre espera al hijo. Si el `wait` es interumpido por una señal, `errno` es igual a `EINTR` y el padre vuelve a comenzar la espera.

Programa 2.7: Programa que crea un proceso para ejecutar el comando que es pasado como argumento de la línea comando.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>

void main(int argc, char *argv[])
{
   pid_t childpid;
   int status;

   if ((childpid = fork()) == -1) {
      perror("Error al ejecutar fork");
```

```
        exit(1);
    } else if (childpid == 0) {                    /* código del hijo */
        if (execvp(argv[1], &argv[1]) < 0) {
            perror("Falla en la ejecución del comando");
            exit(1);
            }
    } else                                         /* código del padre */
        while(childpid != wait(&status))
             if ((childpid == -1) && (errno != EINTR))
                 break;
    exit(0);
}
```

Programa 2.7

Ejercicio 2.6

Ejecute `myexec` del programa 2.7 con la línea comando `myexec ls -l *.c`. ¿Cuál es el tamaño del arreglo de argumentos pasado como el segundo argumento a `execvp`?

Respuesta:

El tamaño depende del número de archivos con extensión `.c` presentes en el directorio, ya que el shell amplía `*.c` antes de pasar la línea comando a `myexec`.

El programa 2.8 llama a la función `makeargv` del programa 1.2 para crear un arreglo de argumento a partir de la cadena de caracteres pasada como el primer argumento de la línea comando. Luego, se hace un `execvp` del comando representado por dicha cadena de caracteres. El paso de una cadena que contiene muchos componentes léxicos se hace delimitando ésta con comillas dobles (por ejemplo, `myexec "ls -l"`).

Observe que la llamada a la función `makeargv` la hace sólo el proceso hijo del programa 2.8. Si el padre llama a `makeargv` antes del `fork`, entonces el padre tiene un arreglo de argumentos sin utilizar asignado en su *heap*. Una sola llamada a `makeargv` no presenta ningún problema. Sin embargo, en un *shell* donde el paso de asignación puede repetirse cientos de veces, la limpieza de la memoria puede convertirse en un problema.

Programa 2.8: Programa que crea un proceso para ejecutar el comando que es pasado como argumento de la línea comando. El programa 1.2 muestra una implantación de la función `makeargv`.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
```

```
int makeargv(char *s, char *delimiters, char ***argvp);

void main(int argc, char *argv[]) {
    char **myargv;
    char delim[] = " \t";
    pid_t childpid;
    int status;

    if (argc != 2) {
        fprintf(stderr, Uso: %s string\n", argv[0];
        exit(1);
    }
    if ((childpid = fork()) == -1) {
        perror("Error al ejecutar fork");
        exit(1);
    } else if (childpid == 0) {                    /* código del hijo */
      if (makeargv(argv[1], delim, &myargv) < 0) {
          fprintf(stderr, "No fue posible construir el arreglo de argumentos\n");
          exit(1);
      } else if (execvp(myargv[0], &myargv[0]) < 0) {
          perror("Falla en la ejecución del comando");
          exit(1);
      }
    } else                                          /* código del padre */
        while(childpid != wait(&status))
            if ((childpid == -1) && (errno != EINTR))
                break;
    exit(0);
}
```

Programa 2.8

Exec copia un ejecutable nuevo en la imagen del proceso. No se sabe con exactitud qué es lo que se conserva del proceso original. Tanto el texto del programa como las variables, la pila y el *heap* son sobrescritos. El nuevo proceso hereda el ambiente (lo que significa la lista de variables de ambiente y sus valores asociados), a menos que el proceso original llame a execle o execve. Normalmente, los archivos que están abiertos antes del exec por lo común permanecen abiertos (como se explica en el capítulo 3.3). El capítulo 5 estudia los efectos de exec sobre las señales y los bloqueos.

La tabla 2.4 presenta un resumen de los atributos heredados por los procesos ejecutados por exec. La segunda columna de la tabla proporciona las llamadas al sistema relacionada con los ítems. Los ID asociados con el proceso quedan intactos después del exec. Si un proceso establece una alarma antes de llamar a exec, ésta generará de todos modos una señal cuando el tiempo termine. Las señales que están pendientes también serán traspasadas al exec, contrariamente a lo que sucede con fork. Los procesos crean archivos con los mismos permisos igual que antes del exec, y la contabilidad del tiempo de CPU continúa sin ser reiniciada.

Atributo	Llamada relevante al sistema
ID del proceso	`getpid( )`
ID del padre del proceso	`getppid( )`
ID de grupo del proceso	`getpgid( )`
membresía de sesión	`getsid( )`
ID real del usuario	`getuid( )`
ID real del grupo	`getgid( )`
ID de grupos complementarios	`getgroups( )`
tiempo que resta en la señal de alarma	`alarm( )`
directorio de trabajo en uso	`getcwd( )`
directorio raíz	
máscara del modo de creación del archivo	`umask( )`
máscara de señal del proceso	`sigprocmask( )`
señales pendientes	`sigpending( )`
tiempo utilizado hasta el momento	`times( )`

Tabla 2.4 : Atributos conservados después de hacer llamadas a la función exec. En la segunda columna aparecen las llamadas al sistema importantes para estos atributos.

2.8 Procesos en plano secundario y demonios

El *shell* es un intérprete de comandos dispuesto a aceptar comandos, leerlos de la entrada estándar, crear hijos para ejecutarlos, y esperar a los hijos terminen. Cuando la entrada y la salida provienen de un dispositivo tipo terminal, el usuario puede terminar la ejecución de un comando presionando el carácter de interrupción. (Éste puede ser configurado, pero lo más común es que sea `ctrl-c`.)

Ejercicio 2.7

Vaya a un directorio que contenga muchos archivos (por ejemplo, `/etc`), y ejecute el siguiente comando.

```
ls -l
```

¿Qué sucede? Ahora ejecute `ls -l` de nuevo pero presione `ctrl-c` tan pronto como aparezca el listado en la pantalla. Compare los resultados con lo ocurrido en el primer caso.

Respuesta:

En el primer caso el resultado se imprime después de que el listado del directorio se ha completado debido a que el *shell* espera al hijo antes de continuar. En el segundo caso, `ctrl-c` termina la ejecución de `ls`.

Muchos *shells* interpretan una línea que termina con un & como un comando que debe ser ejecutado por un proceso en plano secundario, en asincronía. Cuando el *shell* crea un proceso de fondo, no espera a que éste termine para imprimir el indicador de línea comando y aceptar más comandos. Por otra parte, el `ctrl-c` del teclado no termina un proceso en plano secundario. (El capítulo 7 presenta una discusión más técnica de los procesos en plano secundario.)

Ejercicio 2.8

Compare los resultados del ejercicio 2.7 con los que se obtiene al ejecutar el siguiente comando.

```
ls -l &
```

Vuelva a introducir el comando `ls - l &` e intente terminarlo presionando `ctrl-c`.

Respuesta:

En el primer caso el indicador de línea comando aparece antes de que listado esté completo. El `ctrl-c` no afecta al proceso de fondo, de modo que el segundo caso se comporta igual que el primero.

Un *demonio* es un proceso en plano secundario que normalmente se ejecuta por tiempo indefinido. El sistema operativo UNIX depende de muchos procesos demonios para llevar a cabo tareas rutinarias (y otras no tan rutinarias). Bajo Solaris 2, el demonio `pageout` maneja la paginación para la administración de la memoria. El `in.rlogind` maneja las solicitudes remotas de acceso al sistema. Otros demonios se encargan del correo electrónico, ftp, estadísticas, y solicitudes de impresión, para nombrar sólo unas cuantas tareas. Cuando la ejecución del demonio termina, éste no deja ninguna pista de su procedencia.

El programa `runback` del programa 2.9 ejecuta el primer argumento de la línea comando como un proceso en plano secundario. El hijo llama a `setsid` para no ser molestado por ninguna interrupción de tipo `ctrl-c` proveniente de la terminal de control.

Programa 2.9 : Programa `runback` que crea un proceso para ejecutar un comando en plano secundario.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int makeargv(char *s, char *delimiters, char ***argvp);

void main(int argc, char *argv[]) {
   char **myargv;
   char delim[] = " \t";
   pid_t childpid;

   if (argc != 2) {
```

```
        fprintf(stderr, Uso: %s string\n", argv[0]);
        exit(1);
    }

    if ((childpid = fork()) == -1) {
        perror("Error al ejecutar fork");
        exit(1);
    } else if (childpid == 0) { /* el hijo se convierte en un proceso en plano
                                   secundario */
      if (setsid() == -1)
          perror("No es posible convertirse en un líder de sesión");
      else if (makeargv(argv[1], delim, &myargv) < 0)
          fprintf(stderr, "No fue posible construir el arreglo de argumentos\n");
      else if (execvp(myargv[0], &myargv[0]) < 0)
          perror("Falla en la ejecución del comando");
      exit(1);                              /* el hijo nunca debe regresar */
    }
    exit(0);                                          /* el padre termina */
}
```

Programa 2.9

El programa `runback` utiliza a `setsid` para crear una sesión nueva que no tenga una terminal de control. El ID de sesión determina si el proceso tiene una terminal de control (de modo que pueda recibir una señal proveniente de `ctrl-c`). El capítulo 7 explora estos aspectos más detalladamente.

Ejemplo 2.13

El siguiente comando es similar a la introducción directa en el shell *de* `ls -l &`.

```
runback "ls -l"
```

Algunos sistemas cuentan con un servicio denominado `biff` que permite la notificación de correo. Cuando un usuario está conectado al sistema y recibe correo, `biff` lo notifica de alguna manera, como puede ser una señal audible o la presentación de un mensaje. (En el mundo UNIX, se dice que el autor original de este programa tenía un perro, Biff, que ladraba a todos los carteros.) El programa 2.10 presenta el código de un programa en C, `simplebiff.c`, que emite una señal audible en la terminal a intervalos regulares de tiempo si el usuario, `oshacker`, tiene correo.

El programa notifica al usuario que tiene correo enviando un carácter `ctrl-g` (ASCII 7) al dispositivo de error estándar. Muchas terminales manejan la recepción de un `ctrl-g` produciendo una señal audible breve. El programa continúa produciendo esta señal cada 10 segundos hasta que es terminado o se elimina el archivo de correo.

Ejemplo 2.14

El siguiente comando da inicio a la ejecución de `simplebiff`.

```
simplebiff &
```

Programa 2.10: Programa simple para notificar a `oshacker` que hay correo.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#define MAILFILE "/var/mail/oshacker"
#define SLEEPTIME 10

void main(void)
{
   int mailfd;

   for( ; ; ) {
      if ( (mailfd = open(MAILFILE, O_RDONLY)) != -1) {
         fprintf(stderr,"%s", "\007");
         close(mailfd);
      }
      sleep(SLEEPTIME);
   }
}
```

Programa 2.10

El programa 2.10 ilustra la forma en que pueden trabajar los demonios. Por lo común, el correo se guarda en el directorio `/var/mail` o en `/var/spool/mail`. Todo el correo que no ha sido leído por el usuario se halla contenido en un archivo, en alguno de estos directorios, que tiene el mismo nombre de inicio de sesión del usuario. Si `oshacker` tiene correo, entonces la llamada a `open` para abrir `/var/mail/oshacker` tendrá éxito, de lo contrario `open` fallará. Si el archivo existe, entonces el usuario tiene correo que no ha leído y el programa hará que se emita una señal audible. En cualquier caso, el programa duerme y luego repite el proceso indefinidamente.

El programa 2.10 no es muy general porque el nombre del usuario, el directorio del correo y el tiempo durante el cual el programa debe dormir están especificados de antemano. La manera de obtener el nombre del usuario aprobada para POSIX es hacer, primero, una llamada a `getuid` para determinar el ID del usuario y después llamar a `getpwuid` para obtener el nombre de la sesión. La llamada al sistema `stat` proporciona más información sobre un archivo sin todas las complicaciones de `open`.

La estructura de directorio para el correo cambia de un sistema a otro, de modo que el usuario debe determinar la ubicación de los archivos de correo del sistema para poder hacer uso de `simplebiff`. El programa debe permitir que el usuario especifique el directorio en la línea comando, o depender de la información específica del sistema comunicada por las variables de ambiente, si es que esta información se encuentra disponible. El estándar POSIX define diversas variables de ambiente importantes, tales como `MAIL`, `MAILCHECK`, `MAILDIR` y `MAILPATH`. Si estas variables guardan información, entonces el programa debe hacer uso de

ésta. La sección 2.9 estudia la forma en que un programa puede tener acceso a estas variables y la manera en que éstas pueden comunicar información específica del sistema. En la sección 2.14 se analiza un `biff` más conveniente. En la sección 5.5 se modifica el programa `biff` para que haga uso de señales con objeto de habilitar o deshabilitar la notificación.

2.9 Ambiente de los procesos

El biff del programa 2.10 ilustra la importancia que reviste el hacer uso de información dependiente del sistema para hacer una implantación de la aplicación independiente del mismo. Ningún programador experimentado distribuirá un programa que requiera que el usuario cambie las rutas de acceso a los directorios codificadas de antemano para que éste funcione. Las *variables de ambiente* proporcionan un mecanismo para hacer uso de información específica del sistema o del usuario para establecer los valores por omisión dentro del programa.

Una *lista de variables de ambiente* está formada por un arreglo de apuntadores a cadenas de caracteres de la forma *nombre* = *valor*. El *nombre* señala una variable de ambiente. El *valor* especifica una cadena de caracteres. Cada aplicación interpreta la lista de ambiente en una forma que depende de la aplicación. POSIX.2 especifica el significado de las variables de ambiente que aparecen en la tabla 2.5.

Variable	Significado
`HOME`	directorio de trabajo inicial del usuario
`LANG`	localidad cuando no está especificada por `LC_ALL` o `LC_*`
`LC_ALL`	sustitución del nombre de la localidad
`LC_COLLATE`	nombre de la localidad para recopilar información
`LC_CTYPE`	nombre de la localidad para clasificación de caracteres
`LC_MONETARY`	nombre de la localidad para edición monetaria
`LC_NUMERIC`	nombre de la localidad para edición numérica
`LC_TIME`	nombre de la localidad para información fecha/hora
`LOGNAME`	nombre de la contraseña de inicio de sesión asociada con un proceso
`PATH`	prefijos de ruta de acceso para encontrar el ejecutable
`TERM`	tipo de terminal para enviar la salida
`TZ`	información sobre uso horario

Tabla 2.5: Variables de ambiente POSIX.2 y sus significados.

La variable externa environ apunta a la lista de ambiente del proceso cuando comienza la ejecución de éste. La variable environ está definida por

```
extern char **environ
```

Las cadenas de caracteres en la lista de ambiente pueden aparecer en cualquier orden. Si el proceso ha sido iniciado por `execl`, `execlp`, `execv` o `execvp`, entonces éste hereda la lista de ambiente que tiene el proceso justo antes de la ejecución de `exec`. Las llamadas a `execle` y `execve` permiten especificar la lista de ambiente.

Ejemplo 2.15

El siguiente programa en C imprime el contenido de su lista de ambiente y finaliza después de hacerlo.

```
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

void main(int argc, char *argv[])
{
    int i;

    printf("La lista de ambiente para %s es\n", argv[0];
    for (i = 0; environ[i] != NULL; i++)
        printf("environ[%d]: %s\n", i, environ[i]);
    exit(0);
}
```

Para determinar si cierta variable de ambiente está definida haga uso de `getenv`.

SINOPSIS

```
#include <stdlib.h>

char *getenv(const char *name);
```

POSIX.1, Spec 1170

El nombre de la variable de ambiente debe ser pasado como una cadena de caracteres. La función `getenv` devuelve `NULL` si dicha variable no está definida. Si tiene un valor, `getenv` devuelve un apuntador a la cadena de caracteres que contiene el valor. En caso de usar `getenv` varias veces debe tenerse cuidado de copiar en un *buffer* la cadena devuelta por la función. Algunas implantaciones de `getenv` emplean un *buffer* estático para poner en él las cadenas devueltas por la función, y sobrescriben el *buffer* en cada llamada.

POSIX.2 indica que el *shell* sh debe utilizar la variable de ambiente `MAIL` como ruta de acceso a los buzones en lo que respecta al correo que llega, siempre y cuando la variable `MAILPATH` no tenga ningún valor. (Vea la sección 2.14 para más información sobre `MAIL` y `MAILPATH`.)

Ejemplo 2.16

El siguiente fragmento de código pone en `mailp` *el valor de* `MAIL` *si éste se encuentra definido, de lo contrario hace que* `MAILP` *sea igual a* `MAILPATH`. *En cualquier otro caso, pone en* `mailp` *un valor preestablecido.*

```
#include <stdlib.h>
#define MAILDEFAULT "/var/mail"
char *mailp = NULL;

if (getenv("MAILPATH") == NULL)
   mailp = getenv("MAIL");
if (mailp == NULL)
   mailp = MAILDEFAULT;
```

La primera llamada a `getenv` del ejemplo 2.16 sólo verifica la existencia de `MAILPATH`, de modo que no es necesario copiar el valor devuelto en un *buffer* aparte antes de volver a llamar a `getenv` otra vez.

No confunda las variables de ambiente con constantes predefinidas como `MAX_CANON`. Estas constantes están definidas en los archivos de encabezado con un `#define`. Sus valores son constantes y se los conoce al momento de la compilación. Para saber si existe la definición de alguna constante, utilice la directiva del compilador `#ifndef`, como se hace en el programa 2.3. Por el contrario, las variables de ambiente son dinámicas y sus valores no son conocidos al momento de la compilación.

POSIX.2 especifica una utilería `env` para examinar el ambiente y modificarlo con el fin de ejecutar otro comando.

SINOPSIS

```
env [-i] [name=value] ... [utility [argument ...]]
```

POSIX.2, Spec 1170

Cuando es invocado sin argumentos, el comando `env` muestra el ambiente en uso. Los argumentos opcionales `[name=value]`, indican las variables de ambiente que serán modificadas. El argumento opcional `utility` señala el comando que será ejecutado bajo el ambiente modificado. El argumento opcional `-i` indica que el ambiente especificado por los argumentos deberá reemplazar al ambiente en uso para los fines que la ejecución de `utility` implique. La utilería `env` no modifica el ambiente del *shell* que la ejecuta.

Ejemplo 2.17

El siguiente es un listado de la salida generada por la ejecución de `env` *en una máquina que corre bajo Sun Solaris 2.3. El guión (-) indica la continuación de una línea larga.*

```
DISPLAY=:0.0
FONTPATH=/home/robbins/vttool/crttool-2.0/fonts:-
/usr/local/lib/font/85dpi
HELPPATH=/usr/openwin/lib/locale:/usr/openwin/lib/help
HOME=/data1/robbins
HZ=100
LD_LIBRARY_PATH=/usr/openwin/lib:/opt/tex/lib
LOGNAME=robbins
MAIL=/var/mail/robbins
MANPATH=/usr/openwin/share/man:/opt/SUNWspro/man:/usr/man
OPENWINHOME=/usr/openwin
PATH=/usr/openwin/bin:/opt/SUNWspro/bin:/usr/bin:/usr/sbin:-
/usr/ccs/bin:/usr/bin/X11:/opt/gnu/bin:/opt/tex/bin:-
/opt/bin:/data1/robbins/bin:/usr/local/bin:/usr/ccs/lib:.
PROCDIR=/usr/local/bin
PWD=/data1/robbins
SHELL=/bin/csh
TERM=sun-cmd
```

```
TEXFONTS=.:/usr/local/tex/fonts/tfm:/usr/local/src/dvips/PStfms:-
/usr/local/src/dlx/benchmarks/tex/latex:
/data/src/tex/unix3.0/ams/amsfonts/pk/pk300
TZ=US/Central
USER=robbins
XDVIFONTS=/usr/local/tex/fonts/pk
XENVIRONMENT=/data1/robbins/.Xdefaults
XFILESEARCHPATH=/usr/openwin/lib/locale/%L/%T/%N%S:-
/usr/openwin/lib/%T/%N%S
WINDOW_TERMIOS=
TERMCAP=sun-cmd:te=\E[>4h:ti=\E[>4l:tc=sun:
```

2.10 Terminación de procesos en UNIX

Cuando termina un proceso, el sistema operativo recupera los recursos asignados al proceso terminado, actualiza las estadísticas apropiadas y notifica a los demás procesos la defunción. La terminación puede ser *normal* o *anormal*. Las actividades realizadas durante la terminación del proceso incluyen la cancelación de los temporizadores y señales pendientes, la liberación de los recursos de memoria virtual así como la de otros recursos del sistema retenidos por el proceso, tales como los bloqueos, y el cierre de archivos abiertos. El sistema operativo registra el estado del proceso y el uso de los recursos, y notifica al padre en respuesta a una llamada `wait` al sistema. Cuando un proceso termina, sus hijos huérfanos son adoptados por el proceso `init`, cuyo ID de proceso es 1. Si el padre no espera a que el proceso termine, entonces éste se convierte en un *zombie*. El proceso `init` espera de manera periódica a los hijos para librarse de los *zombies* huérfanos.

La terminación normal se presenta cuando existe un `return` de *main*, un regreso implícito de *main* (el procedimiento *main* no es capaz de terminar), una llamada a la función de C `exit`, o una llamada a la función del sistema `_exit`. La función `exit` de C llama a los manejadores de la salida del usuario y puede proporcionar tareas de limpieza adicionales antes de invocar la llamada al sistema `_exit`.

SINOPSIS

```
#include <unistd.h>

void _exit(int status);
```

POSIX.1, Spec 1170

SINOPSIS

```
#include <stdlib.h>

void exit(int status);
```

ISO C, POSIX.1, Spec 1170

Tanto `exit` como `_exit` toman un parámetro entero, `status`, que indica el estado de terminación del programa. Para indicar una terminación normal, haga uso de un valor 0 para `status`. Los valores de `status` distintos de 0 y definidos por el programador indican errores. El ejemplo 2.8 de la página 48 ilustra la manera en que el padre puede determinar el valor de `status` cuando espera por los hijos. El valor es devuelto por una llamada a `exit` o `_exit` en cualquier parte del programa, o por el empleo de un `return` en el programa principal. `exit` y `_exit` devuelven un valor al padre incluso si la declaración de *main* indica que el valor devuelto es de tipo `void`.

La función `atexit` de C instala un manejador de terminación definido por el usuario. Cuando se llama a `exit` estos manejadores son ejecutados sobre la base de que el último en ser instalado es el primero en ser ejecutado. Para instalar varios manejadores, haga uso de varias llamadas a `atexit`. Cuando la terminación es normal, el proceso llama a los manejadores como parte de la terminación, siendo el primero instalado el último en ser ejecutado.

El programa 2.11 tiene un manejador de salida denominado `show_times`, el cual hace que se imprima en el dispositivo de error estándar, las estadísticas sobre el tiempo utilizado por el programa y sus hijos antes de que el programa termine. La función `times` devuelve información sobre el tiempo con un número de pulsos del reloj. La función `show_times` convierte el tiempo en segundos dividiendo éste entre el número de pulsos registrados por segundo, dato que se obtiene al llamar a `sysconf`. (El capítulo 6 proporciona un estudio más completo del tiempo en UNIX.)

Programa 2.11: Programa con manejador de salida que imprime el uso de la CPU.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/times.h>
#include <limits.h>

static void show_times(void)
{
   status tms times_inf;
   double ticks;

   if ((ticks = (double) sysconf(_SC_CLK_TCK)) < 0)
      perror("No es posible determinar el número de pulsos por segundo");
   else if (times(&times_info) < 0)
      perror("No es posible obtener información sobre la hora")
   else {
      fprintf(stderr, "Tiempo de usuario:            %8.3f segundos\n",
        (times_info.tms_utime/ticks);
      fprintf(stderr, "Tiempo del sistema:           %8.3f segundos\n",
        (times_info.tms_stime/ticks);
      fprintf(stderr, "Tiempo de usuario del hijo:   %8.3f segundos\n",
        (times_info.tms_cutime/ticks);
```

```
        fprintf(stderr, "Tiempo del sistema del hijo:  %8.3f segundos\n",
          (times_info.tms_cstime/ticks);


    void main(void)
    {
        if (atexit(show_times)) {
            fprintf(stderr, "No es posible instalar el manejador de salida
                             show_times\n");
            exit(1);
        }

          /* el resto del programa principal va aquí */
}
```

Programa 2.11

Un proceso también puede ser terminado anormalmente ya sea invocando a la llamada `abort` o procesando una señal que cause la terminación. La señal puede ser generada por un evento externo (como un `ctrl-c` proveniente del teclado) o por un error interno, por ejemplo un intento por tener acceso a una localidad de memoria ilegal. Si un proceso aborta anormalmente, tal vez se produzca un vaciado de memoria (*core dump*), y no se llamará a los manejadores de terminación instalados por el usuario.

2.11 Secciones críticas

Imagine el siguiente escenario. Suponga que un sistema de cómputo cuenta con una impresora a la que tienen acceso directo todos los procesos del sistema. Cada vez que un proceso desea imprimir algo, lo hace con un `write` al dispositivo de impresión. ¿Qué apariencia presentará la salida si más de un proceso intenta escribir en la impresora al mismo tiempo? Recuerde que a cada proceso sólo se le permite un quantum fijo del tiempo del procesador. Si el proceso comienza a escribir, pero lo que debe imprimir es mucho y su quantum termina, entonces se escoge otro proceso. La impresión resultante tendrá mezclada la salida de muchos procesos, lo que constituye una característica poco deseable.

El problema con el escenario anterior es que los procesos intentan simultáneamente acceder a un recurso compartido –un recurso que debe ser utilizado sólo por un proceso a la vez. Esto es, la impresora requiere el *acceso exclusivo* por los procesos del sistema. La parte del código donde cada proceso intenta tener acceso a un recurso compartido recibe el nombre de *sección crítica*. Se debe hacer algo para asegurar la *exclusión mutua* de los procesos mientras éstos ejecutan sus secciones críticas.

Un método para alcanzar la exclusión mutua es utilizar un mecanismo de bloqueo en el que el proceso adquiere un candado que excluye a todos los demás procesos antes de comenzar

la ejecución de su sección crítica. Cuando el proceso termina la sección crítica, libera el candado. Otro enfoque es encapsular los recursos compartidos de modo que se asegure un acceso exclusivo. Por lo común el manejo de las impresoras lo hace un solo proceso (el demonio de impresión), que es el que en realidad tiene acceso a la impresora. Cuando otro proceso desea imprimir, éste hace una solicitud al demonio de impresión enviando un mensaje. El demonio pone la solicitud en una cola y procesa un mensaje a la vez.

Además de los dispositivos, los archivos y las variables compartidas, existen otros muchos recursos compartidos. Las tablas y otra información del núcleo son compartidas entre los procesos que administran el sistema. Un sistema operativo grande tiene muchas partes diversas, con secciones críticas que posiblemente se traslapan. Cuando se modifica una de estas partes, es difícil determinar si la modificación afectará negativamente otras partes sin una compresión completa de todo el sistema operativo. Para reducir la complejidad de las interacciones internas, algunos sistemas operativos emplean un diseño *orientado a objetos*. Las tablas compartidas y otros recursos son encapsulados como objetos, con funciones de acceso bien definidas. La única manera de tener acceso a una de estas tablas es a través de dichas funciones, las cuales tienen el acceso exclusivo construido internamente. En un sistema distribuido, el objeto de interfaz utiliza mensajes. Los cambios en otros módulos de un sistema orientado a objetos no tienen el mismo impacto que en uno donde el acceso no es controlado.

En apariencia, el enfoque orientado a objetos es similar a los demonios descritos en la sección 2.8, pero estructuralmente estos enfoques pueden ser muy diferentes. No hay ningún requerimiento para que los demonios encapsulen los recursos. Ellos pueden pelear por tener acceso a estructuras de datos compartidos de manera no controlada. El enfoque orientado a objetos requiere que las estructuras de datos sean encapsuladas y que el acceso a ellas se haga a través de interfaces controladas. Los demonios pueden ser implantados utilizando un diseño orientado a objetos, pero ellos no tienen por qué serlo.

2.12 Ejercicio: cadenas de procesos

Esta sección extiende las cadenas de procesos del ejemplo 2.5. La cadena es un vehículo para experimentar con `wait` y con el hecho de compartir dispositivos. En este ejercicio los procesos comparten el dispositivo de error estándar y presentan secciones críticas cuando escriben en dicho dispositivo. En capítulos posteriores se extiende este ejercicio a las secciones críticas que involucran a otros dispositivos (capítulo 3) y a la simulación de una red *token-ring* (capítulo 4).

El programa 2.12 crea una cadena de procesos. El programa toma sólo un argumento, que especifica el número de procesos que debe crearse. Antes de salir, cada proceso imprime su valor *i*, su ID de proceso, el ID de su padre, y su ID como hijo. No hay ningún `wait`, de modo que el proceso padre puede terminar antes que sus hijos, lo que hará que éstos sean adoptados por el proceso `init` (el cual tiene un ID igual a 1). Como resultado, algunos de los procesos pueden indicar que el ID de su padre es 1.

Programa 2.12: Programa simple para generar una cadena de procesos.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
/*
 *   Programa muestra en C para generar una cadena de procesos.
 *   Utilice este programa con un argumento de línea comando que indique
 *   el número de procesos en la cadena, después de la creación de ésta.
 *   Cada proceso se identifica a sí mismo con su ID de proceso,
 *   el ID de proceso de su padre, y el ID de proceso de su hijo.
 *   Después, cada hijo termina su ejecución.
 */

void main (int argc, char *argv[])
{
   int    i;
   pid_t  childpid;        */ ID de proceso del hijo creado con un fork */
   int    n;       */ número total de procesos en la cadena */

        */ se verifica que la línea comando tenga un número válido de argumentos */
   if (argc ! = 2){
      fprintf(stderr, "Uso: %s procesos\n", argv[0]);
      exit(1);
   }
   n = atoi(argv[1]);

   childpid = 0;
   for(i = 1; i < n; ++1)
      if (childpid = fork())
         break;
   if (childpid == -1) {
      perror("Error al ejecutar fork");
      exit(1);
   }

   fprintf(stderr,
       "i:%d ID del proceso :%ld ID del padre :%ld ID del hijo :%ld\n",
       i, (long)getpid(), (long)getppid()),(long)childpid);
   exit(0);
}
```

Programa 2.12

Todos los procesos que están en la cadena creada por el programa 2.12 tienen los mismos dispositivos de entrada, salida y error estándar. El `fprintf` para el error estándar es una sección crítica. Este ejercicio analiza algunas implicaciones de las secciones críticas.

- Ejecute el programa 2.12 y observe los resultados para diferentes números de procesos.
- Indique los ID de los procesos en el diagrama de la figura 2.4 en el caso de una ejecución con una línea de comando de valor 4.
- Experimente con valores distintos del argumento de la línea comando para encontrar el mayor número de procesos que el programa puede generar. Observe qué fracción de ellos son adoptados.
- Coloque una instrucción `sleep(10)` directamente antes de la última instrucción `fprintf` del programa 2.12. ¿Cuál es el número máximo de procesos que puede generarse en este caso?
- Ponga la última instrucción `fprintf` del programa 2.12 en un ciclo. Ejecute el ciclo `k` veces. Ponga un `sleep(m)` dentro del ciclo, después de `fprintf`. Pase los valores de `k` y `m` como argumentos de línea comando. Ejecute el programa con varios valores distintos de `n`, `k` y `m`, y observe los resultados.
- Modifique el programa 2.12 colocando una llamada al sistema `wait` antes de la última instrucción `fprintf`. ¿Cómo afectó esto a la salida generada por el programa?
- Modifique el programa 2.12 reemplazando la última instrucción `fprintf` con un ciclo que lea `nchars` (n caracteres) provenientes del dispositivo de entrada estándar, un carácter a la vez, y los ponga en un arreglo, `mybuf`. Los valores de `n` y `nchars` deben ser pasados al programa como argumentos de línea comando. Después del ciclo, ponga un carácter `'\0'` en la entrada `nchars` del arreglo, de modo que éste contenga una cadena de caracteres terminada en un *null*. Imprima en el dispositivo de error estándar con una sola instrucción `fprintf` el ID del proceso, después dos puntos y a continuación la cadena de caracteres contenida en `mybuf`. Ejecute el programa para varios valores de `n` y `nchars`, y observe los resultados. Continúe tecleando hasta que todos los procesos terminen.

2.13 Ejercicio: abanicos de procesos

Los ejercicios de esta sección amplían la estructura de abanico del ejemplo 2.6 mediante el desarrollo de un servicio sencillo de procesamiento por lotes. Las modificaciones en la sección 8.8 conducirán a una aplicación para la administración del número de ejercicios o administrador de licencias (*license manager*) para un programa de aplicación. El ejecutable para el procesamiento por lotes tiene el nombre `runsim`. Escriba un programa de prueba, `testsim`, para probar este servicio.

2.13.1 Especificación de `runsim`

El programa `runsim` ejecuta hasta `pr_limit` procesos a la vez. El `runsim` toma exactamente un argumento de línea comando que especifica el número máximo de ejecuciones simultáneas. Para implantar `runsim`, siga el esbozo que se presenta a continuación. Las funciones de la biblioteca, así como las llamadas al sistema que se ha sugerido, aparecen entre paréntesis.

- Compruebe que el argumento de la línea comando sea apropiado, e imprima un mensaje de error si éste es incorrecto.
- Inicie `pr_limit` con el argumento de línea comando. La variable `pr_limit` especifica el número máximo de hijos que se permite ejecutar al mismo tiempo.
- La variable `pr_count` contiene el número de hijos activos. El valor inicial de ésta debe ser 0.
- Ejecute el siguiente ciclo principal hasta que se alcance el fin de archivo en el dispositivo de entrada estándar:
  * Si `pr_count` es igual a `pr_limit`, el proceso debe esperar a que el hijo termine (`wait`) y disminuir el valor de `pr_count`.
  * Lea una línea del dispositivo de entrada estándar (`fgets`) hasta de `MAX_CANON` caracteres y ejecute el programa especificado por dicha línea creando un hijo (`fork, makeargv, execvp`).
  * Incremente `pr_count` para rastrear el número de hijos activos.
  * Verifique si alguno de los hijos ha terminado (`waitpid` con la opción `WNOHANG`). Disminuya el valor de `pr_count` por cada hijo que haya terminado su tarea.
- Después de encontrar un fin de archivo en el dispositivo de entrada estándar, espere a que todos los demás hijos terminen (`wait`) y finalice la ejecución del programa.

2.13.2 Prueba del programa `runsim`

Escriba un programa de prueba, `testsim`, que tome dos argumentos de línea comando: el tiempo de suspensión y el factor de repetición. El programa `testsim` queda suspendido por un tiempo específico y luego escribe un mensaje con su ID de proceso en el dispositivo de error estándar. Utilice `runsim` para ejecutar varias copias del programa `testsim`.

Escriba un archivo de prueba, `testing.data`, que contenga los comandos a ejecutar. Por ejemplo, el contenido del archivo puede ser

```
testsim 5 10
testsim 8 10
testsim 4 10
testsim 13 6
testsim 1 12
```

Ejecute el programa introduciendo un comando tal como

```
runsim 2 < testing.data
```

2.14 Ejercicio: `biff` sencillo

Este ejercicio requiere un sistema que cuente con un servicio de correo UNIX estándar más que un sistema de correo complejo.

- Averigue dónde se guarda el correo y cambie la especificación de directorio de correo de la variable de ambiente `MAILDIR` de modo que usted pueda hacer referencia a su propio archivo de correo. (Si está poco familiarizado con las variables de ambiente, consulte la sección A.7.)
- Cambie el valor de `MAILFILE` del programa 2.10 a la ruta de acceso y el nombre de su archivo de correo.
- Compile el archivo fuente del programa 2.10 para generar un ejecutable `mybiff`.
- Ejecute `mybiff` como proceso de fondo introduciendo `mybiff &`.
- Envíese correo y confirme que recibe la notificación correspondiente.
- Modifique `mybiff` de modo que haga uso de `stat` para determinar cuándo ha llegado correo nuevo.
- Observe que `mybiff` sigue ejecutándose incluso después del *log-off* puesto que el programa fue iniciado como proceso en plano secundario. Ejecute `ps -a` para determinar el ID del proceso `mybiff`. Termine el proceso `mybiff` con el comando `kill -KILL pid`. Asegúrese que `mybiff` ha desaparecido introduciendo otro comando `ps -a`.
- Modifique el programa `mybiff` para determinar al usuario mediante una llamada a `getuid` seguida de otra a `getpwuid`. Haga una prueba para ver si `mybiff` sigue trabajando.
- Modifique el programa `mybiff` de modo que éste tenga la siguiente resumen:

  ```
  mybiff [-s n] [-p PATHname]
  ```

  El [ ] en el resumen indica argumentos de línea comando opcionales. El primer argumento indica el tiempo de suspensión. Si `-s n` no es proporcionada en la línea comando, entonces debe emplearse el valor de `SLEEPTIME` como valor preestablecido. El parámetro `-p PATHname` indica la ruta de acceso del directorio de correo del sistema. Si esta opción no se indica en la línea comando, entonces debe emplearse el valor de la variable de ambiente `MAIL` como valor preestablecido. Utilice `getopt` para analizar los argumentos de la línea comando.
- POSIX.2 define tres variables de ambiente para el manejo del correo: `MAIL`, `MAILCHECK` y `MAILPATH`. Modifique el programa `mybiff` de modo que haga uso de estas variables como valores preestablecidos si es que ellas se encuentran definidas. Las versiones abreviadas de las definiciones de POSIX.2 son las siguientes:

MAIL es la ruta de acceso del archivo buzón del usuario para fines de notificación de llegada de correo. El usuario es informado de la llegada del correo sólo si MAIL está definido pero no MAILPATH.

MAILCHECK es un entero decimal que especifica (en segundos) con cuánta frecuencia debe verificar el *shell* la llegada del correo a los archivos señalados por las variables MAILPATH o MAIL. El valor preestablecido es 600 segundos.

MAILPATH es una lista de rutas de acceso y mensajes opcionales, separados por dos puntos. Si esta variable se encuentra definida, el *shell* informa al usuario si se crea algunos de los archivos que aparecen en la lista o si cambia la fecha de modificación de cualquiera de ellos. Cada ruta de acceso puede estar seguida de un % y una cadena de caracteres que será impresa en el dispositivo de error estándar cuando cambie la fecha de modificación. Si el carácter % en la ruta de acceso va precedido de una diagonal invertida, entonces éste se toma como una literal % de la ruta de acceso.

Modifique biff de modo que ahora el programa utilice las variables MAIL, MAILCHECK y MAILPATH en forma congruente con la manera en que el *shell* las interpreta. Si se proporciona un valor para sleeptime o PATHname en los argumentos de la línea comando, entonces éstos deben sustituir a los dados por las variables de ambiente.

2.15 Ejercicio: News biff

El programa biff informa al usuario del correo que llega. Tal vez, el usuario también quiera estar informado sobre cambios en otro archivos, como los Internet News. Si un sistema es un servidor de noticias, entonces probablemente organiza los artículos como archivos individuales cuya ruta de acceso presenta el nombre del grupo de noticias.

Ejemplo 2.18

Un sistema mantiene sus archivos de noticias en el directorio /var/spool/news. *El artículo 1034 del grupo de noticias* comp.os.research *se localiza en el siguiente archivo.*

```
/var/spool/news/comp/os/research/1034
```

El siguiente ejercicio desarrolla una utilería para avisar cuándo cambia cualquiera de los archivos que aparecen en una lista de archivos.

2.15.1 Analizando un sólo archivo

- Escriba una función lastmod que devuelva la fecha en que el archivo fue modificado por última vez. El prototipo para lastmode es

```
time_t lastmod(char *pathname) ;
```

Utilice `stat` para determinar la fecha del último acceso. `time_t` es tiempo transcurrido, en segundos, desde el primero de enero de 1970, 00:00:00 UTC. La función `lastmod` devuelve `-1` si existe algún error y pone en `errno` el número de error generado por `stat`. Sea cuidadoso. POSIX permite que `errno` sea un macro, de modo que el programa no debe trabajar directamente con `errno`.

- Escriba un programa que tome una ruta de acceso como argumento de línea comando y llame a `lastmod` para determinar la fecha en que se modificó por última vez el archivo correspondiente. Utilice la función ctime para imprimir el valor de `time_t` en un formato apropiado. Compare los resultado con los obtenidos con `ls -l`.

- Escriba una función, `convertnews`, que convierta el nombre de un grupo de noticias en una ruta de acceso completamente calificada. El prototipo de `convertnews` es

```
char *convertnews(char *newsgroup) ;
```

Si la variable de ambiente `NEWSDIR` está definida, utilice ésta para determinar la ruta de acceso. En caso contrario, emplee como directorio preestablecido `/var/spool/news`. (Llame a `getenv` para determinar si la variable está o no definida.) Por ejemplo, si *newsgroup* es `comp.os.research` y `NEWSDIR` no está definida, la ruta de acceso es

```
/var/spool/news/comp/os/research
```

La función `convertnews` reserva espacio para retener la cadena de caracteres producto de la conversión y regresa un apuntador a dicho espacio. (Un error común es regresar un apuntador a una variable automática definida dentro de `convertnews`.) No modifique `newsgroup` en `convertnews`. Si existe un error, `convertnews` debe devolver un apuntador `NULL`.

- Escriba un programa que tome como argumentos de línea comando `newsgroup` y `sleeptime`. Imprima la fecha de la última modificación del `newsgroup` y luego implante un ciclo de la siguiente manera:
  * Suspenda la ejecución por un tiempo igual a `sleeptime`.
  * Haga una prueba para determinar si `newsgroup` ha sido modificado.
  * Si el directorio `newsgroup` ha sido modificado, imprima un mensaje con el nombre `newsgroup` y la fecha de modificación.

Pruebe el programa en varios grupos de noticias (*newsgroups*). Envíe mensajes a un grupo de noticias local y verifique que el programa funciona. El directorio `newsgroup` puede ser modificado tanto por la llegada de nuevas noticias como por la flecha de vencimiento. La flecha de vencimiento de muchos sistemas de noticias ocurre a la medianoche.

2.15.2 Creación de un objeto lista

Implante una lista enlazada para guardar en ella las rutas de acceso de los grupos de noticias y las fechas en que éstos fueron modificados por última vez. La lista debe ser con enlaces simples (*singly linked*) y ésta es similar a la desarrollada en la sección 2.2. Mantenga un apuntador para la cabeza de la lista, otro para la cola y uno más para recorrerla. Este último apuntador es un apuntador a la localidad que contiene la posición actual dentro de la lista. Incluya las siguientes operaciones en la lista:

- `add_data` inserta un nodo al final de la lista. La rutina recibe como argumentos la ruta de acceso, el nombre del archivo y la fecha de modificación. Haga uso de `malloc` para asignar el espacio para el nodo. El prototipo de `add_data` es

  ```
  int add_data(char *pathname, *char monitorname,
                  time_t mtime) ;
  ```

  `pathname` es el nombre de la ruta de acceso completamente calificado del grupo de noticias. Para el grupo `comp.os.research` éste será

  ```
  /var/spool/news/comp/os/research
  ```

  `monitorname` es el nombre del grupo de noticias que proporciona el usuario (por ejemplo, `comp.os.research`). La variable `mtime` es la fecha de la última modificación del archivo que corresponde a `pathname`. `add_data` devuelve 0 si tiene éxito, de lo contrario devuelve `-1`.

- `rewind_list` inicia el apuntador de recorrido de modo que apunte al primer nodo de la lista. El prototipo de `rewind_list` es

  ```
  int rewind_list(void);
  ```

  `rewind_list` devuelve 0 si la lista no se encuentra vacía, y `-1` si lo está.

- update_list verifica el tiempo de modificación del grupo de noticias del nodo designado por el apuntador de recorrido. El prototipo de `update_list` es

  ```
  int update_list(char *modifyname,time_t *modtime) ;
  ```

  `modifyname` es el nombre que el usuario emplea para identificar un grupo de noticias particular (por ejemplo, `comp.os.research`), y `*modtime` contiene la fecha en que se hizo la última modificación. El programa que llama a esta rutina debe proporcionar el espacio para las variables a las que apuntan `modifyname` y `modtime`. Si el archivo ha sido modificado después del tiempo guardado en el nodo, éste se actualiza y la rutina devuelve 1. Si el archivo no ha sido modificado, entonces se devuelve un 0. Si la entrada no está en la lista, el valor devuelto es `-1`.

Escriba un programa que pruebe que el objeto lista funciona antes de implantar `newsbiff`.

Escriba un programa principal que tome dos argumentos en la línea comando: el nombre de un archivo y un valor de `sleeptime`. El archivo contiene la lista de grupos de noticias que

debe vigilarse. Añada cada grupo de noticias al objeto lista. Después genere un ciclo y haga lo siguiente:

- Suspenda la ejecución del programa por un tiempo igual a `sleeptime` segundos.
- Llame a `update_list` en un ciclo hasta que todo el objeto lista haya sido recorrido. Si un grupo de noticias ha sido modificado, imprima un mensaje con el nombre y la fecha de modificación y haga sonar la campana.

2.16 Lecturas adicionales

El libro *The Design of the UNIX Operating System* de Bach [6] presenta la implantación de procesos en el System V. *The Design and Implementation of the 4.3BSD UNIX Operating System* de Leffer *et al.* [59] analiza la implantación de procesos para BSD UNIX. Los dos libros contienen análisis excelentes y detallados de la forma en que se implanta los sistemas operativos reales. *Operating Systems: Design and Implementation* de Tanenbaum [89] desarrolla una implantación completa de un sistema operativo similar a UNIX, denominado MINIX. Desafortunadamente todos estos libros están basados en versiones anteriores de UNIX y no son aplicables de manera específica a POSIX.

Existen libros más generales sobre sistemas operativos, tales como *Operating Systems Concepts*, cuarta edición, por Silberschatz y Galvin [80] y *Modern Operating Systems* de Tanenbaum [92] que abordan el modelo de procesos. Los dos libros presentan estudios de caso sobre UNIX y Mach, un sistema operativo de microkernel bastante conocido. En este momento es bueno hacer una comparación entre estos dos sistemas. *PS to Operating Systems* por Dowdy y Lowery [26] se encuentra enfocado a aspectos de desempeño y modelos analíticos.

Capítulo 3

Archivos

Quizá la mejor característica de UNIX es su independencia de los dispositivos. La interfaz de dispositivos uniforme a través de los descriptores de archivo permite la utilización de las mismas llamadas de E/S para terminales, discos, cintas, audio e incluso comunicación por red. Este capítulo explora la E/S independiente del dispositivo de UNIX, la E/S con bloqueo y sin bloqueo, y archivos especiales como los de entubamiento o *pipes*. Este capítulo también trata acerca de los filtros, los entubamientos de procesos (*pipelines*), el redireccionamiento, el recorrido de directorios y la herencia de descriptores.

Un *dispositivo periférico* es una pieza de *hardware* conectada a un sistema de cómputo. Entre los dispositivos periféricos más comunes se incluye discos, cintas, CD ROM, pantallas, teclados, impresoras, ratones e interfaces de red. Los programas de los usuarios realizan el control y la E/S para estos dispositivos haciendo llamadas a ciertos programas del sistema operativo denominados *controladores de dispositivo (device drivers)*. El controlador oculta detalles sobre la forma en que funciona el dispositivo y lo protege del uso no autorizado. La forma en que funcionan los dispositivos del mismo tipo cambia sustancialmente, de modo que incluso una máquina de un solo usuario necesita de controladores de dispositivo para poder ser utilizable. Ciertos sistemas operativos proporcionan *controladores de pseudodispositivos* para simular dispositivos como las terminales. Los dispositivos pseudoterminales, por ejemplo, simplifican el manejo del inicio de sesión remota en sistemas de cómputo sobre red o vía módem.

Algunos sistemas operativos proporcionan llamadas específicas para cada tipo de dispositivo soportado por ellos. En este caso, el programador de sistemas debe aprender un conjunto complejo de llamadas para el control del dispositivo, y tiene que volver a escribir los programas para utilizar dispositivos diferentes. UNIX ha simplificado mucho la interfaz de dispositivos del programador al proporcionar un acceso uniforme a muchos dispositivos a través de cinco llamadas al sistema: `open`, `close`, `read`, `write` e `ioctl`. Todos los dispositivos están representados por archivos denominados *archivos especiales*, los cuales están localizados en el

directorio `/dev`. De esta forma, se logra el acceso a los archivos en disco y otros dispositivos con nombre. Un *archivo regular* es sólo un archivo de datos ordinario en disco. Un *archivo especial por bloque* representa un dispositivo con características similares a un disco. El controlador de dispositivo transfiere la información contenida en un dispositivo especial de bloque en bloques o porciones, y lo más usual es que los dispositivos brinden apoyo a la capacidad de recuperar un bloque desde cualquier parte del dispositivo. Un *archivo especial de caracteres* representa un dispositivo con características similares a las de un teclado. El dispositivo aparece como representando un flujo de bytes a los que se debe tener acceso en forma secuencial.

Las secciones 3.1 y 3.2 abordan los aspectos básicos de la organización y representación de archivos en UNIX. La sección 3.3 explica las diferencias entre los apuntadores y descriptores de archivos e ilustra la forma en que los programas emplean estos identificadores para tener acceso a archivos abiertos. La sección 3.4 muestra cómo utilizar el redireccionamiento para cambiar durante la ejecución el significado del identificador de un descriptor de archivo en particular. La sección 3.5 introduce el entubamiento para la comunicación entre procesos. La sección 3.6 presenta las llamadas al sistema `read` y `write` con bloque, mientras que la sección 3.7 examina situaciones donde las llamadas sin bloque mejoran la eficiencia. La sección 3.8 trata acerca de la llamada al sistema `select` para vigilar varios descriptores de archivos. La sección 3.9 estudia los FIFO, y la 3.10 presenta el dispositivo de audio para ilustrar el uso de `ioctl`. El capítulo termina con cuatro secciones de ejercicios que ilustran aspectos diferentes del acceso y control de dispositivos.

3.1 Directorios y rutas de acceso

Los datos y los programas se organizan en *archivos* para guardarlos de manera permanente en un disco. En lugar de especificar la posición de cada archivo en el disco, el usuario proporciona un nombre de archivo y el sistema operativo hace una traslación a la posición del archivo físico. El sistema operativo mantiene en *directorios* una relación de los nombres de los archivos con su ubicación física.

Cuando los discos eran pequeños, el uso de una tabla sencilla de nombres de archivo y posición era una representación suficiente. Los discos de mayor capacidad requieren una organización más flexible y ahora muchos sistemas utilizan directorios con estructura de árbol. Esta representación aparece con bastante naturalidad cuando los mismos directorios son archivos. La figura 3.1 muestra una organización con estructura de árbol denominada *sistema de archivo*. En este árbol los nodos cuadrados son los directorios, y la / designa al *directorio raíz* del sistema de archivo. El directorio raíz está en la parte superior del árbol del sistema de archivo, y todo lo demás se encuentra debajo de él.

El directorio indicado como `dirA` en la figura 3.1 contiene los archivos `my1.dat`, `my2.dat` y `dirB`. El archivo `dirB` recibe el nombre de *subdirectorio* de `dirA` dado que `dirB` está debajo de `dirA` en el árbol del sistema de archivo. Nótese que `dirB` también contiene un archivo, `my1.dat`.

Es evidente que el nombre del archivo no basta para especificar de manera unívoca un archivo, puesto que el nombre `my1.dat` aparece varias veces en el sistema de archivo. La

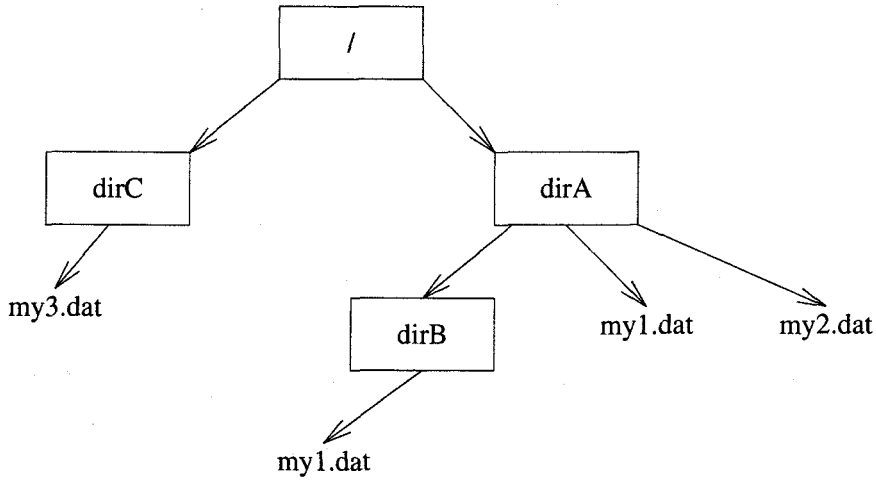


Figura 3.1: Estructura de árbol de un sistema de archivo.

manera de especificar de modo único un archivo es con una *ruta de acceso completa* o *absoluta*. La ruta de acceso absoluta especifica todos los nodos del árbol de directorio que hay que recorrer desde la raíz hasta el archivo buscado. UNIX emplea una / para separar los nombres de los directorios que están en la trayectoria. (Desafortunadamente MS-DOS emplea \ para separar los nombres de los directorios, de modo que esta característica confunde a las personas que emplean los dos sistemas operativos.) El archivo `my1.dat` en `dirA` de la figura 3.1 tiene como ruta de acceso completa `/dirA/my1.dat`, mientras que `my1.dat` en `dirB` tiene como ruta de acceso completa `/dirA/dirB/my1.dat`.

No es necesario que un programa proporcione en cada caso las rutas de acceso completas. En todo momento, los procesos tienen un directorio asociado con ellos que es el *directorio de trabajo activo*, el cual se emplea para resolver la ruta de acceso. Si el nombre de una ruta no comienza con /, entonces se supone que ésta comienza en la ruta de acceso del directorio de trabajo activo. De aquí que las rutas de acceso que no comiencen con una / algunas veces reciben el nombre de *ruta de acceso relativa*. Un punto (.) indica el directorio activo, y dos puntos (..) el que se encuentra encima de él. El directorio raíz tiene asociados con él tanto el punto como los dos puntos. El directorio de trabajo activo asociado con el *shell* de inicio de sesión del usuario es el *directorio hogar* del usuario.

Ejemplo 3.1

Después de ejecutar el siguiente comando, el directorio de trabajo activo para el proceso shell es `/dirA/dirB`.

```
cd /dirA/dirB
```

Si el directorio activo de trabajo es `/dirA/dirB`, puede hacerse la referencia al archivo `my1.dat` que se encuentra en el directorio `dirA` de la figura 3.1 como `../my1.dat`, mientras que la del archivo `my1.dat` que se encuentra en el directorio `dirB` puede hacerse como `my1.dat`,

`./my1.dat` o `../dirB/my1.dat`. La referencia al archivo `my3.dat` en `dirC` se hace como `../../dirC/my3.dat`.

La función `getcwd` de la biblioteca de C devuelve la ruta de acceso del directorio de trabajo activo.

SINOPSIS

```
#include <unistd.h>

extern char *getcwd(char *buf, size_t size);
```

POSIX.1, Spec 1170

El parámetro `size` especifica la longitud máxima del nombre de la ruta de acceso que puede emplear quien la llama. Si el nombre es mayor que este máximo, `getcwd` devuelve `-1` y hace `errno` igual a `ERANGE`. Si `buf` no es `NULL`, `getcwd` copia el nombre en `buf`. Si `buf` es `NULL`, POSIX.1 establece que el comportamiento de `getcwd` es indefinido. En algunas implantaciones, `getcwd` hace uso de `malloc` para crear un *buffer* dónde poner la ruta de acceso. Después el programa puede llamar a `free` para liberar el espacio. En cualquier caso, si `getcwd` tiene éxito, devuelve un apuntador al nombre de la ruta de acceso.

Ejemplo 3.2

El siguiente programa imprime la ruta de acceso del directorio de trabajo activo.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <limits.h>
#ifndef PATH_MAX
#define PATH_MAX 255
#endif

void main(void)
{
   char mycwd[PATH_MAX + 1];
   if (getcwd(mycwd, PATH_MAX) == NULL) {
       perror("Could not get current working directory");
       exit(1);
   }
   printf("Current working directory: %s\n", mycwd);
   exit(0);
}
```

`PATH_MAX` es una constante opcional de POSIX.1 que especifica la longitud máxima del nombre de la ruta de acceso para la implantación. `PATH_MAX` puede o no estar definida en `limits.h`. Las constantes opcionales de POSIX.1 pueden ser omitidas de `limits.h` si sus valores son indeterminados aunque mayores que el mínimo requerido por POSIX. Para `MAX_PATH`, la constante `_POSIX_PATH_MAX` especifica que la implantación debe acomodar

longitudes de nombres de rutas de acceso cuya longitud sea de al menos 255 caracteres. Un vendedor puede permitir que el valor de PATH_MAX dependa de la cantidad de espacio de memoria disponible sobre una instancia específica de una implantación particular. La función pathconf proporciona el valor real de PATH_MAX.

Ejemplo 3.3

El siguiente programa imprime el directorio de trabajo activo después de determinar la longitud máxima del nombre de la ruta de acceso de la implantación relativa al directorio raíz.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main(void)
{
   char *mycwdp;
   long maxpath;
   if ((maxpath= pathconf("/", _PC_PATH_MAX)) == -1) {
      perror("Could not determine maximum pathname length");
      exit(1);
   } else if ((mycwdp = (char *) malloc(maxpath+1)) == NULL) {
      perror("Could not allocate space for directory pathname");
      exit(1);
   } else if (getcwd(mycwdp, maxpath) == NULL) {
      perror("Could not get current working directory");
      exit(1);
   }
   printf("Current working directory: %s\n", mycwdp);
   exit(0);
}
```

Existen tres funciones que determinan información específica de la implantación.

SINOPSIS

```
#include <unistd.h>

long sysconf(int name);
long pathconf(const char *path, int name);
long fpathconf(int fildes, int name);
```

POSIX.1, Spec 1170

La función sysconf devuelve los valores límites de todo el sistema, tales como el número de pulsos de reloj por segundo (_SC_CLK_TCK) o el número máximo de procesos permitido por usuario (_SC_CHILD_MAX). El programa 2.11 hace uso de sysconf para calcular el número de segundos en que se ejecuta un programa. Las funciones pathconf y fpathconf informan sobre límites asociados con un archivo o directorio en particular. Utilícese fpathconf para abrir un archivo. El ejemplo 3.3 emplea pathconf para encontrar la longitud máxima del nombre de la ruta de acceso, comenzando desde el directorio raíz.

3.1.1 Lectura de directorios

La estructura de árbol del sistema de archivo oculta los archivos y directorios que se encuentran en niveles más profundos del árbol. Para localizar un archivo o grupo de archivos en un sistema, ejecute desde el *shell* la utilería `find`.

SINOPSIS

```
find path ... [operand_expression]
```

POSIX.2, Spec 1170

El `operand_expression` puede ser bastante complicado. Para más detalles, consúltese las páginas del manual.

Ejemplo 3.4

El siguiente comando imprime una lista de todos los archivos que tienen extensión `.c` *en el directorio activo o por debajo de él y que son mayores* 10 *bloques de* 512 bytes.

```
find . -name "*.c" -size +10 -print
```

La búsqueda se lleva a cabo sólo en los directorios que cuentan con los permisos apropiados. La especificación `*.c` *está delimitada por comillas para que el* shell *no la expanda.*

Utilice `readdir` y las llamadas relacionadas con ella para tener acceso a la información del directorio desde un programa en C.

SINOPSIS

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *filename);
struct dirent *readdir(DIR *dirp);
void rewinddir(DIR *dirp);
int closedir(DIR *dirp);
```

POSIX.1, Spec 1170

SINOPSIS

```
#include <sys/types.h>
#include <dirent.h>

long telldir(DIR *dirp);
void seekdir(DIR *dirp, long loc);
```

Spec 1170

La función `opendir` proporciona un identificador de bloque al directorio para las demás funciones de directorio. Cada llamada subsecuente a `readdir` devuelve un apuntador a una estructura que contiene información sobre la siguiente entrada del directorio. La función

`readdir` devuelve `NULL` cuando llega al final del directorio. Utilice `rewinddir` para volver a empezar, o `closedir` para terminar. La función `telldir` devuelve el desplazamiento dentro del directorio de la entrada en uso, mientras que `seekdir` reposiciona la entrada en uso del directorio a la posición especificada por `loc`. Para regresar a una entrada particular, haga uso de `telldir` para guardar el desplazamiento de ésta y luego emplee `seekdir`. (Este enfoque funciona siempre y cuando el directorio no sea reorganizado para consolidar el espacio libre entre las llamadas a estas funciones.) Las funciones `telldir` y `seekdir` no son parte de POSIX. Si es necesario, los servicios que brindan éstas pueden alcanzarse al guardar un nombre de archivo, utilizar `rewinddir` y después ejecutar `readdir` en un lazo hasta que se encuentre el archivo deseado.

El programa 3.1 muestra los nombres de los archivos contenidos en el directorio cuyo nombre de ruta de acceso se pasa al programa como un argumento de la línea comando. Para recorrer un árbol de directorio como en `find`, utilice `stat` para determinar qué archivos son directorios. La sección 3.11 desarrolla un programa para recorrer el árbol de directorio.

Programa 3.1: Programa para imprimir la lista de archivos contenidos en un directorio.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <errno.h>

void main(int argc, char *argv[])
{
    DIR *dirp;
    struct dirent *direntp;

    if (argc != 2) {
       fprintf(stderr,"Usage: %s directory_name\n", argv[0]);
       exit(1);
    }

    if ((dirp = opendir(argv[1])) == NULL) {
       fprintf(stderr, "Could not open %s directory: %s\n",
              argv[1], strerror(errno));
       exit(1);
    }
    while ( (direntp = readdir( dirp )) != NULL )
       printf("%s\n", direntp->d_name );

   closedir(dirp);
   exit(0);
}
```

Programa 3.1

El programa 3.1 no asigna una variable `struct dirent` para guardar la información del directorio. En lugar de esto, `readdir` mantiene internamente una estructura `struct dirent` y devuelve un apuntador hacia ella. La devolución de una estructura implica que `readdir` no es segura en cuanto a la ejecución de hilos de control *(threads)*. POSIX.1c propone `readdir_r` como una versión segura en cuanto a hilos de control *(threads)* para dar soporte a los hilos de control *(threads)* de POSIX.

Las entradas del directorio no son de tamaño fijo, y la estructura `struct dirent` está diseñada para acomodar estas entradas de longitud variable. La estructura `struct dirent` incluye el miembro

```
char d_name[];
```

el cual es una cadena de caracteres terminada en *null* no mayor de `NAME_MAX` caracteres. Una implantación puede hacer que otros miembros de la estructura `struct dirent` (como el número del inodo) sean visibles al usuario. Nótese que `struct dirent` no refleja la forma en que realmente están representadas en el disco las entradas del directorio, sino sólo la estructura que `readdir` emplea para guardar la entrada que ha recuperado.

3.1.2 Búsqueda de rutas de acceso

En UNIX el usuario ejecuta un programa escribiendo el nombre de la ruta de acceso del archivo que contiene el ejecutable. Muchos de los programas y utilerías más utilizados no se encuentran en el directorio de trabajo del usuario (por ejemplo, `vi`, `cc`). Imagine el lector los inconvenientes que habría si los usuarios tuviesen que saber la ubicación de todos los ejecutables del sistema para poder hacer uso de ellos. Afortunadamente, UNIX tiene un método para buscar los ejecutables de una manera sistemática. El sistema busca el ejecutable en todos los directorios que aparecen en la variable de ambiente `PATH`, utilizando cada uno de ellos, uno a la vez, como directorio de trabajo. `PATH` contiene los nombres completos de las rutas de acceso de los directorios más importantes, separados por dos puntos.

Ejemplo 3.5

La línea que sigue es un valor típico de la variable de ambiente `PATH`.

```
/usr/bin:/etc:/usr/local/bin:/usr/ccs/bin:/home/robbins/bin:.
```

La especificación del ejemplo 3.5 dice que cuando se ejecuta un comando primero se busca en `/usr/bin`. Si el comando no se encuentra allí, entonces se examina el directorio `/etc`, y así sucesivamente.

Debe recordarse que la búsqueda no se hace en ningún subdirectorio de los directorios indicados por la variable `PATH`, a menos que éstos aparezcan explícitamente en ella. Si existe alguna duda sobre qué versión de un programa en particular es la que se está ejecutando, use `which` para obtener el nombre completo de la ruta de acceso del ejecutable.

Ejercicio 3.1

Después de la instalación de un sistema operativo nuevo, al programador le gustaría comenzar a emplear el nuevo compilador ANSI C, más que el compilador, que no era ANSI, que utilizaba con anterioridad. Sin embargo, parece ser que el compilador no acepta los prototipos de funciones. ¿Qué enfoque debe emplearse para resolver este problema?

Respuesta:

Ejecute `which cc` para determinar el nombre completo de la ruta de acceso del compilador de C predeterminado. Por ejemplo, el lugar estándar para el compilador ANSI de Sun en un sistema Sun que corre bajo el control de Solaris 2 es `/opt/SUNWspro/bin`, pero el compilador anterior puede permanecer tal vez en `/usr/ucb`. Si `/usr/ucb` aparece antes que `/opt/SUNWspro/bin` en la variable de ambiente `PATH`, entonces `cc` se refiere al anterior compilador BSD. (Nota: `which` no está disponible en algunos sistemas.) Ejecute `find` para buscar los ejecutables `cc` comenzando desde el nodo raíz:

```
find / -name "cc" -print
```

No lo intente en un sistema que tiene mucho espacio en disco, ya que la búsqueda puede durar mucho tiempo y producir bastantes mensajes autorización denegada *(permission denied)*.

Es común que los programadores creen un directorio `bin` como un subdirectorio de sus directorios hogar para guardar los ejecutables. El directorio `/home/robbins/bin` en la variable `PATH` del ejemplo 3.5 es un ejemplo de ese directorio `bin`. El directorio `bin` aparece antes del punto (.), el directorio de trabajo, en la ruta de búsqueda, lo que conduce al problema considerado en el ejercicio siguiente.

Ejercicio 3.2

Un usuario desarrolla un programa, `calhit`, en el subdirectorio `progs` de su directorio hogar y pone un copia del ejecutable en el directorio `bin` de la misma cuenta. Después, el usuario modifica `calhit` en el directorio `progs`, sin copiarlo en el directorio `bin`. ¿Qué sucede?

Respuesta:

El resultado depende del valor de la variable de ambiente `PATH`. Si el `PATH` del usuario es configurado de la manera usual, el *shell* buscará primero en el directorio `bin` y ejecutará la versión anterior del programa.

El lector debe resistir la tentación de poner el punto (.) al inicio del `PATH` a pesar del problema mencionado en el ejercicio 3.2. La especificación del `PATH` es considerada como un riesgo que afecta la seguridad y puede conducir a resultados extraños cuando se ejecuta programas locales en lugar de los programas estándar del sistema, que tienen el mismo nombre.

3.1.3 Sistema de archivo de UNIX

Cuando se da formato a un disco, el proceso divide el disco físico en regiones denominadas *particiones*. Cada partición puede tener un sistema de archivo asociado con ella. Un sistema

de archivo está formado por un árbol de directorio. Por otra parte, un sistema de archivo puede montarse en cualquier nodo del árbol de directorio de cualquier otro sistema de archivo. El nodo que está en la parte superior de un sistema de archivo recibe el nombre de *raíz* del sistema de archivo. El directorio raíz de un proceso (denotado por `/`) es el directorio más alto en la jerarquía del árbol al que el proceso puede tener acceso. Todos los nombres completos de las rutas de acceso en UNIX comienzan en el directorio raíz `/`. La figura 3.2 muestra la raíz típica de un árbol de sistema de archivo.

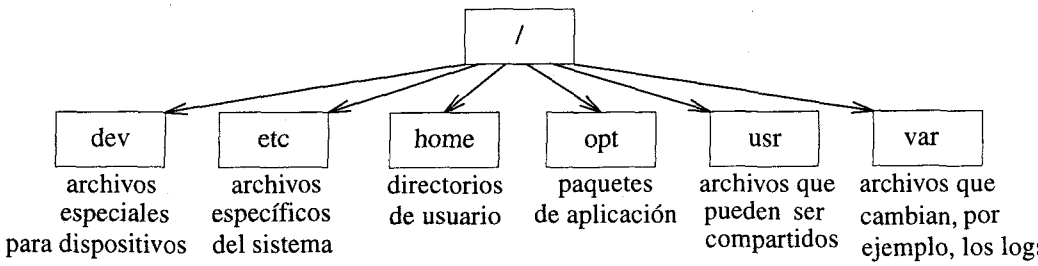


Figura 3.2: Estructura de un sistema de archivo UNIX típico.

Existen algunos directorios estándares que aparecen en la raíz del sistema de archivo. El directorio `/dev` contiene las especificaciones de los dispositivos (archivos especiales) del sistema. El directorio `/etc` guarda archivos que contienen información relacionada con la red, las cuentas y otras bases de datos que son específicas de la máquina. El directorio `/home` es el directorio preestablecido para las cuentas de los usuarios. El directorio `/opt` es un sitio estándar para aplicaciones en el System V Release 4. Los archivos `include` se encuentran en el directorio `/usr/include`. El directorio `/var` contiene archivos del sistema que cambian y pueden crecer de manera arbitraria y volverse muy grandes (por ejemplo, los archivos bitácoras o el correo cuando llega, pero antes de que sea leído por el usuario).

3.2 Representación de archivos en UNIX

Un archivo en UNIX tiene una descripción que se guarda en una estructura denominada *inodo*. Este inodo contiene información sobre el tamaño del archivo, su localización, el propietario del archivo, fecha de creación, fecha de última modificación, permisos y varias cosas más. Muchos archivos de usuario son *archivos ordinarios*. Los directorios también están representados como archivos y tienen asociados con ellos un inodo. Los dispositivos están representados por *archivos especiales*. Los *archivos especiales de caracteres* son utilizados para representar dispositivos como terminales, mientras que los *archivos especiales por bloque* son empleados para dispositivos de disco. Para la comunicación entre procesos se hace uso de los *archivos especiales FIFO*.

La figura 3.3 muestra la estructura de un inodo para un archivo común. Además de la información que describe al archivo, el inodo contiene apuntadores hacia los primeros bloques de datos del archivo. Si éste es grande, el apuntador indirecto contiene un apuntador a un conjunto de apuntadores a los demás bloques de datos. Si el archivo sigue siendo grande, el

apuntador doble indirecto contiene un apuntador a un conjunto de apuntadores indirectos a bloques de apuntadores directos a los bloques de datos después de los bloques de datos indirectos. Si en realidad el archivo es enorme, el apuntador indirecto triple contiene un apuntador a un bloque de apuntadores indirectos dobles. La palabra *bloque* puede significar varias cosas (incluso dentro de UNIX). En este contexto, un bloque tiene típicamente 8K. El número de bytes en un bloque siempre es una potencia de 2.

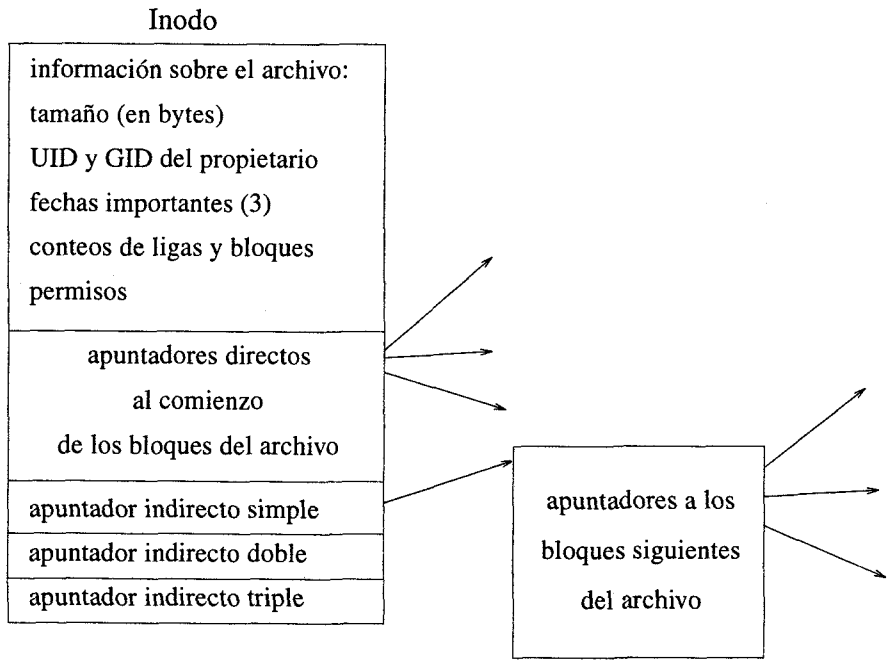


Figura 3.3: Estructura esquemática de un archivo UNIX.

Ejercicio 3.3:

Suponga que el inodo es de 128 bytes, los apuntadores son de 4 bytes, y la información sobre el estado del archivo puede ocupar hasta 68 bytes. El tamaño del bloque es 8K. ¿Cuánto espacio existe para los apuntadores en el inodo (o nodo i)? ¿Cuán grande puede ser un archivo representado por apuntadores directos? ¿Indirectos? ¿Indirectos dobles? ¿Indirectos triples?

Respuesta:

Los apuntadores indirectos simples, dobles y triples ocupan cada uno 4 bytes, así que existen 128 – 68 – 12 = 48 bytes disponibles para 12 apuntadores directos. El tamaño del inodo así como el del bloque son características que dependen del sistema. Con apuntadores directos es posible representar archivos con un tamaño hasta de 8 192 × 12 = 98 304 bytes. Si el tamaño del bloque es de 8K, el apuntador indirecto simple direcciona un bloque de 8K que puede tener hasta 8 192 ÷ 4 = 2 048 apuntadores a bloques de datos. Por tanto, el apuntador indirecto simple proporciona la capacidad de direccionamiento de 2 048 × 8 192 = 16 777 216 bytes, o 16 megabytes, de infor-

mación adicional. El direccionamiento indirecto doble proporciona 2 048 × 2 048 apuntadores con la capacidad de tener acceso a 32 gigabytes adicionales. El direccionamiento indirecto triple proporciona 2 048 × 2 048 × 2 048 apuntadores, los que permiten direccionar 64 terabytes adicionales (cantidad que debe ser suficiente para un futuro cercano :-)).

Para recuperar la información contenida en el inodo de un archivo en particular, utilice la llamada al sistema `stat`.

SINOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *path, struct stat *buf);
int fstat(int fildes, struct stat *buf);
```

POSIX.1, Spec 1170

SINOPSIS

```
#include <sys/stat.h>

int lstat(const char *path, struct stat *buf);
```

Spec 1170

Para abrir archivos haga uso de `fstat` en lugar de `stat`. La llamada `lstat` recupera información para ligas (links) simbólicas, las cuales son descritas en la sección 3.2.2. Puesto que las ligas simbólicas no forman parte todavía de POSIX, la función `lstat` no es parte de POSIX. La estructura `struct stat` contiene los miembros siguientes:

```
mode_t    st_mode;     /* Modo del archivo (vea mknod(2)) */
ino_t     st_ino;      /* Número de inodo */
dev_t     st_dev;      /* ID del dispositivo que contiene */
                       /* una entrada de directorio para este archivo */
dev_t     st_rdev;     /* ID del dispositivo */
                       /* Esta entrada está definida sólo para */
                       /* archivos especiales de caracteres o de bloque */
nlink_t   st_nlink;    /* Número de ligas */
uid_t     st_uid;      /* ID de usuario del propietario del archivo */
gid_t     st_gid;      /* ID de grupo del grupo del archivo */
off_t     st_size;     /* Tamaño del archivo en bytes */
time_t    st_atime;    /* Fecha del último acceso */
time_t    st_mtime;    /* Fecha de la última modificación */
time_t    st_ctime;    /* Fecha del último cambio al estado del archivo */
                       /* Tiempo medido en segundos desde */
                       /* 00:00:00 UTC, Enero 1, 1970 */
long      st_blksize;  /* Tamaño preferencial del bloque de E/S */
long      st_blocks;   /* Número de bloques st_blksize asignado */
```

Ejemplo 3.6

El siguiente programa hace uso de `stat` *para determinar si el nombre de archivo pasado a él por la línea comando es un directorio. Si el nombre es el de un directorio, también se imprime la fecha en que éste fue modificado por última vez.*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>

void main(int argc, char *argv[])
{
    struct stat statbuf;

    if (argc != 2) {
       fprintf(stderr, "Usage: %s filename\n", argv[0]);
       exit(1);
    }
    if (stat(argv[1], &statbuf) == -1) {
       fprintf(stderr, "Could not get stat on file %s: %s\n",
               argv[1], strerror(errno));
       exit(1);
    }
    if (statbuf.st_mode & S_IFDIR) {
        printf("%s is a directory: ", argv[1]);
        printf("last modified at %s\n", ctime(&statbuf.st_mtime));
    } else
        printf("%s is not a directory\n", argv[1]);
    exit(0);
}
```

La representación de archivos como estructuras de árbol es bastante eficiente para archivos pequeños, además de ser muy flexible si el tamaño del archivo cambia. Cuando se crea un archivo, el sistema operativo encuentra bloques libres sobre el disco dónde poner los datos. Las consideraciones de desempeño dictan que los bloques deben estar lo más cerca unos de otros sobre el disco para reducir el tiempo de búsqueda. Se requiere 20 veces más tiempo para leer un archivo de 16 megabytes cuando los bloques de datos se hallan dispuestos aleatoriamente sobre el disco, que cuando éstos son contiguos.

3.2.1 Representación del directorio

En UNIX, un *directorio* es un archivo que contiene una serie de nombres de archivo e inodos que se corresponden. El inodo no contiene el nombre del archivo. Cuando un programa hace referencia a un archivo mediante el nombre de una ruta de acceso, el sistema operativo recorre el árbol del sistema de archivo para encontrar el nombre del archivo y el número del inodo en

el directorio apropiado. Una vez que se tiene el número de inodo, el sistema operativo puede determinar el resto de la información sobre el archivo mediante el acceso al inodo. (Por razones de desempeño, esto no es tan simple como parece, ya que el sistema operativo pone tanto las entradas de directorio como las de los inodos en la memoria principal.)

Una representación de directorio que contiene sólo nombres y números de inodos ofrece varias ventajas:

- El cambio de nombre de un archivo requiere únicamente la modificación de la entrada en el directorio. El archivo puede moverse de un directorio a otro al trasladar la entrada del directorio, siempre y cuando el movimiento mantenga al archivo en la misma partición o segmento. El comando `mv` emplea esta técnica para mover archivos a otros sitios dentro del mismo sistema de archivo. Puesto que una entrada del directorio se refiere a un inodo que está en la misma partición que la propia entrada del directorio, `mv` no puede emplear este enfoque para mover archivos entre particiones diferentes.
- Sólo es necesario tener una copia física del archivo en el disco, pero el archivo puede tener varios nombres, o el mismo nombre en directorios diferentes. De nuevo, deben hacerse todas estas referencias sobre la misma partición física.
- La longitud de las entradas del directorio es variable debido a que el nombre del archivo tiene una longitud variable. Las entradas del directorio son pequeñas, puesto que la mayor parte de la información se mantiene en el inodo. Entonces, el manejo de estructuras pequeñas de longitud variable puede hacerse con eficiencia. Las estructuras de los inodos son de longitud fija.

3.2.2 Enlaces o ligas

Un enlace o *liga* es una asociación entre el nombre de un archivo y su inodo. UNIX tiene dos tipos de enlaces o ligas: duras y simbólicas (también conocidas como suaves). Las entradas del directorio son *ligas duras* porque ellas ligan directamente los nombres de los archivos con los inodos. Las ligas suaves o *simbólicas* utilizan el archivo como un apuntador a otro nombre de archivo. La diferencia entre las ligas duras y suaves no es tan notoria.

La entrada de un directorio corresponde a una liga dura simple, pero un inodo puede tener varias de estas ligas. Cada inodo contiene el número de ligas duras para él, esto es, el número total de entradas de directorio que contienen el número del inodo. El sistema operativo crea una entrada de directorio nueva y le asigna un inodo para cada archivo de nueva creación. Con el comando `ln` o con la llamada al sistema `link` los usuarios pueden crear ligas duras adicionales para el archivo. Una liga dura adicional asigna sólo una entrada más en el directorio, pero no hace uso de más espacio sobre el disco. La nueva liga dura hace que la entrada que corresponde al conteo de ligas en el inodo aumente. Recuerde que la liga dura es sólo una entrada de directorio. En ocasiones se hace referencia a las ligas duras sólo como ligas.

Cuando un usuario borra un archivo con el comando `rm` o con la llamada al sistema `unlink`, el sistema operativo borra la entrada correspondiente en el directorio y reduce el conteo de ligas en el inodo. Lo anterior no libera el inodo ni los bloques de datos correspondientes, a menos que el conteo de ligas sea 0.

La figura 3.4 muestra una entrada de directorio para un archivo con nombre `name1` en el directorio `/dirA`. El archivo hace uso del inodo 12345. El inodo tiene una liga y el primer bloque de datos es el bloque 23567. Todos los datos del archivo están contenidos en este bloque, los cuales están representados por el texto breve que aparece en la figura.

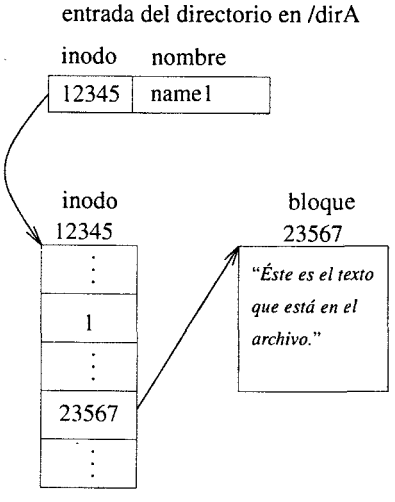


Figura 3.4: Directorio, inodo y bloque de datos para un archivo simple.

Ejemplo 3.7

El comando siguiente crea una entrada en `dirB` *que contiene un apuntador al inodo de* `/dirA/name1`.

```
ln /dirA/name1 /dirB/name2
```

La figura 3.4 muestra los directorios antes de la ejecución del comando `ln` del ejemplo 3.7, mientras que la figura 3.5 muestra los directorios después de ésta. El comando `ln` crea una liga (entrada de directorio) que hace referencia al mismo inodo que `dirA/name1`. No se requiere de espacio adicional en disco, excepto posiblemente si la nueva entrada de directorio aumenta el número de bloques de datos necesarios para guardar la información del directorio. Ahora el inodo tiene dos ligas.

Ejercicio 3.4

¿Qué sucede si un usuario utiliza un editor para hacer un cambio pequeño al archivo `/dirA/name1` de la figura 3.5? ¿Cómo están relacionados los archivos `/dirA/name1` y `/dirB/name2`?

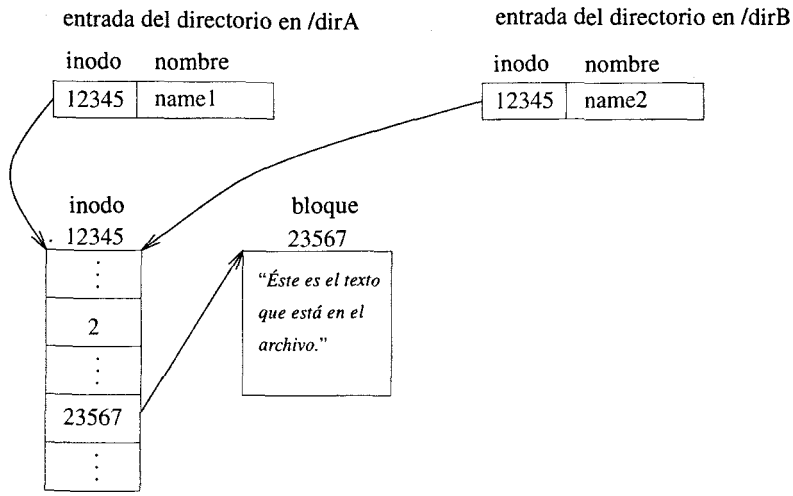


Figura 3.5: Dos ligas duras al mismo archivo de la figura 3.4.

Respuesta:

El resultado depende de la forma en que el editor trabaje. Muchas versiones de `vi` no cambian el inodo, de modo que se aplica la situación de la figura 3.5. Cuando algunos editores modifican `/dirA/name1`, ellos copian el archivo en memoria, modifican la imagen en memoria y escriben ésta en el disco con el nombre `/dirA/name1`. El editor puede hacer esto desligando primero `/dirA/name1` y luego creando un archivo nuevo asociado con el nombre `/dirA/name1`. Es así como `/dirA/name1` se refiere al nuevo archivo y `/dirB/name2` al archivo anterior. Existen dos inodos diferentes, cada uno con una liga, como se ilustra en la figura 3.6. Algunos editores emplean `rename` (que es similar a `mv`) para hacer una copia de respaldo del archivo que van a editar en lugar de ejecutar un `unlink`. Tales editores pueden cambiar el nombre `/dirA/name1` por `/dirA/name1.bak` antes de escribir el nuevo archivo. En este caso, `/dirA/name1.bak` y `/dirB/name2` tendrán el mismo inodo con dos ligas, como se muestra en la figura 3.7.

El comportamiento ilustrado en el ejercicio 3.4 puede resultar indeseable. Existe otro tipo de liga, la liga simbólica, que se comporta de manera diferente. Una *liga simbólica* es un archivo que contiene el nombre de otro archivo o directorio. La referencia al nombre de una liga simbólica hace que el sistema operativo localice el nodo que corresponde a dicha liga. Lo anterior supone que los bloques de datos que corresponden al inodo contienen otro nombre de archivo. Entonces, el sistema operativo busca el nuevo nombre de archivo en un directorio y continúa siguiendo la cadena hasta que finalmente encuentra una liga dura y un archivo real. POSIX.1 no incluye ligas simbólicas, pero históricamente muchas implantaciones las incluyen y Spec 1170 también las especifica.

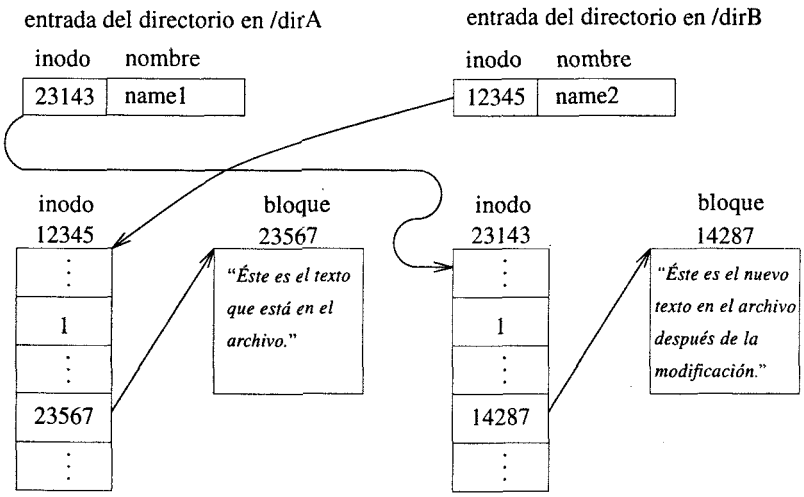


Figura 3.6: Situación al editar un archivo. El archivo original tiene el inodo 12345 y dos ligas duras antes de ser editado.

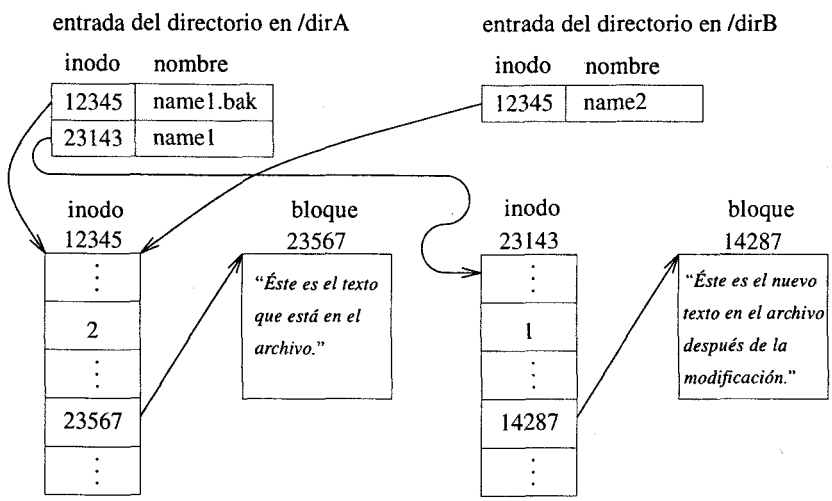


Figura 3.7: Situación después de editar un archivo con un editor que crea una copia de respaldo.

La creación de ligas simbólicas se hace utilizando el comando `ln` con la opción `-s` o con una llamada al sistema `symlink`.

Ejemplo 3.8

Si se inicia con la situación mostrada en la figura 3.4, el siguiente comando crea la liga simbólica `/dirB/name2` *como se muestra en la figura 3.8.*

```
ln -s /dirB/name2 /dirA/name1
```

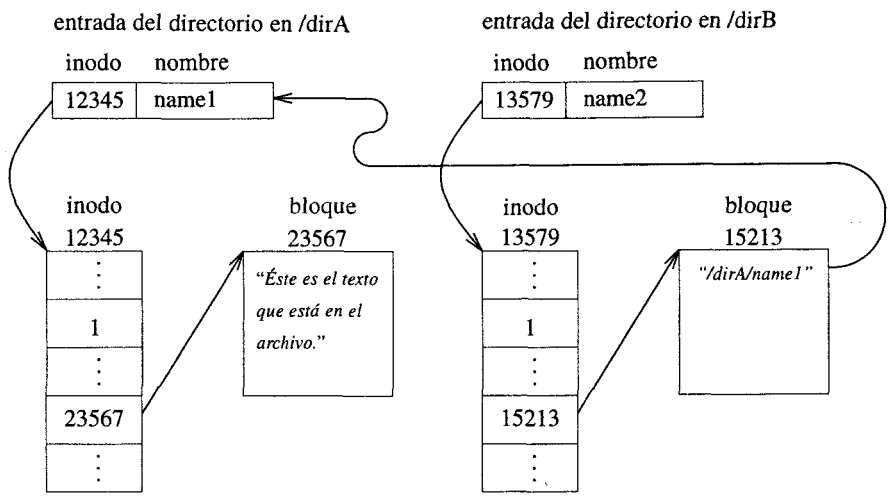


Figura 3.8: Archivo ordinario y liga simbólica hacia él.

A diferencia del ejercicio 3.4, el comando `ln` del ejemplo 3.8 hace uso de un nuevo inodo, 13579, para la liga simbólica. Los inodos contienen información sobre el tipo de archivo que ellos representan (esto es, ordinario, directorio, especial o liga simbólica), de modo que el inodo 13579 contiene información que indica que es una liga simbólica. Esta liga requiere al menos un bloque de datos. En este caso se emplea el bloque 15213. El bloque de datos contiene el nombre del archivo con el que está ligado `/dirB/name2`, en este caso, `/dirA/name1`. El nombre puede ser completo, como en el ejemplo, o relacionado con su propio directorio.

Ejercicio 3.5

Suponga que `/dirA/name1` es un archivo ordinario y que `/dirB/name2` es una liga simbólica a `/dirA/name1`, como en la figura 3.8. ¿Cómo están relacionados los archivos `/dirB/name2` y `/dirA/name1` después de que un usuario edita, cambia y guarda `/dirA/name1` como en el ejercicio 3.4?

Respuesta:

`/dirA/name1` ahora se refiere a un inodo diferente, pero `/dirB/name2` hace referencia al nombre `/dirA/name1`, con lo que ambos siguen haciendo referencia al mismo archivo de la figura 3.9. El conteo de ligas en el inodo cuenta sólo las ligas duras, no las simbólicas. Cuando el editor desliga `/dirA/name1`, el sistema operativo borra el archivo con el inodo 12345. Si otros intentan editar `/dirB/name2` en el lapso durante el cual se desliga `/dirA/name1` pero aún no se crea, lo que obtienen es un error.

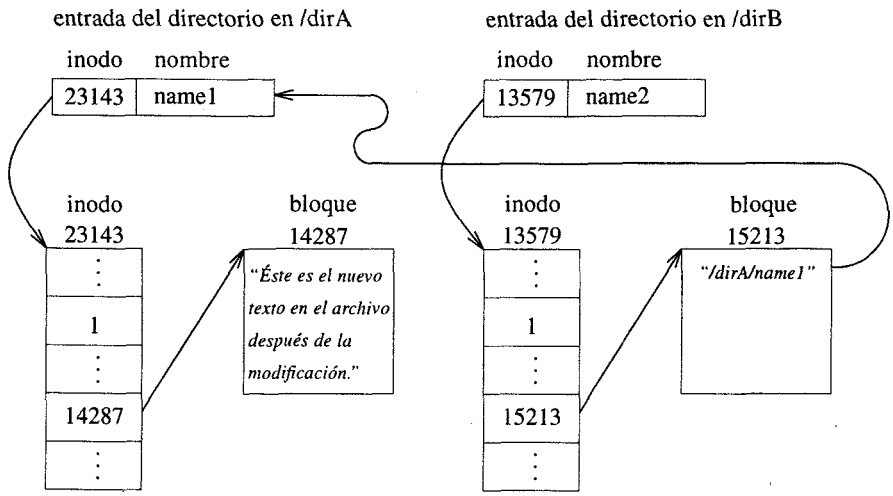


Figura 3.9: Situación después de editar un archivo que tiene una liga simbólica.

Ejercicio 3.6

Muchos programas suponen que los archivos de encabezado para el X Window System se encuentran en `/usr/include/X11`, pero bajo Sun Solaris 2 estos archivos están en el directorio `/usr/openwin/share/include/X11`. ¿Cómo puede el administrador del sistema tratar con esta inconsistencia?

Respuesta:

Existen varias maneras de abordar este problema:

- Copiar todos los archivos en `/usr/include/X11`.
- Mover todos los archivos a `/usr/include/X11`.
- Pedir a los usuarios que modifiquen todos los programas que contienen líneas de la forma

```
#include <X11/xyz.h>
```

para que las reemplacen con

```
#include "/usr/openwin/share/include/X11/xyz.h"
```

- Pedir a los usuarios que modifiquen sus archivos *make* de modo que el compilador busque los archivos de encabezado en

  `/usr/openwin/share/include`

  (Veáse la sección A.3 si no está familiarizado con los archivos *make*.)

- Crear una liga simbólica de `/usr/include/X11` al directorio

  `/usr/openwin/share/include/X11`

Todas las alternativas anteriores, con excepción de la última, tienen serios problemas. Si los archivos de encabezado son copiados en el directorio `/usr/include/X11`, entonces habrá dos copias de éstos en el sistema. Además del espacio adicional en disco requerido, cualquier actualización puede hacer que estos archivos sean inconsistentes. Mover los archivos (copiarlos al directorio `/usr/include/X11` y luego borrarlos de `/usr/openwin/share/include/X11`) puede interferir con las actualizaciones del sistema operativo. Pedir a los usuarios que modifiquen todos sus programas o incluso todos sus archivos *make* es poco razonable. Otra alternativa, que no se mencionó, es el uso de una variable de ambiente apropiada que haga que el compilador busque de manera correcta los archivos de encabezado.

Ejercicio 3.7

Dada la gran cantidad de correo que llega a un sistema, la partición raíz de un servidor está a punto de llenarse. ¿Qué puede hacer el administrador del sistema?

Respuesta:

Lo usual es mantener el correo pendiente en un directorio con un nombre tal como `/var/mail` o `/var/spool/mail`, que puede ser parte de la partición raíz. Una posibilidad es aumentar el tamaño de la partición raíz. En general, este incremento requiere volver a instalar el sistema operativo. Otra posibilidad es montar en `var` una partición no utilizada. Si no se tiene una partición de reserva disponible, entonces el directorio `/var/spool/mail` puede ser una liga simbólica a cualquier otro directorio en una partición que tenga espacio suficiente.

Ejercicio 3.8

Considere la situación mostrada en la figura 3.8. Se ejecuta el comando `rm /dirA/name1`. ¿Qué sucede con `/dirB/name2`?

Respuesta:

La liga simbólica sigue existiendo, pero apunta a algo que ya no está allí. Una referencia a `/dirB/name2` generará un error como si no existiera la liga simbólica `/dirB/name2`. Sin embargo, si tiempo después se crea un archivo nuevo con el nombre `/dirA/name1`, entonces la liga simbólica apuntará a dicho archivo.

3.3 Representación de archivos con identificadores

Dentro de los programas en C los archivos están designados por apuntadores de archivo o por descriptores de archivo. La biblioteca estándar de E/S para ANSI C (`fopen`, `fscanf`, `fprintf`, `fread`, `fwrite`, `fclose` y varias más) utiliza apuntadores de archivo, mientras que la biblioteca de E/S de UNIX (`open`, `read`, `write`, `close` e `ioctl`) emplea descriptores de archivo. Los apuntadores y descriptores de archivo proporcionan nombres lógicos o *identificadores* para llevar a cabo la entrada y salida independiente del dispositivo. Los identificadores para el apuntador de archivo de la entrada, la salida y el error estándares son `stdin`, `stdout` y `stderr`, respectivamente, y están definidos en `stdio.h`. Los identificadores para el descriptor de archivo de la entrada, salida y error estándares son `STDIN_FILENO`, `STDOUT_FILENO` y `STDERR_FILENO`, respectivamente, y están definidos en `unistd.h`.

Un `fopen` o un `open` proporcionan una asociación entre un archivo o dispositivo físico y el identificador lógico utilizado en el programa. El archivo o dispositivo físico está especificado por una cadena de caracteres (por ejemplo `/home/johns/my.dat` o `/dev/tty`). En el caso de `open`, el identificador es un índice en una tabla de descriptores de archivo, mientras que para `fopen` el identificador es un apuntador hacia una estructura de archivo.

3.3.1 Descriptores de archivo

La llamada al sistema `open` asocia un descriptor de archivo (el identificador utilizado en el programa) con un archivo o dispositivo físico.

SINOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int  oflag, ...);
```

POSIX.1, Spec 1170

Los valores de POSIX.1 para `oflag` incluyen: `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_APPEND`, `O_CREAT`, `O_EXCL`, `O_NOCTTY`, `O_NONBLOCK`, y `O_TRUNC`. Se tiene que especificar exactamente una de las banderas `O_RDONLY`, `O_WRONLY` o `O_RDWR`, las cuales designan acceso sólo para lectura, sólo para escritura o lectura y escritura, respectivamente. También puede hacer el OR (|) entre varias banderas para obtener un efecto combinado.

Ejemplo 3.9

El siguiente fragmento de código abre el archivo `/home/ann/my.dat` *para lectura.*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int myfd;
myfd = open("/home/ann/my.dat", O_RDONLY);
```

La llamada al sistema `open` pone las banderas de estado del archivo de acuerdo con el valor del segundo parámetro. `O_RDONLY` está definida en `fcntl.h`. Si el campo `oflag` es `O_CREAT`, se incluye un tercer parámetro para `open`. Éste es de tipo `mode_t` y especifica los permisos para el archivo como un valor `mode_t`.

Cada archivo se encuentra asociado con tres clases: un usuario (o propietario), un grupo y los demás. Los permisos o privilegios posibles son leer (r), escribir (w) y ejecutar (x). Estos privilegios son especificados por separado para el usuario, el grupo y los demás. La figura 3.10 muestra una disposición típica de la máscara de permisos. Un 1 en la posición que corresponde al *bit* designado en la máscara indica que para esa clase está acordado el privilegio correspondiente. POSIX.1 define los nombres simbólicos para las máscaras que corresponden a dichos *bits*, de modo que el usuario pueda especificar modos independientemente de la implantación. Estos nombres están definidos en `sys/stat.h`. La tabla 3.1 contiene los nombres simbólicos y su significado.

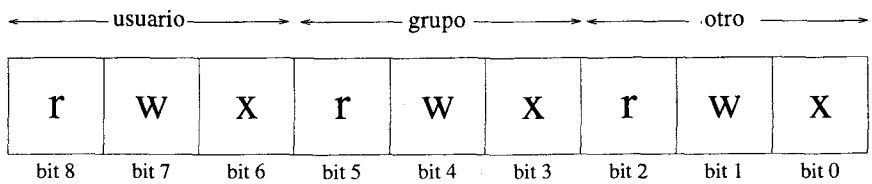


Figura 3.10: Estructura típica de la máscara de servicios.

Ejemplo 3.10

El siguiente fragmento de código crea un archivo que el usuario puede leer y escribir, pero que por los demás sólo puede ser leído.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
int fd;
mode_t fd_mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

if ((fd = open("/home/ann/my.dat", O_RDWR | O_CREAT, fd_mode))
        == -1)
    perror("No es posible abrir /home/ann/my.dat");
```

La bandera `O_CREAT` indica que si el archivo no existe entonces deberá crearse. Si el valor del segundo parámetro para la llamada a `open` en el ejemplo 3.10 fuese `O_RDWR | O_CREAT | O_TRUNC` y el archivo ya existe, entonces éste será acortado a una longitud cero (esto es, se perderá su contenido).

El descriptor de archivo `myfd` del ejemplo 3.10 es sólo un entero que especifica un índice hacia la tabla de descriptores de archivo, como se muestra en la figura 3.11. La tabla de

Modos de archivo POSIX.1

Símbolo	Significado
S_IRUSR	*bit* de permiso de lectura para el propietario
S_IWUSR	*bit* de permiso de escritura para el propietario
S_IXUSR	*bit* de permiso de ejecución para el propietario
S_IRWXU	lectura, escritura y ejecución para el propietario
S_IRGRP	*bit* de permiso de lectura para el grupo
S_IWGRP	*bit* de permiso de escritura para el grupo
S_IXGRP	*bit* de permiso de ejecución para el grupo
S_IRWXG	lectura, escritura y ejecución para el grupo
S_IROTH	*bit* de permiso de lectura para otros
S_IWOTH	*bit* de permiso de escritura para otros
S_IXOTH	*bit* de permiso de ejecución para otros
S_IRWXO	lectura, escritura y ejecución para otros
S_ISUID	fijar el ID del usuario al momento de la ejecución
S_ISGID	fijar el ID de grupo al momento de la ejecución

Tabla 3.1: Nombres simbólicos POSIX.1 para los *bits* de permiso de un archivo.

descriptores de archivo es específica de un proceso y contiene una entrada para cada archivo abierto por éste. La tabla de descriptores de archivo se encuentra en el área del proceso del usuario, pero el programa no puede tener acceso a ella más que a través de llamadas al sistema utilizando el descriptor de archivo.

La figura 3.11 muestra una entrada en la tabla de descriptores de archivo que apunta a una entrada en la tabla de archivos del sistema. La *tabla de archivos del sistema* contiene una entrada por cada `open` activo y está compartida por todos los procesos del sistema. La tabla contiene información que indica si el archivo fue abierto para lectura o escritura, si tiene protección (de lectura, escritura y ejecución para el usuario, el grupo y los demás), e información de bloqueo. La entrada de la tabla de archivos del sistema también contiene el desplazamiento del archivo, el cual indica dónde escribir o leer el siguiente dato.

También es posible que varias entradas de la tabla de archivos del sistema correspondan al mismo archivo físico. Cada una de estas entradas apunta a la misma entrada en la *tabla de inodos en memoria*. La tabla de inodos contiene una entrada para cada archivo activo en el sistema. Cuando se abre un archivo físico particular y ningún otro proceso lo ha abierto, entonces se crea una entrada en esta tabla para tal archivo. La figura 3.11 muestra que el archivo `/home/ann/my.dat` ya había sido abierto debido a que existen dos entradas en la tabla de archivos del sistema con apuntadores a la misma entrada en la tabla de inodos. (En la figura el apuntador anterior está indicado por B.)

Por eficiencia, el sistema operativo mantiene en la memoria copias de los inodos de los archivos que están activos. De lo contrario, de acuerdo con la escala de tiempo de un ciclo de procesador por segundo de la tabla 1.1, el acceso al inodo tardaría once días y al menos otros once para tener acceso al bloque de datos.

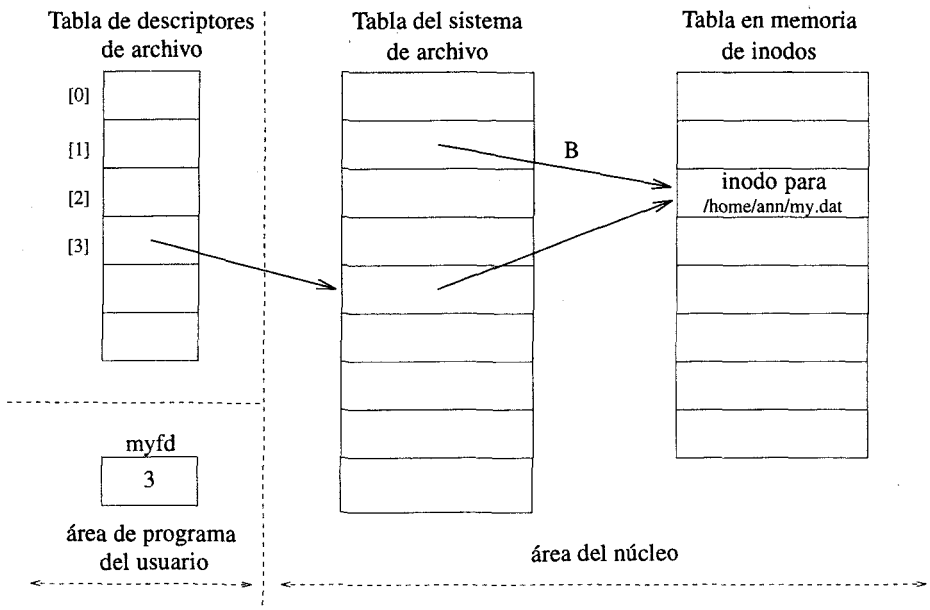


Figura 3.11: Relación entre la tabla de descriptores de archivo, la tabla del sistema de archivo y la tabla en memoria de inodos.

Ejercicio 3.9

¿Qué sucede cuando el proceso del ejemplo 3.10 ejecuta la llamada al sistema `close(myfd)`?

Respuesta:

El sistema operativo borra la cuarta entrada de la tabla de descriptores de archivo así como la entrada correspondiente a ésta en la tabla de archivos del sistema. (Véase la sección 3.3.3 para un estudio más completo.) Si el sistema operativo también borra la entrada de la tabla de inodos, entonces esto dejará al apuntador B en la tabla de archivos del sistema apuntando a cualquier cosa. En consecuencia, la entrada de la tabla de inodos debe tener una cuenta del número de entradas en la tabla de archivos del sistema que apuntan hacia ella. Cuando un proceso ejecuta la llamada al sistema `close`, el sistema operativo reduce la cuenta en la entrada del inodo, y si ésta es 0, borra la entrada de la memoria. (El sistema operativo tal vez no la borre de inmediato, en espera de que se tenga acceso a ella en un futuro próximo.)

Ejercicio 3.10

La entrada de la tabla de archivos del sistema contiene un desplazamiento que indica la posición en uso dentro del archivo. Si dos procesos abren el mismo archivo para leerlo, cada uno tiene su propio desplazamiento en el archivo. Cada proceso lee en forma independiente del otro, así que cada uno puede leer todo el archivo. ¿Qué sucede con la escritura? ¿Qué sucedería si el desplazamiento del archivo se guardara en la tabla de inodos en lugar de hacerlo en la tabla de archivos del sistema?

Respuesta:

Las operaciones de escritura son independientes la una de la otra. Cada usuario puede escribir sobre lo que el otro usuario ha escrito. Por otra parte, si los desplazamientos se guardaran en la tabla de inodos, lo escrito por aperturas distintas aparecería en forma consecutiva. En este caso, los procesos sólo leerían partes del archivo debido a que el desplazamiento de éste que ellos están empleando puede ser actualizado por otros procesos.

Ejercicio 3.11

Suponga que un proceso abre un archivo para lectura y luego ejecuta un *fork*. Tanto el padre como el hijo pueden leer el archivo. ¿Cómo están relacionadas las lecturas hechas por los dos procesos? ¿Qué sucede con la escritura?

Respuesta:

Los procesos comparten una entrada en la tabla de archivos del sistema y por tanto comparten el desplazamiento del archivo. Los dos procesos leen partes diferentes del archivo. Si ningún otro proceso ha abierto el archivo, entonces lo que se escriba se añadirá al final del archivo y ningún dato se perderá como resultado de esta operación. La sección 3.3.3 explica esta situación con más detalle.

3.3.2 Apuntadores de archivo y almacenamiento intermedio

La biblioteca de E/S estándar de ANSI C utiliza para operaciones de E/S, apuntadores de archivo más que descriptores de archivo. Un *apuntador de archivo* es un apuntador a una estructura de datos denominada estructura de archivo que está en el área de usuario del proceso. Esta estructura de datos contiene un *buffer* y un descriptor de archivo. (En el System V lo anterior se asocia con una interfaz de flujo de información (*stream interface*), pero el lector no debe preocuparse por el momento de esto.)

Ejemplo 3.11

El siguiente fragmento de código abre el archivo `/home/ann/my.dat` *para salida y luego escribe una cadena de caracteres en dicho archivo.*

```
#include <stdio.h>
FILE *myfp;

if ((myfp = fopen("/home/ann/my.dat", "w")) == NULL)
   fprintf(stderr,"No es posible abrir con fopen el archivo\n");
else
   fprintf(myfp, "Ésta es una prueba");
```

La figura 3.12 muestra la estructura de archivo asignada por la función `fopen` del ejemplo 3.11. La estructura contiene un *buffer* y un valor para el descriptor de archivo. El valor de este descriptor es el índice de la entrada en la tabla de descriptores de archivo que más adelante `fprintf` utilizará para escribir el archivo en el disco. En cierto sentido, el apuntador de archivo es un identificador para otro identificador.

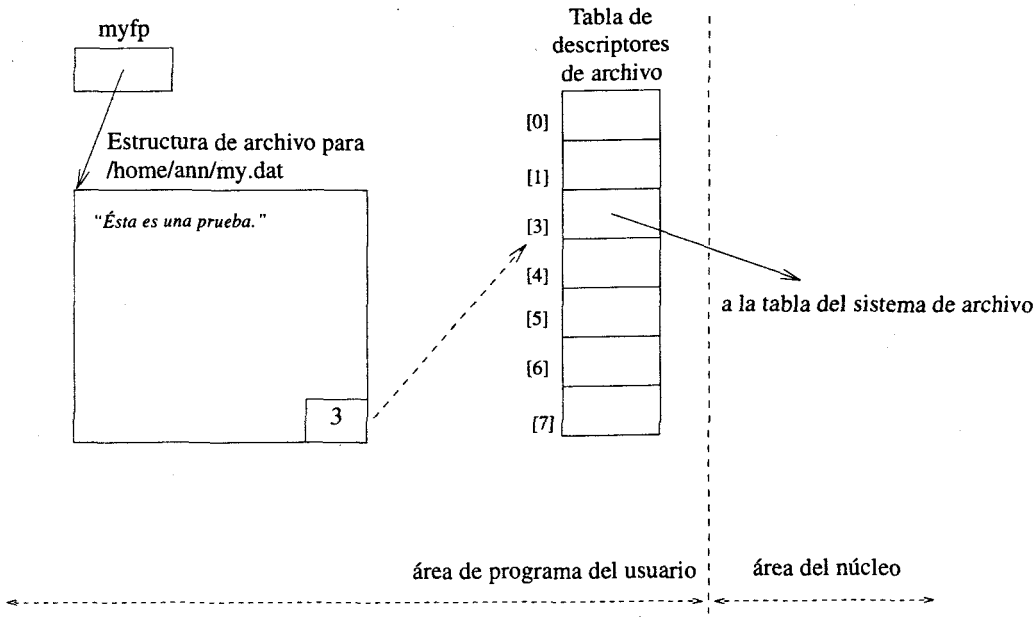


Figura 3.12: Uso esquemático del apuntador de archivo después de `fopen`.

¿Qué sucede cuando el programa llama a `fprintf`? El resultado depende del tipo de archivo que fue abierto. Lo usual es que los archivos en disco sean colocados completamente en un *buffer*, lo que significa que `fprintf` en realidad no escribe la cadena `Esta es una prueba` en el disco, sino en el *buffer* de la estructura de archivo. Cuando el *buffer* está lleno, `fprintf` llama a `write` con el descriptor de archivo, del mismo modo que en la sección anterior. El lapso entre el momento en que el programa ejecuta `fprintf` y el instante en que ocurre la escritura puede tener consecuencias interesantes, en especial si el programa termina abruptamente. En estas situaciones llegan a perderse los datos que están en el *buffer*, y también es posible que un programa termine normalmente, pero que su salida al disco sea incompleta.

¿Cómo puede evitar un programa los efectos del uso de un *buffer*? Una llamada a `fflush` obligará a escribir en el disco todo lo que se haya colocado en el *buffer* de la estructura de archivo. El programa también puede llamar a `setvbuf` para deshabilitar el empleo del *buffer*.

La E/S en terminales funciona de manera un poco diferente. La acción de almacenamiento temporal de los archivos asociados con las terminales se hace por línea más que en la totalidad del *buffer* (con excepción del error estándar, el cual no utiliza el *buffer*). En la salida, este almacenamiento por líneas significa que la línea será escrita hasta que el *buffer* esté lleno o se encuentre un símbolo de nueva línea.

Ejemplo 3.12

Los caracteres `'a'` y `'b'` *aparecen en la pantalla después de dos mensajes de error estándar debido a que la salida estándar utiliza almacenamiento intermedio, no así el error estándar* buffered.

```
#include <stdio.h>

fprintf(stdout, "a");
fprintf(stderr, "a has been written\n");
fprintf(stdout, "b");
fprintf(stderr, "b has been written\n");
fprintf(stdout, "\n");
```

Ejemplo 3.13

En el siguiente fragmento de código `scanf` *vacía el buffer de* `stdout` *de modo que la* `'a'` *aparezca antes de la lectura del número.*

```
#include <stdio.h>
int i;

fprintf(stdout, "a");
scanf("%d", &i);
fprintf(stderr, "a has been written\n");
fprintf(stdout, "b");
fprintf(stderr, "b has been written\n");
fprintf(stdout, "\n");
```

Los aspectos del almacenamiento intermedio son más sutiles de lo que el estudio anterior puede hacer creer. Si un programa que utiliza apuntadores de archivo para un dispositivo que emplea *buffers* termina anormalmente, tal vez nunca se haya escrito en el disco el último *buffer* parcial creado por `fprintf`. Cuando el *buffer* está lleno, se ejecuta un `write`. La finalización de un `write` no significa que los datos se encuentren en realidad sobre el disco. De hecho, el sistema operativo copia los datos en un *caché de buffer* del sistema. Si todo el sistema deja de trabajar anormalmente antes de que el sistema operativo escriba el bloque en el disco, el programa perderá los datos. Lo más común es que una terminación anormal del sistema sea menos probable que la de un programa.

3.3.3 Herencia de descriptores de archivo

Cuando `fork` crea un hijo, éste hereda una copia de la mayor parte del ambiente y contexto del padre, que incluye el estado de las señales, los parámetros de la planeación de procesos y la tabla de descriptores de archivo. Las implicaciones de la herencia no siempre resultan obvias. Dado que los hijos heredan la tabla de descriptores de su padre, el padre y los hijos comparten el mismo desplazamiento de archivo para los archivos que fueron abiertos por el padre antes del `fork`.

Ejemplo 3.14

En el fragmento de código siguiente el hijo hereda el descriptor de archivo de `my.dat`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#include <unistd.h>
#include <stdio.h>
int myfd;
pid_t childpid;

if ((myfd = open("my.dat", O_RDONLY)) == -1)
   perror("Could not open file");
else if ((childpid = fork()) == -1)
   perror("Could not fork");
else if (childpid == 0)
   /* El código del hijo va aquí */
else
   /* El código del padre va aquí */
```

La figura 3.13 muestra las tablas de descriptores de archivo del padre y el hijo para el ejemplo 3.14. Las entradas de la tabla de descriptores de los dos procesos apuntan a la misma entrada en la tabla de archivos del sistema. Por consiguiente, el padre y el hijo comparten el desplazamiento de archivo que está guardado en la tabla de archivos del sistema.

Cuando un programa cierra un archivo, se libera la entrada en la tabla de descriptores. ¿Qué sucede con la entrada correspondiente en la tabla del sistema de archivo? Ésta sólo puede liberarse si ya no existen más entradas en la tabla de descriptores de archivo que apunten hacia ella. Por esta razón, cada entrada de la tabla del sistema de archivo contiene la cuenta del número de entradas en la tabla de descriptores de archivo que apuntan hacia ella. Cuando se cierra un archivo, el sistema operativo reduce la cuenta y borra la entrada solamente cuando la cuenta es 0. De manera similar, después de la eliminación de una liga dura el sistema operativo disminuye la cuenta de ligas en un inodo, como ya se estudió en la sección 3.2.2.

Ejemplo 3.15

En el siguiente fragmento de código, tanto el padre como el hijo abren el archivo `my.dat` *para lectura.*

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
int myfd;
pid_t childpid;

if ((childpid = fork()) == -1)
   perror("Could not fork");
else if ((myfd = open("my.dat", O_RDONLY)) == -1)
   perror("Could not open file");
else if (childpid == 0)
      /* el código del hijo va aquí */
else
      /* el código del padre va aquí */
```

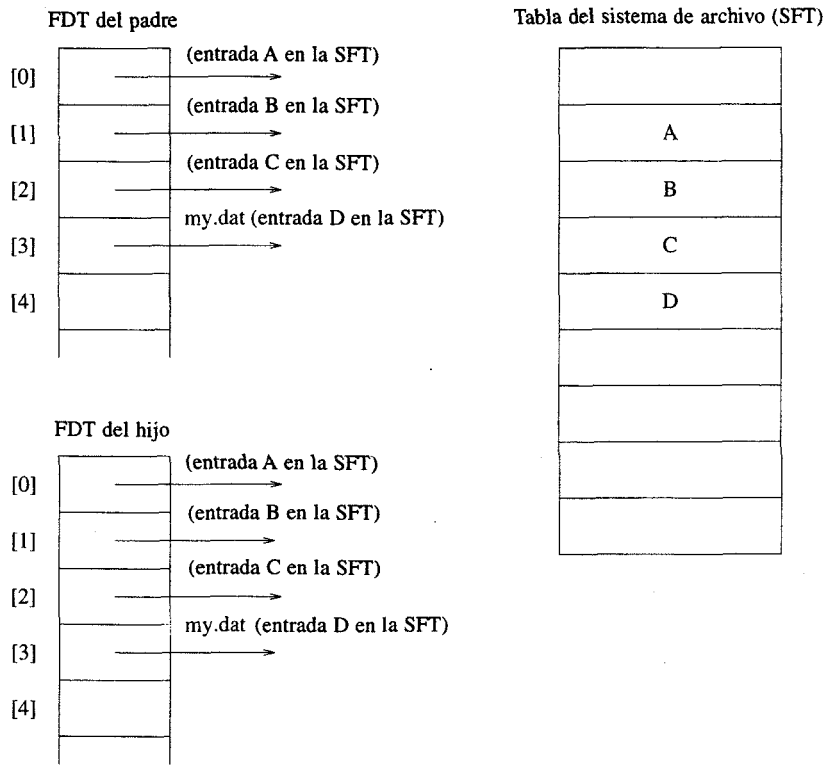


Figura 3.13: Si el padre abre `my.dat` antes del `fork`, entonces el padre y el hijo comparten la entrada de la tabla del sistema de archivo.

La figura 3.14 muestra las tablas de descriptores de archivo para el ejemplo 3.15. Las entradas en la tabla de descriptores de archivo apuntan a entradas distintas en la tabla del sistema de archivo. En consecuencia, el padre y el hijo no comparten el mismo desplazamiento de archivo. El hijo no hereda el descriptor de archivo debido a que cada proceso abrió el archivo después del `fork`, creando con esto entradas separadas en la tabla del sistema de archivo. Aun con esto, el padre y el hijo comparten las entradas de la tabla del sistema de archivo para la entrada, el error y la salida estándares.

3.4 Filtros y redireccionamiento

UNIX proporciona una cantidad muy grande de utilerías que están escritas como filtros. Un *filtro* lee la entrada estándar, lleva a cabo una transformación y pone el resultado en la salida estándar. Los filtros escriben sus mensajes de error en la salida de error estándar. Todos los parámetros de los filtros se comunican como argumentos de línea comando. Los datos de

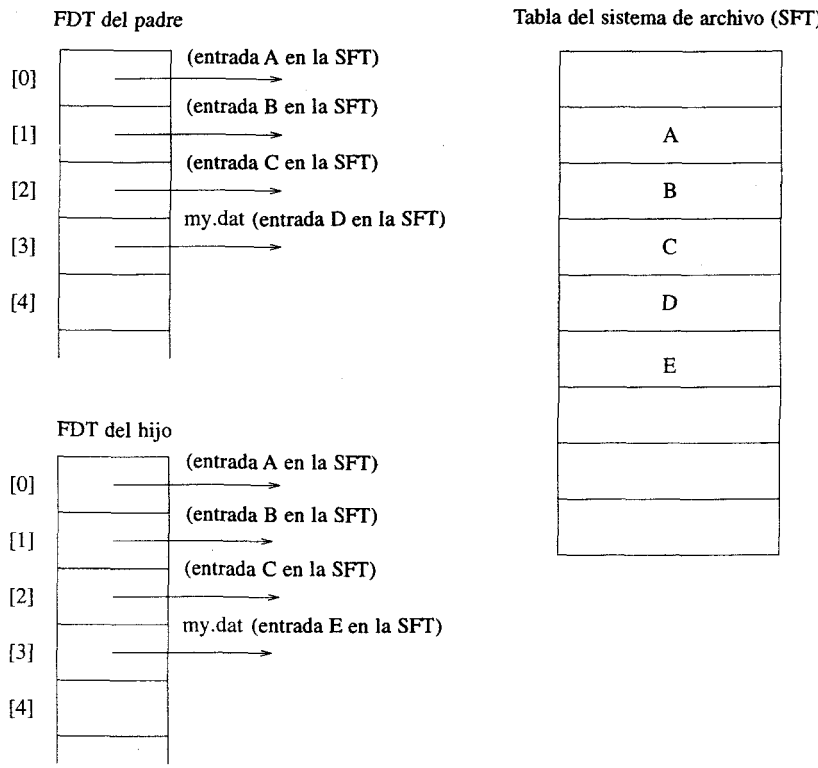


Figura 3.14: Si el padre y el hijo abren `my.dat` después del `fork`, las entradas de sus tablas de descriptores de archivos apuntan a entradas diferentes en la tabla del sistema de archivo.

entrada al filtro no deben tener encabezados o finales, y éste no debe requerir de ninguna interacción con el usuario.

Ejemplos de filtros útiles de UNIX incluyen `head`, `tail`, `more`, `sort`, `grep` y `awk`. El comando `cat` toma una lista de nombres de archivo como argumentos de línea comando, lee cada archivo en sucesión e imprime el contenido de cada uno en la salida estándar. Sin embargo, si no se especifica ningún archivo, `cat` toma su entrada de la entrada estándar y envía su salida a la salida estándar. En este caso, `cat` se comporta como un filtro.

Recuérdese que un descriptor de archivo es un índice hacia la tabla de descriptores de archivo del proceso. Cada entrada en esta tabla apunta a una entrada en la tabla del sistema de archivo, entrada que se crea cuando se abre el archivo. Un programa puede modificar la entrada de la tabla de descriptores de archivo de modo que apunte a una entrada diferente en la tabla del sistema de archivo. Esta acción se conoce como *redireccionamiento*. Muchos *shell* interpretan el carácter mayor que (>) en la línea comando como un redireccionamiento de la salida estándar, y un carácter menor que (<) como un redireccionamiento de la entrada estándar. (Asocie > con la salida imaginando el símbolo como una flecha que apunta en la dirección del archivo de salida.)

Ejemplo 3.16

El comando `cat` *sin argumentos en la línea comando lee la entrada estándar y la reproduce en la salida estándar. El comando siguiente redirige la salida estándar hacia* `my.file` *con el símbolo* >.

```
cat > my.file
```

El comando `cat` del ejemplo 3.16 toma todo lo que se escribe con el teclado y lo pone en el archivo `my.file`. La figura 3.15 ilustra la tabla de descriptores de archivo del ejemplo 3.16. Antes de la redirección, la entrada `[1]` de la tabla de descriptores apunta hacia la entrada correspondiente en la tabla del sistema de archivo, la cual corresponde al dispositivo usual de salida estándar. Después del redireccionamiento, la entrada `[1]` apunta a la entrada de la tabla del sistema de archivo que corresponde a `my.file`.

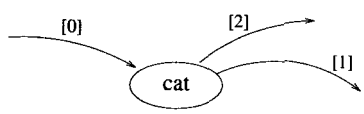


Tabla de descriptores de archivo antes del redireccionamiento

[0]	*entrada estándar*
[1]	*salida estándar*
[2]	*error estándar*

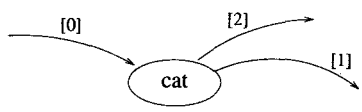


Tabla de descriptores de archivo después del redireccionamiento

[0]	*entrada estándar*
[1]	*salida estándar*
[2]	*error estándar*

Figura 3.15: Estado de la tabla de descriptores de archivo antes y después del redireccionamiento para el proceso que ejecuta `cat > my.file`.

El redireccionamiento de la salida estándar en `cat > my.file` ocurre debido a que el *shell* cambia la entrada de la salida estándar de la tabla de descriptores de archivo (un apuntador a la tabla del sistema de archivo) para que apunte a la entrada de la tabla del sistema de archivo asociada con `my.file`. Para lograr este redireccionamiento en un programa en C, primero es necesario abrir el archivo `my.file` para establecer una entrada apropiada en la tabla del sistema de archivo. Después del `open`, se copia el apuntador a `my.file` en la entrada de la salida estándar mediante la ejecución de la llamada al sistema `dup2`. La llamada `dup2` requiere dos parámetros, `fildes` y `fildes2`. La llamada cierra la entrada `fildes2` de la tabla de descriptores de archivo y luego copia el apuntador de la entrada `fildes` en la entrada `fildes2`.

SINOPSIS

```
#include <unistd.h>

int dup2(int fildes, int fildes2);
```

POSIX.1, Spec 1170

Ejemplo 3.17

El siguiente código redirige la entrada estándar hacia el archivo `my.file`.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int fd;
mode_t fd_mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

if ((fd = open("my.file", O_WRONLY | O_CREAT, fd_mode)) == -1)
   perror("Could not open my.file");
else {
   if (dup2(fd, STDOUT_FILENO) == -1)
       perror("Could not redirect standard output");
   close(fd);
}
```

La figura 3.16 muestra el efecto de la redirección sobre la tabla de descriptores de archivo del ejemplo 3.17. El `open` hace que el sistema operativo cree una entrada nueva en la tabla del sistema de archivo y que la entrada `[3]` de la tabla de descriptores de archivo apunte hacia ella.

La llamada `dup2` cierra el descriptor que corresponde al segundo parámetro (la salida estándar, `STDOUT_FILENO`) y luego copia la entrada correspondiente al primer parámetro (`fd`) en la entrada que corresponde al segundo parámetro (`STDOUT_FILENO`). A partir de ese punto del programa, todo lo que se escriba en la salida estándar irá a `my.file`.

3.5 Entubamiento (pipes)

El programador puede escribir transformaciones complicadas a partir de filtros simples enviando la salida estándar de un filtro a la entrada estándar del siguiente.

Ejemplo 3.18

Los comandos siguientes hacen uso del filtro `sort` *en conjunción con* `ls` *para producir un listado del directorio ordenado por tamaño.*

```
ls -l > my.file
sort -n +4 < my.file
```

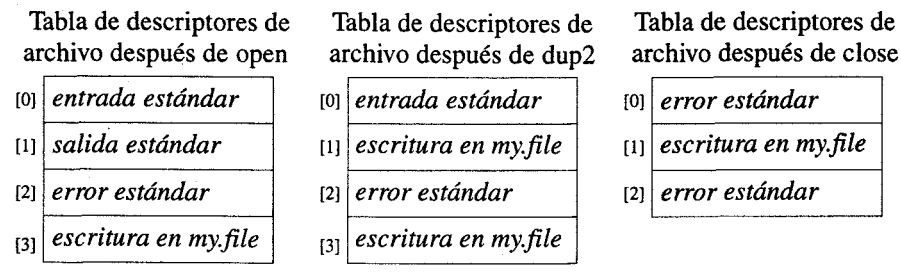


Figura 3.16: Estado de la tabla de descriptores de archivo durante la ejecución del ejemplo 3.17.

La primera opción del filtro `sort` del ejemplo 3.18 indica el tipo de ordenamiento (`n` significa numérico). La segunda opción señala que la llave para hacer el ordenamiento debe encontrarse al saltar cuatro campos. (El número de campos por saltar depende de la versión de UNIX.) El mandato `ls` escribe su salida en un archivo intermedio (`my.file`) que `sort` emplea como entrada.

Ejemplo 3.19

La siguiente alternativa para la realización del ejemplo 3.18 produce un listado de directorio ordenado sin crear el archivo intermedio `my.file`.

```
ls -l | sort -n +4
```

La conexión entre `ls` y `sort` del ejemplo 3.19 es diferente del redireccionamiento, ya que `ls` y `sort` no comparten una tabla de descriptores de archivo común. La salida estándar de `ls` está "conectada" a la entrada estándar de `sort` a través de un *buffer* de comunicación denominado *entubamiento (*pipes*)*. La figura 3.17 muestra un diagrama de la conexión y las tablas de descriptores de archivo correspondientes después del establecimiento de la conexión. El `ls` redirige su salida estándar para escribir en el conector, y `sort` redirige su entrada estándar para hacer la lectura en el entubamiento. El mandato `sort` lee los datos que `ls` escribe de acuerdo con el esquema primero en entrar-primero en salir. El mandato `sort` no tiene que leer los datos con la misma rapidez con que `ls` los escribe en el entubamiento.

SINOPSIS

```
#include <unistd.h>

int pipe(int fildes[2]);
```

POSIX.1, Spec 1170

La llamada `pipe` crea un *buffer* de comunicación que puede tener acceso al proceso que la llama, a través de los descriptores de archivo `fildes[0]` y `fildes[1]`. Los datos que se escriben en `fildes[1]` son leídos de `fildes[0]` de acuerdo con el esquema primero en

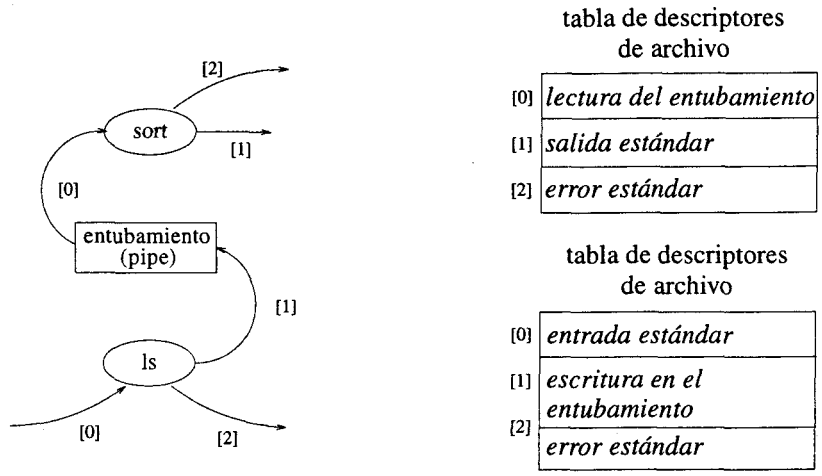


Figura 3.17: Estado de la tabla de descriptores de archivo durante la ejecución del ejemplo 3.19.

entrar-primero en salir. Un entubamiento no tiene ningún nombre externo o permanente, de modo que un programa sólo puede tener acceso a él mediante los dos descriptores de archivo. Por esta razón el entubamiento puede ser utilizado únicamente por el proceso que lo crea, así como por los descendientes de éste que heredan los descriptores del `fork`.

Nota: La llamada `pipe` descrita aquí crea un *buffer* de comunicación unidireccional tradicional. El System V Release 4 implanta los conectores utilizando un mecanismo de comunicación bidireccional denominado STREAMS. Los *conectores STREAMS* permiten que los datos escritos en `fildes[0]` sean leídos de `fildes[1]` y viceversa. El estándar POSIX.1 no prohíbe esta extensión. En el capítulo 12 se estudian los STREAMS.

Ejemplo 3.20

El siguiente programa implanta el redireccionamiento del ejemplo 3.19. Para simplificar, se ha omitido la mayor parte de las instrucciones de verificación de errores de este código .

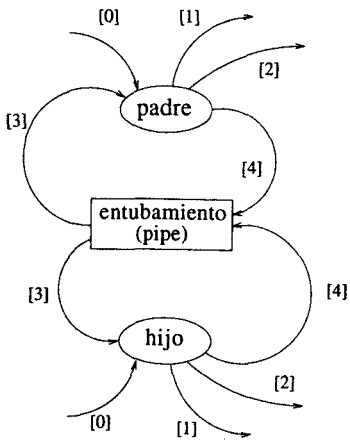
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

void main(void)
{
   int fd[2];
   pid_t childpid;

   pipe(fd);
   if ((childpid = fork()) == 0) {   /* ls es el hijo */
      dup2(fd[1], STDOUT_FILENO);
```

```
        close(fd[0]);
        close(fd[1]);
        execl("/usr/bin/ls", "ls", "-l", NULL);
        perror("The exec of ls failed");
    } else {                          /* sort es el padre */
        dup2(fd[0], STDIN_FILENO);
        close(fd[0]);
        close(fd[1]);
        execl("/usr/bin/sort", "sort", "-n", "+4", NULL);
        perror("Falló el exec de sort");
    }
    exit(0);
}
```

Las figuras 3.18 a la 3.20 muestran el estado de la tabla de descriptores de archivo para el ejemplo 3.20. En la figura 3.18, el proceso hijo hereda una copia de la tabla de descriptores de archivo del padre. La figura 3.19 muestra la tabla de descriptores de archivo después de que el hijo y el padre redirigen su salida y entrada estándar respectivamente, pero antes que cualquiera de los procesos cierre los descriptores de archivo innecesarios. La figura 3.20 presenta la configuración final.



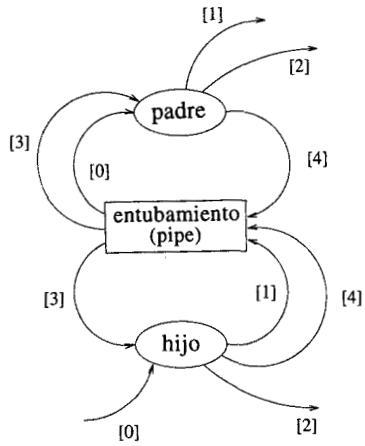
Padre
tabla de descriptores de archivo

[0]	*entrada estándar*
[1]	*salida estándar*
[2]	*error estándar*
[3]	*lectura del entubamiento (pipe)*
[4]	*escritura del entubamiento (pipe)*

Hijo
tabla de descriptores de archivo

[0]	*entrada estándar*
[1]	*salida estándar*
[2]	*error estándar*
[3]	*lectura del entubamiento (pipe)*
[4]	*escritura del entubamiento (pipe)*

Figura 3.18: Estado de la tabla de descriptores de archivo después de la ejecución del `fork` en el ejemplo 3.20.



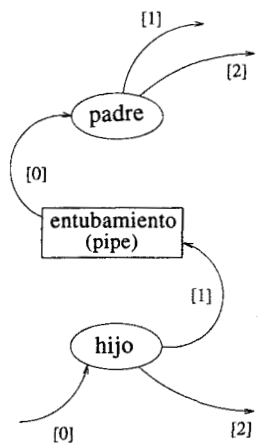
Padre
tabla de descriptores de archivo

[0]	*lectura del entubamiento (pipe)*
[1]	*salida estándar*
[2]	*error estándar*
[3]	*lectura del entubamiento (pipe)*
[4]	*escritura del entubamiento (pipe)*

Hijo
tabla de descriptores de archivo

[0]	*entrada estándar*
[1]	*escritura del entubamiento (pipe)*
[2]	*error estándar*
[3]	*lectura del entubamiento (pipe)*
[4]	*escritura del entubamiento (pipe)*

Figura 3.19: Estado de la tabla de descriptores de archivo después de la ejecución de `dup2` en los dos procesos del ejemplo 3.20.



Padre
tabla de descriptores de archivo

[0]	*lectura del entubamiento (pipe)*
[1]	*salida estándar*
[2]	*error estándar*

Hijo
tabla de descriptores de archivo

[0]	*entrada estándar*
[1]	*escritura del entubamiento (pipe)*
[2]	*error estándar*

Figura 3.20: Estado de la tabla de descriptores justo antes de la ejecución de los `execl` en los procesos del ejemplo 3.20.

3.6 Lectura y escritura de archivos

UNIX proporciona acceso secuencial a los archivos a través de las llamadas al sistema `read` y `write`. Estas llamadas leen o escriben un bloque de datos a partir del desplazamiento en uso (*current offset*) dentro del archivo. Las llamadas actualizan el desplazamiento de modo que la siguiente operación comience donde la última se quedó.

SINOPSIS

```
#include <unistd.h>

ssize_t read(int fildes, void *buf, size_t nbyte);
```

POSIX.1, Spec 1170

La llamada `read` pide que se lea `nbytes` bytes del archivo cuyo descriptor es `fildes`, y que éstos sean colocados en `buf`. El que hace la llamada debe proporcionar un *buffer* de tamaño suficiente para guardar en él `nbyte` bytes de datos. (Un error común es proporcionar un apuntador no iniciado a un `char` más que a un *buffer*.) Si `read` tiene éxito, devuelve entonces el número de bytes que leyó. Las condiciones de error para `read` son mencionados en la página 115.

SINOPSIS

```
#include <unistd.h>

ssize_t write(int fildes, const void *buf, size_t nbyte);
```

POSIX.1, Spec 1170

`write` intenta escribir `nbyte` bytes tomados del `buf` en el archivo cuyo descriptor es `fildes`. Si la llamada tiene éxito, `write` devuelve el número de bytes que fueron escritos.

Ejemplo 3.21

El siguiente fragmento de código lee de `from_fd` *y escribe en* `to_fd`.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#define BLKSIZE 1024

int from_fd, to_fd;
char buf[BLKSIZE];
int bytesread;

while ((bytesread = read(from_fd, buf, BLKSIZE)) > 0)
    if (write(to_fd, buf, bytesread) <= 0)
        break;
```

Nótese que el `write` del ejemplo 3.21 intenta escribir `bytesread` bytes en lugar de `BLKSIZE` bytes, tomando en cuenta de esta manera el hecho de que `read` en realidad puede no

leer todo el número de bytes pedido. Sin embargo, no existe ninguna garantía de que `write` escriba todos los bytes solicitados. Por otra parte, `read` o `write` pueden ser interrumpidos por una señal. En este caso, la llamada regresa un `-1` y hace que `errno` sea `EINTR`.

El programa 3.2 copia un archivo. Los nombres de los archivos fuente y destino se pasan al programa como argumentos de línea comando. Dado que la apertura del archivo de destino tiene como parámetros `O_CREAT | O_EXCL`, la copia del archivo falla si éste ya existe. El ciclo para hacer la copia maneja escrituras parciales y continúa copiando incluso si existe una interrupción generada por una señal. El capítulo 5 estudia las señales y la interrupción de las llamadas del sistema.

Programa 3.2: Programa para copiar un archivo.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define BLKSIZE 1024

void main(int argc, char *argv[])
{
   int from_fd, to_fd;
   int bytesread, byteswritten;
   char buf[BLKSIZE];
   char *bp;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s from_file to_file\n", argv[0]);
      exit(1);
   }

   if ((from_fd = open(argv[1], O_RDONLY)) == -1) {
      fprintf(stderr, "Could not open %s: %s\n",
              argv[1], strerror(errno));
      exit(1);
   }

   if ((to_fd = open(argv[2], O_WRONLY | O_CREAT | O_EXCL,
              S_IRUSR | S_IWUSR)) == -1) {
      fprintf(stderr, "Could not create %s: %s\n",
              argv[2], strerror(errno));
      exit(1);
```

```
    }

    while (bytesread = read(from_fd, buf, BLKSIZE)) {
        if ((bytesread == -1) && (errno != EINTR))
            break;              /* real error occurred on the descriptor */

        else if (bytesread > 0) {
            bp = buf;
            while(byteswritten = write(to_fd, bp, bytesread)) {
                if ((byteswritten == -1) && (errno != EINTR))
                    break;
                else if (byteswritten == bytesread)
                    break;
                else if (byteswritten > 0) {
                    bp += byteswritten;
                    bytesread -= byteswritten;
                }
            }
            if (byteswritten == -1)
                break;
        }
    }
    close(from_fd);
    close(to_fd);
    exit(0);
}
```

Programa 3.2

El `read` de un archivo ordinario regresa menos bytes que los solicitados si la operación de lectura llega al final del archivo antes de leer todos los bytes o si es interrumpido por una señal. Cuando un `read` solicita cierto número de bytes de un entubamiento, `read` regresa cuando el entubamiento no está vacío. El número de bytes leído es menor o igual al número de bytes solicitado. `read` devuelve el número de bytes leídos en realidad. Si `read` es interrumpido por una señal y no se ha transferido ningún dato, entonces el valor devuelto por `read` es `-1` y `errno` es igual a `EINTR`.

La llamada al sistema `read` regresa 0 si un programa intenta comenzar a leer después del final de un archivo ordinario. Una vez que se presenta la condición de fin de archivo, el programa ya no puede leer más allá de dicho punto aunque el archivo se extienda posteriormente. La detección del fin de un archivo para archivos especiales, como los entubamientos, es más complicada. El entubamiento puede estar vacío debido a una operación anterior de escritura fallida. La condición de fin de archivo para entubamientos ocurre sólo cuando el conector está vacío y no hay más procesos con descriptores para escritura abiertos para el conector. Por tanto, es necesario *cerrar todos los descriptores no utilizados o el programa no será capaz de detectar el final del archivo*. Existen ciertos problemas que pueden presentarse con los descriptores de archivo abiertos errantes.

3.7 E/S sin bloqueo

Los *buffers* de comunicación como los entubamientos pueden vaciarse si se lee toda la información previamente escrita en ellos. Un *buffer* vacío no es un indicador de fin de archivo. Más bien, refleja la naturaleza asíncrona de la comunicación entre procesos. Normalmente, cuando un proceso intenta leer dicho *buffer*, lo bloquea (espera en el `read`) hasta que la entrada esté disponible.

El bloqueo de E/S presenta un problema para un proceso que está vigilando más de un *buffer* de comunicación de entrada. El proceso no tiene ninguna manera de conocer qué entrada es la que llegará primero —si el intento por averiguarlo resulta equivocado, el proceso puede quedar congelado indefinidamente.

Una manera de manejar este problema es mediante el empleo de E/S sin bloqueo. Si se configuran las banderas de control apropiadas, el programa puede hacer que el `read` regrese de inmediato si no hay entrada disponible. Después, el proceso intentará continuamente leer los descriptores de los archivos de entrada, uno a la vez. Este método, denominado escrutinio (*polling*), utiliza la CPU de una manera ineficiente.

Un programa lleva a cabo operaciones de E/S sin bloqueo utilizando la bandera `O_NONBLOCK` asociada con el descriptor de archivo. La llamada al sistema `fcntl` modifica las banderas y el estado asociadas con un objeto que tiene un descriptor de archivo.

SINOPSIS

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fildes, int cmd, /* arg */ ...);
```

POSIX.1, Spec 1170

Ejemplo 3.22

El fragmento de código siguiente modifica el descriptor de un archivo abierto `fd` *para hacer E/S sin bloqueo.*

```
#include <fcntl.h>
#include <stdio.h>
int fd_flags;

if ((fd_flags = fcntl(fd, F_GETFL, 0)) == -1)
   perror("Could not get flags for fd");
else {
   fd_flags |= O_NONBLOCK;
   if (fcntl(fd, F_SETFL, fd_flags) == -1)
      perror("Could not set flags for fd");
}
```

El segmento de código del ejemplo 3.22 lee el valor que tienen las banderas asociadas con `fd`, lleva a cabo un OR bit a bit con `O_NONBLOCK`, e instala las banderas modificadas. Después

de la ejecución de este segmento, las llamadas al sistema `read` asociadas con los `fd` regresan de inmediato si no hay entrada disponible.

Ejemplo 3.23

El código siguiente cambia a bloqueo el modo de E/S asociado con el descriptor de archivo `fd` *al borrar la bandera* `O_NONBLOCK`.

```
#include <fcntl.h>
#include <stdio.h>
int fd_flags;

if ((fd_flags = fcntl(fd, F_GETFL, 0)) == -1)
   perror("Could not get flags for fd");
else {
   fd_flags &= ~O_NONBLOCK;
   if (fcntl(fd, F_SETFL, fd_flags) == -1)
      perror("Could not set flags for fd");
}
```

3.8 La llamada `select`

Una alternativa para el escrutinio (polling) es dejar que el proceso haga el bloqueo hasta que en cualquiera de los descriptores de archivo esté disponible y se presenten datos en la E/S. El bloqueo hasta que al menos un miembro de un conjunto de condiciones sea verdadero se conoce como *sincronización OR*. La condición para el caso descrito es "entrada disponible" (input available) en un descriptor.

La llamada `select` proporciona un método para vigilar los descriptores de archivo en lo que respecta a una de tres condiciones: que la lectura puede hacerse sin bloqueo, la escritura puede hacerse sin bloqueo, o la existencia de una condición excepcional pendiente. (Esta última no significa la ocurrencia de un error. Más bien, indica la presencia de un dato fuera de banda durante la comunicación por red.)

SINOPSIS

```
#include <sys/time.h>
#include <sys/types.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

Spec 1170

El primer parámetro de `select` es el número de *bits* por examinar en los conjuntos de descriptores de archivo. Este valor debe ser, al menos, una unidad mayor que el descriptor de archivo más grande que se va a examinar. El parámetro `readfd` indica el conjunto de descriptores que habrá de vigilarse para operaciones de lectura. De manera similar, `writefds` especifica

el conjunto de descriptores que deben vigilarse para operaciones de escritura, y `exceptfds` indica los descriptores de archivo por vigilar para condiciones excepcionales. Los conjuntos de descriptores son de tipo `fd_set`.

Al regresar, `select` borra todos los descriptores en cada uno de los `readfds`, `writefds` y `exceptfds`, con excepción de los que están listos. El valor de regreso para `select` es el número de descriptores que están listos o `-1` si ocurre un error. El último parámetro es un valor de tiempo (timeout) empleado para obligar el regreso de `select` después de haber transcurrido cierto lapso, incluso si no hay descriptores listos. Cuando `timeout` es `NULL`, `select` puede hacer el bloqueo indefinidamente. Si `select` es interrumpida por una señal, devuelve entonces `-1` y hace `errno` igual con `EINTR`.

Históricamente el conjunto de descriptores fue implantado como una máscara de *bits*, pero esto no funciona para más de 32 descriptores de archivo. Ahora lo usual es representar los conjuntos de descriptores por medio de campos de *bits* en arreglos de enteros, proporcionando los macros `FD_SET`, `FD_CLR`, `FD_ISSET` y `FD_ZERO` para manejar los conjuntos de descriptores de una manera que no dependa de la implantación.

SINOPSIS

```
#include <sys/time.h>
#include <sys/types.h>

void FD_SET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
int FD_ISSET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

Spec 1170

El macro `FD_SET` establece el *bit* en `fdset` que corresponde al descriptor de archivo `fd`, mientras que el macro `FD_CLR` lo borra. El macro `FD_ZERO` borra todos los *bits* en `fdset`. Estos macros deben ser empleados para construir las máscaras del descriptor antes de hacer la llamada a `select`. El macro `FD_ISSET` se utiliza después de `select` para determinar si el correspondiente *bit* del descriptor de archivo `fd` está definido en la máscara `fdset`.

Ejemplo 3.24

El siguiente fragmento de código vigila de manera continua dos descriptores de archivo `pipe1` *y* `pipe2` *para ver si hay entrada.*

```
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <sys/types.h>
#include <errno.h>

fd_set readset;
int maxfd;
int  pipe1;
```

```
int  pipe2;

    /* el código de configuración de pipe1 y pipe2 va aquí */
maxfd = pipe1;                  /* se determina el fd mayor para select */
if (pipe2 > maxfd)
   maxfd = pipe2;

for( ; ; ) {
   FD_ZERO(&readset);
   FD_SET(pipe1, &readset);
   FD_SET(pipe2, &readset);
   if ( (select(maxfd+1, &readset, NULL, NULL, NULL) == -1) &&
      (errno != EINTR) )
      perror("Falla en select");
   else {
      if (FD_ISSET(pipe1, &readset)) {
          /* lectura y proceso de la entrada pipe1 */
      }
      if (FD_ISSET(pipe2, &readset)) {
         /* lectura y proceso de la entrada pipe2 */
      }
   }
}
```

El `select` del ejemplo 3.24 hace el bloqueo hasta que exista algo qué leer en `pipe1` o `pipe2`, así que a diferencia del escrutinio, este método no desperdicia ciclos de la CPU. La llamada `select` no es parte de POSIX.1. Uno de los mayores problemas de POSIX es que no hay una manera directa de vigilar dos descriptores de archivo sin tener que esperar. Aunque `select` forma parte de UNIX 4.3 BSD, y no de System V, aparece como parte de Spec 1170. La sección 9.1 proporciona más ejemplos de `select` y estudia también a `poll`, la contraparte de `select` en el System V.

3.9 FIFO

Los entubamientos son temporales en el sentido que éstos desaparecen cuando ningún proceso los abre. Los FIFO, también conocidos como *entubamientos con nombre* (*named pipes*), están representados por archivos especiales y persisten aun después de que todos los procesos los hayan cerrado. Cualquier proceso que tenga los permisos apropiados, puede tener acceso a un FIFO. Los FIFO pueden crearse con el comando `mkfifo` desde el *shell*, o dentro de un programa, con una llamada al sistema `mkfifo`.

SINOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
```

POSIX.1, Spec 1170

Ejemplo 3.25

El siguiente código crea un entubamiento con nombre `myfifo`*, que puede ser leído por cualquiera, pero sobre el que sólo su propietario puede escribir.*

```
#include <sys/stat.h>
#include <sys/types.h>

mode_t fifo_perms = S_IRUSR | S_IWUSR | S_IRGRP| S_IROTH;

if (mkfifo("myfifo", fifo_perms) == -1)
   perror("No fue posible crear myfifo");
```

El programa 3.3 crea un entubamiento con nombre a partir de una ruta de acceso especificada en la línea comando. Luego crea un hijo. El proceso hijo escribe en el entubamiento con nombre y el padre lee lo que el hijo escribió.

Programa 3.3: El padre lee lo que el hijo ha escrito en un entubamiento con nombre.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#define BUFSIZE 256
void main (int argc, char *argv[])
{
   mode_t fifo_mode = S_IRUSR | S_IWUSR;
   int fd;
   int status;
   char buf[BUFSIZE];
   unsigned strsize;
   int mychild;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s pipename\n", argv[0]);
      exit(1);
   }
                          /* create a named pipe with r/w for user */
   if ((mkfifo(argv[1], fifo_mode) == -1) && (errno != EEXIST)) {
      fprintf(stderr, "Could not create a named pipe: %s\n", argv[1]);
      exit(1);
   }

   if ((mychild = fork()) == -1){
```

```
        perror("Could not fork");
        exit(1);
    } else if (mychild == 0) {                    /* El hijo escribe */
        fprintf(stderr, "Child[%ld] about to open FIFO %s\n",
                        (long)getpid(), argv[1]);
        if ((fd = open(argv[1], O_WRONLY))== -1) {
            perror("Child cannot open FIFO");
            exit(1);
        }
        sprintf(buf,
            "This was written by the child[%ld]\n", (long)getpid());
        strsize = strlen(buf) + 1;
        if (write(fd, buf, strsize) != strsize) {
            fprintf(stderr, "Child write to FIFO failed\n");
            exit(1);
        }
        fprintf(stderr, "Child[%ld] is done\n", (long)getpid());
    } else {                                       /* El padre lee */
        fprintf(stderr, "Parent[%ld] about to open FIFO %s\n",
                        (long)getpid(), argv[1]);
        if ((fd = open(argv[1], O_RDONLY | O_NONBLOCK)) == -1) {
            perror("Parent cannot open FIFO");
            exit(1);
        }
        fprintf(stderr,"Parent[%ld] about to read\n", (long)getpid());
        while ((wait(&status)== -1) && (errno == EINTR))
            ;
        if (read(fd, buf, BUFSIZE) <= 0) {
            perror("Parent read from FIFO failed\n");
            exit(1);
        }
        fprintf(stderr, "Parent[%ld] got: %s\n", (long)getpid(), buf);
    }
    exit(0);
}
```

Programa 3.3

Ejercicio 3.12

Lea la página del manual que corresponde a `open(2)` e intente determinar por qué el padre utilizó `O_NONBLOCK` pero el hijo no. ¿Por qué tiene que esperar el padre al proceso hijo? ¿Tiene importancia el punto donde el padre invoca el `wait`?

Respuesta:

Si el hijo intenta ejecutar un `open` con `O_NONBLOCK`, éste devuelve un error si el padre aún no ha abierto el conector para lectura. Si ninguno hace uso de `O_NONBLOCK`, entonces ambos hacen el bloqueo hasta que el otro tenga éxito, causando con ello un

interbloqueo (deadlock). Si el padre intenta leer antes de que el hijo escriba, el read regresa un error puesto que el padre hizo la apertura con `O_NONBLOCK`. Si el padre espera hasta que el hijo termine para abrir el entubamiento, el hijo lo bloqueará en su `open`.

La complicación mencionada en el ejercicio 3.12 puede evitarse si los procesos padre e hijo abren el FIFO para lectura y escritura. El `open` puede hacerse antes del `fork`, pero esto entonces ya no ilustra la diferencia entre los FIFO y los entubamientos. La diferencia importante es que para los FIFO no es necesario que los procesos que comunican estén relacionados; ellos sólo necesitan conocer el nombre del FIFO y compartir un sistema de archivo.

3.10 Archivos especiales —el dispositivo audio

Muchos usuarios desean conocer sólo lo necesario sobre los detalles de operación de los periféricos, y UNIX proporciona una interfaz independiente de dispositivos al representar éstos como *archivos especiales* que pueden ser abiertos, leídos o escritos como cualquier otro archivo. Los FIFO son archivos especiales. Otro ejemplo es el dispositivo de audio disponible en muchas estaciones de trabajo (el altavoz y el micrófono). La designación de dispositivo para este dispositivo en las estaciones de trabajo Sun es `/dev/audio`. Nota: Si el usuario se da de alta en una terminal ASCII o en una terminal X, entonces no podrá hacer uso del dispositivo de audio aunque el sistema tenga uno.

Ejemplo 3.26

El siguiente comando reproduce el archivo de audio `sample.au` *en el altavoz de una estación de trabajo Sun.*

```
cat sample.au > /dev/audio
```

El programa 3.4 contiene una biblioteca de funciones para leer y escribir en el dispositivo de audio. Ninguna de estas funciones pasan el descriptor de archivo correspondiente al dispositivo de audio. En lugar de ello, la biblioteca de audio es tratada como un objeto que los programas llaman a través de la interfaz proporcionada (`open_audio`, `close_audio`, `read_audio` y `write_audio`).

El `open_audio` abre `/dev/audio` para acceso de lectura/escritura utilizando para ellos E/S con bloqueo. Si el dispositivo de audio ya está abierto, la llamada a `open` se queda pendiente hasta que se cierre el dispositivo. Si el dispositivo de audio hubiera sido abierto con la bandera `O_NONBLOCK`, el `open` habría regresado con un error si el dispositivo se encontraba ocupado.

La función `open_audio` intenta abrir tanto el micrófono como el altavoz. Un proceso que sólo desee grabar puede llamar a `open` con `O_RDONLY`, mientras que otro que sólo quiera reproducir puede llamar a `open` con `O_WRONLY`. Si éste es interrumpido por una señal, `open_audio` vuelve a iniciar el `open`.

Programa 3.4: Objeto del dispositivo de audio y operaciones básicas sobre éste.

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <stropts.h>
#include <errno.h>
#define AUDIO_DEVICE "/dev/audio"

static int audio_fd = -1;    /* descriptor de archivo del dispositivo de audio */

/* Apertura del dispositivo de audio. Se devuelve 0 si hay éxito o -1 en caso
contrario */
int open_audio(void)
{
   while (((audio_fd = open(AUDIO_DEVICE, O_RDWR)) == -1) &&
          (errno == EINTR))
       ;
   if (audio_fd <= 0)
      return -1;
   return 0;
}

/* Cierre del dispositivo de audio */
void close_audio(void)
{
   close(audio_fd);
   audio_fd = -1;
}

/* Lectura del audio hasta de maxcnt bytes.
 * Se devuelve el número de bytes leídos o -1 si ocurre un error.
 */
int read_audio(char *buffer, int maxcnt)
{
   ssize_t bytes;

   while (((bytes = read(audio_fd, buffer, (size_t)maxcnt)) == -1) &&
          (errno == EINTR))
      ;
   return (int)bytes;
}

/* Escritura de length bytes en el buffer de audio.
 * Se devuelve el número de bytes escritos o -1 si ocurre un error.
 */
int write_audio(char *buffer, int length)
{
   ssize_t byteswritten;
```

```
    size_t bytestried;
    char *buffp;

    buffp = buffer;
    bytestried = length;
    while (bytestried != 0) {
        if ((byteswritten = write(audio_fd, buffp, bytestried)) >= 0) {
            bytestried -= byteswritten;
            buffp += byteswritten;
        } else if (errno != EINTR)
            break;
    }
    if (byteswritten == -1)
        return (int)byteswritten;
    else
        return length;
}
```

Programa 3.4

El altavoz puede manejar datos sólo con una rapidez predeterminada, de modo que `write_audio` tal vez no pueda enviar todo el *buffer* al altavoz en una llamada `write`. De manera similar, `read_audio` lee sólo los datos del micrófono que están disponibles y devuelve el número de bytes leídos. La sección 3.13 explora la E/S al dispositivo de audio.

El programa 3.5 lee la entrada de micrófono y escribe en el altavoz. El programa termina al presionar en el teclado la combinación `ctrl-c`. Cuando intente usar este programa, lo mejor será utilizar audífonos para evitar la retroalimentación causada por la cercanía entre el micrófono y el altavoz. El archivo de encabezado `audio.h` contiene los siguientes prototipos de funciones de audio:

```
int open_audio(void);
void close_audio(void);
int read_audio(char *buffer, int maxcnt);
int write_audio(char *buffer, int length);
```

Programa 3.5: Programa simple que lee datos del micrófono.

```
#include <stdio.h>
#include <stdlib.h>
#include "audio.h"

#define BUFSIZE 1024
void main (void)
{
    char buffer[BUFSIZE];
    int bytesread;
```

```
    if (open_audio() == -1) {
        perror("Could not open audio:");
        exit(1);
    }
    for( ; ; ) {
        if ((bytesread = read_audio(buffer, BUFSIZE)) == -1) {
            perror("Could not read microphone");
            break;
        } else if (write_audio(buffer, bytesread) == -1) {
            perror("Could not write to speaker");
            break;
        }
    }
    close_audio();
    exit(0);
}
```

Programa 3.5

Si `BUFSIZE` no es igual al tamaño de los bloques transferidos por el controlador de dispositivo de audio, la voz puede sonar discontinua. Una llamada a `ioctl` permite encontrar el tamaño de *buffer* apropiado, así como otra información específica del dispositivo. La llamada `ioctl` no es parte de POSIX.1, ya que el comité de estándares no ha podido resolver algunos conflictos de especificación entre varias implantaciones históricas de UNIX. Spec 1170 especifica la llamada `ioctl` para realizar el control sobre dispositivos *STREAMS*. (En el capítulo 12 se estudian los STREAMS.)

SINOPSIS

```
#include <stropts.h>

int ioctl(int fildes, int request, .... /* arg */);
```

Spec 1170

La llamada `ioctl` proporciona un medio para obtener información sobre el estado del dispositivo o para establecer opciones de control para el mismo. Sun Solaris 2 utiliza la solicitud `AUDIO_GETINFO` de `ioctl` para proporcionar información sobre el dispositivo de audio. El tipo `audio_info_t` definido en `audioio.h`, guarda información de configuración sobre el dispositivo de audio:

```
typedef struct audio_info {
    audio_prinfo_t play;          /* información sobre el estado de la salida*/
    audio_prinfo_t record;        /* información sobre el estado de la*/
                                  /* entrada*/
    uint_t          monitor_gain; /* entrada al mezclador de salida*/
    uchar_t         output_muted; /* distinto de cero si la salida está*/
                                  /* silenciada*/
    uchar_t _xxx[3];              /* Reservado para uso futuro*/
    uint_t _yyy[3];               /* Reservado para uso futuro*/
}   audio_info_t;
```

donde `audio_prinfo_t` está definido como

```
  Donde audio_prinfo_t se define como:
      /* Los siguientes valores describen la codificación de los datos de audio */
     uint_t    sample_rate;  /* muestras por segundo */
     uint_t    channels;     /* número de canales entrelazados */
     uint_t    precision;    /* número de bits por muestra */
     uint_t    encoding;     /* método de codificación de datos */

     /* Los siguientes valores controlan la configuración del dispositivo de audio */
     uint_t    gain;         /* nivel del volumen */
     uint_t    port;         /* puerto de E/S seleccionado */
     uint_t    avail_ports;  /* puertos de E/S disponibles */
     uint_t    _xxx[2];      /* Reservado para uso futuro */
     uint_t    buffer_size;  /* tamaño del buffer de E/S */

     /* Los siguientes valores describen el estado en que se encuentra el dispositivo */
     uint_t    samples;      /* número de muestras convertidas */
     uint_t    eof;          /* contador End Of File (sólo reproducción) */
     uchar_t   pause;        /* distinto de cero si hay pausa, cero para continuar */
     uchar_t   error;        /* distinto de cero si overflow/underflow */
     uchar_t   waiting;      /* distinto de cero si un proceso desea tener acceso */
     uchar_t   balance;      /* balance del canal estéreo */
     ushort_t  minordev;
       /* Los siguientes valores son las banderas únicamente de lectura del estado del
       dipositivo */
     uchar_t   open;         /* distinto de cero si se permite el acceso */
     uchar_t   active;       /* distinto de cero si E/S está activa */
} audio_prinfo_t;
```

El miembro `buffer_size` de la estructura `audio_prinfo_t` indica el tamaño del bloque de datos de audio que el controlador de dispositivo acumula antes de pasar los datos a la solicitud de lectura. El `buffer_size` para la reproducción indica el tamaño del bloque de datos que el controlador de dispositivo acumula antes de enviar los datos al altavoz. El audio tiende a sonar mejor si el programa envía y recibe bloques que concuerden con los valores de `buffer_size` correspondientes. En un programa de aplicación de audio, utilícese `ioctl` para determinar estos tamaños.

Ejemplo 3.27

La función `get_record_buffer_size` *devuelve el tamaño apropiado del bloque que debe emplearse cuando se hace la lectura de un micrófono o* `-1` *si se presenta un error. Esta función puede añadirse a la biblioteca de audio del programa 3.4.*

```
#include <unistd.h>
#include <sys/audioio.h>

int get_record_buffer_size(void)
{
   audio_info_t myaudio;
```

```
    if (ioctl(audio_fd, AUDIO_GETINFO, &myaudio) == -1)
        return -1;
    else
        return myaudio.record.buffer_size;
}
```

La implantación del programa 3.4 abre el dispositivo de audio para E/S con bloqueo. Las lecturas sin bloqueo son complicadas por el hecho de que `read` puede devolver `-1`, ya sea si existe un error o si el dispositivo de audio no tiene listos los datos. Este último caso tiene un valor de `errno` igual a `EAGAIN` y no debe considerarse como un error. La razón principal para abrir el dispositivo de audio en modo de no bloqueo es que el `open` no quede congelado cuando el dispositivo ya se encuentre abierto. Una alternativa es abrir el dispositivo de audio en modo no-bloqueo y luego hacer uso de `fcntl` para cambiar el modo a bloqueo.

Ejemplo 3.28

El siguiente segmento de código abre el dispositivo de audio para E/S sin bloqueo. A continuación lee `BLKSIZE` *bytes del dispositivo de audio y coloca los datos en un* buffer.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#define AUDIO_DEVICE "/dev/audio"

char buffer[BLKSIZE];
char *bp;
int bytes_read;
unsigned bytes_needed;
int audio_fd;

if ( (audio_fd = open(AUDIO_DEVICE, O_NONBLOCK | O_RDWR)) == -1) {
    perror("No fue posible abrir el dispositivo de audio");
    exit(1);
}

bp = buffer;
bytes_needed = BLKSIZE;
while(bytes_needed != 0) {
    bytes_read = read(audio_fd, bp, bytes_needed);
    if ((bytes_read == -1) && (errno != EAGAIN))
        break;
    if (bytes_read > 0) {
        bp += bytes_read;
        bytes_needed -= bytes_read;
    }
}
```

Al probar programas de audio, tenga en cuenta que el dispositivo de audio se cierra cuando la ejecución del programa termina. Si aún quedan datos en el *buffer* de audio que no han llegado al altavoz, éstos pueden perderse. La limpieza de un dispositivo después de un `close` depende del sistema, de modo que debe leerse la página del manual para decidir cómo manejar esta situación.

3.11 Ejercicio: recorrido de directorios

Los ejercicios de esta sección desarrollan programas para recorrer árboles de directorio de manera vertical u horizontal. La búsqueda vertical explora cada rama del árbol hasta sus hojas antes de buscar en otras ramas. La búsqueda horizontal explora todos los nodos hasta un nivel dado, antes de descender por el árbol.

Ejemplo 3.29

Para el árbol de directorio de la figura 3.1, el ordenamiento vertical visita los nodos en el siguiente orden.

```
dirC
   my3.dat
dirA
   dirB
      my1.dat
   my1.dat
   my2.dat
```

El sangrado en los nombres de los archivos del ejemplo 3.29 muestra el nivel en el árbol de directorio.

Ejercicio 3.13

El comando `du` de UNIX muestra el tamaño de los directorios en un árbol de directorio. Intente determinar el orden de búsqueda que emplea el comando.

La búsqueda vertical es de naturaleza recursiva, tal como lo indica el siguiente pseudocódigo:

```
depth_first_search_tree(root) {
    for each node at or below root
        visit node;
        if node is a directory
           depth_first_search_tree(node);
}
```

Ejemplo 3.30

Para el árbol de directorio de la figura 3.1, el orden horizontal visita los nodos en el siguiente orden.

```
/
/dirC
/dirA
/dirC/my3.dat
/dirA/dirB
/dirA/my1.dat
/dirA/my2.dat
/dirA/dirB/my1.dat
```

La búsqueda horizontal puede implantarse utilizando una cola similar a la del programa 2.3. Conforme el programa encuentra cada nodo del directorio en un nivel particular, pone en la cola el nombre completo de la ruta de acceso para examinarla posteriormente. El siguiente pseudocódigo supone la existencia de una cola. La operación `enqueue` pone un nodo al final de la cola, mientras que `dequeue` quita uno del frente de la cola.

```
breath_first_tree_search(root) {
  enqueue(root);
  while (queue is not empty) {
       dequeue (&next);
       for each node directly below next:
       visit the node
       if node is a directory
            enqueue(node)
  }
}
```

- Escriba una función, `isadirectory`, que tenga como prototipo

```
int isadirectory(char *pathname);
```

  La función `isadirectory` utiliza a `stat` para determinar si el archivo especificado por `pathname` es un directorio. La función devuelve `1` si `pathname` es un directorio y `0` si no lo es.

- Escriba una función, `depth_first_apply`, que tenga como prototipo

```
int depth_first_apply(char *pathname,
                                    int pathfun(char *pathname1));
```

  La función `depth_first_apply` recorre el árbol de directorio comenzando en `pathname`. La función aplica la rutina `pathfun` a cada archivo que encuentra en el recorrido. `depth_first_apply` devuelve la suma de los valores devueltos por `pathfun`, o `-1` si falla al recorrer cualquiera de los subdirectorios del directorio. Un ejemplo de una posible rutina `pathhfun` es una función que imprime el nombre de la ruta de acceso con otra información de `stat` en un orden particular. `pathfun` devuelve el tamaño del archivo.

- Escriba una función `sizepathfun` con prototipo

```
int sizepathfun(char *pathname1);
```

La función `sizepathfun` devuelve el tamaño en bloques del archivo especificado por `pathname1` o `-1` si éste no corresponde a un archivo ordinario.

- Utilice `depth_first_apply` con la `pathfun` dada por `sizepathfun` para implantar el comando

```
showtreesize pathname
```

El comando `showtreesize` escribe en la salida estándar `pathname` seguido del tamaño total de éste. Si `pathname` es un directorio, el tamaño total corresponde al tamaño de todo el subárbol que tiene como raíz a `pathname`. Si `pathname` es un archivo especial, se imprime un mensaje informativo pero no el tamaño de éste.

- Escriba un comando, `mydu`, que se invoque con el argumento de línea comando `rootpath`:

```
mydu rootpath
```

El programa `mydu` llama a una función `depth_first_apply` modificada con la función `sizepathfun`. El comando imprime el tamaño de cada directorio seguido de su nombre de ruta de acceso. El tamaño del directorio no cuenta el tamaño de los subárboles de ese directorio. Para terminar, el programa imprime el tamaño total del árbol.

- Escriba la función `breadth_first_apply`, la cual es similar a `depth_first_apply` pero hace uso de la estrategia de búsqueda horizontal.

3.12 Ejercicio: sistema de archivo `proc`

Algunas implantaciones de UNIX, tales como Sun Solaris 2, proporcionan un sistema de archivo `proc` que transforma la imagen de cada proceso en el sistema en un archivo ubicado en el directorio `/proc`. Cada proceso tiene un archivo asociado con él cuyo nombre está dado por su ID de proceso. Los ejercicios de esta sección suponen un sistema con un sistema de archivo `proc` similar al apoyado por Sun Solaris.

Ejemplo 3.31

La siguiente impresión parcial del comando `ls -l /proc` *muestra los procesos junto con sus propietarios, tamaño, hora de creación e ID de proceso.*

```
total 396776
-rw-------   1 root     root           0 Jan 24 12:39 00000
-rw-------   1 root     root      724992 Jan 24 12:40 00001
-rw-------   1 root     root           0 Jan 24 12:40 00002
```

```
-rw-------   1 root     root           0 Jan 24 12:40 00003
-rw-------   1 root     root     1736704 Jan 24 12:40 00297
-rw-------   1 root     root     1236992 Jan 24 12:40 00299
-rw-------   1 root     root     1351680 Jan 24 12:40 00305
-rw-------   1 root     root     1454080 Jan 24 12:40 00314
-rw-------   1 root     root     1294336 Jan 24 12:50 00575
-rw-------   1 robbins  staff     675840 Feb  8 08:07 10522
-rw-------   1 robbins  staff    1449984 Feb  8 08:07 10526
```

La impresión del ejemplo 3.31 indica que el proceso `init` con ID de proceso igual a `1` inició el 24 de enero a las 12:40. El usuario `robbins` tiene dos procesos que dieron inicio el 8 de febrero a las 8:07.

En sistemas que cuentan con un sistema de archivo `proc`, un programa puede utilizar la interfaz de llamadas del sistema estándar, `open`, `close`, `read`, `write` e `ioctl`, para analizar o modificar la imagen de un proceso. El programa puede llevar a cabo varias funciones de control al llamar a `ioctl`. Los encabezados específicos requeridos por el sistema de archivo `proc` de Sun Solaris son

```
#include <sys/types.h>
#include <sys/signal.h>
#include <sys/fault.h>
#include <sys/syscall.h>
#include <sys/procfs.h>
```

Sun Solaris utiliza la estructura `pr_status_t` mostrada en la figura 3.21 para guardar información sobre un proceso, representado bajo el sistema de archivo `proc`. El `prstatus_t` de la figura 3.21 proporciona información detallada sobre la imagen del proceso. Muchos de los campos son explicables por sí mismos. Los valores posibles de los `campos pr_flags`, `pr_why` y `pr_what` están relacionados con el estado del proceso y están explicados en las páginas del manual.

Ejemplo 3.32

En el siguiente fragmento de código, la solicitud `PIOCSTATUS` *de* `ioctl` *llena una estructura de tipo* `prstatus_t` *para el proceso correspondiente especificado por* `fd`.

```
#include <sys/types.h>
#include <sys/signal.h>
#include <sys/fault.h>
#include <sys/syscall.h>
#include <sys/procfs.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define NAMESIZE 25

prstatus_t my_status;
```

```
typedef struct prstatus {
    long            pr_flags;       /* Banderas */
    short           pr_why;         /* Motivo del stop (suspensión, si es que hay) */
    short           pr_what;        /* Razón más detallada */
    id_t            pr_who;         /* Identificador lwp específico */
    u_short         pr_nlwp;        /* Número de lwps en el proceso */
    short           pr_cursig;      /* Señal presente */
    sigset_t        pr_sigpend;     /* Conjunto de señales pendientes del proceso */
    sigset_t        pr_lwppend;     /* Conjunto de señales pendientes lwp del proceso */
    sigset_t        pr_sighold;     /* Conjunto de señales lwp retenidas */
    struct siginfo  pr_info;        /* Información asociada con una señal o falla */
    struct sigaltstack  pr_altstack;/* Pila de información alterna de la señal */
    struct sigaction  pr_action;    /* Señal de acción para la señal presente */
    struct ucontext *pr_oldcontext; /* Dirección del ucontext previo */
    caddr_t         pr_brkbase;     /* Dirección del heap del proceso */
    u_long          pr_brksize;     /* Tamaño en bytes del heap del proceso */
    caddr_t         pr_stkbase;     /* Dirección de la pila del proceso */
    u_long          pr_stksize;     /* Tamaño en bytes de la pila del proceso */
    short           pr_syscall;     /* Número de llamada al sistema (en syscall) */
    short           pr_nsysarg;     /* Número de argumentos para esta syscall */
    long            pr_sysarg[PRSYSARGS];   /* Argumentos de esta syscall */
    caddr_t         pr_brkbase;     /* Dirección del heap del proceso */
    u_long          pr_brksize;     /* Tamaño en bytes del heap del proceso */
    caddr_t         pr_stkbase;     /* Dirección de la pila del proceso */
    u_long          pr_stksize;     /* Tamaño en bytes de la pila del proceso */
    short           pr_syscall;     /* Número de llamada al sistema (en syscall) */
    short           pr_nsysarg;     /* Número de argumentos para esta syscall */
    long            pr_sysarg[PRSYSARGS];   /* Argumentos de esta syscall */
    pid_t           pr_pid;         /* id del proceso */
    pid_t           pr_ppid;        /* id del proceso padre */
    pid_t           pr_pgrp;        /* id del grupo del proceso */
    pid_t           pr_sid;         /* id de sesión */
    timestruc_t     pr_utime;       /* Tiempo de CPU del usuario del proceso */
    timestruc_t     pr_stime;       /* Tiempo de CPU del proceso del sistema */
    timestruc_t     pr_cutime;      /* Suma de los tiempos de usuario del hijo */
    timestruc_t     pr_cstime;      /* Suma de los tiempos de sistema del hijo */
    char            pr_clname[PRCLSZ];  /* Nombre de la clase de planificación */
    long            pr_instr;       /* Instrucción en ejecución */
    prgregset_t     pr_reg;         /* Registros generales */
} prstatus_t;
```

Figura 3.21: Información guardada en `prstatus_t` en Sun Solaris 2.

```
char procname[NAMESIZE];
int fd;

sprintf(procname,"/proc/%05ld", (long)getpid());
if ((fd = open(procname, O_RDONLY)) == -1)
   perror("Could not open my process image");
else if (ioctl(fd, PIOCSTATUS, &my_status) == -1)
   perror("Could not get my process status");
```

Haga los siguientes ejercicios para familiarizarse con el sistema de archivo `proc`.

- Busque `proc` en las páginas del manual. Allí se encuentran todos los detalles.
- Escriba la función `get_prstatus` con prototipo

```
int get_prstatus(int pid, prstatus_t *sp)
```

El `get_prstatus` inicializa la estructura `prstatus_t` a la que apunta `sp` para el proceso cuyo ID de proceso es `pid`. La función devuelve 0 si tiene éxito, o -1 si falla. El esbozo del procedimiento `get_prstatus` es

  * Abrir `/proc/pid` sólo para lectura.
  * Ejecutar la llamada al sistema `ioctl(fildes, code, s)` con `code` igual a `PIOCSTATUS`, donde `s` es un apuntador a una estructura `prstatus_t`.
  * Cerrar el archivo.

Verifique si hay errores en cada una de las llamadas al sistema en `get_prstatus`. No salga de la función inmediatamente después de la ocurrencia de un error—devuelva un código de error. Asegúrese de cerrar los archivos que están abiertos, incluso si hay error.

- Escriba la función `output_prstatus` con prototipo

```
int output_prstatus(FILE *fp, prstatus_t s);
```

La función `output_prstatus` escribe el contenido de `s` en el archivo `fp`, con un formato fácil de leer. Por el momento el lector no debe preocuparse por los miembros de tipo `sigset_t`. La sección 5.11 estudia dichos miembros en una extensión de este ejercicio.

3.13 Ejercicio: audio

Los ejercicios de esta sección extienden y mejoran la biblioteca de audio del programa 3.4.

- Añada las siguientes funciones de acceso al objeto audio del programa 3.4:
  - La función `play_file` reproduce un archivo de audio. El prototipo es

```
int play_file(char *filename);
```

La función `play_file` envía el archivo de audio especificado por `filename` al dispositivo de audio, suponiendo que el altavoz ya está abierto. Si tiene éxito, la función devuelve el número total de bytes enviados, o `-1` si ocurre un error.

  - La función `record_file` guarda la entrada de audio en un archivo en disco. El prototipo es

```
int record_file(char *filename, int seconds);
```

La función `record_file` guarda información de audio por un lapso dado por `seconds` en el archivo especificado por `filename`, *suponiendo que el* micrófono ya está abierto. Si tiene éxito, la función devuelve el número de bytes grabados; de lo contrario, regresa –1.

- La función `get_record_sample_rate` determina la frecuencia de muestreo para la grabación. El prototipo es

```
int get_record_sample_rate(void);
```

La función `get_record_sample_rate` devuelve la frecuencia de muestreo utilizada para hacer la grabación, o –1 si ocurre un error.

- La función `get_play_sample_rate` determina la frecuencia de muestreo para la reproducción. El prototipo es

```
int get_play_sample_rate(void);
```

La función `get_play_sample_rate` devuelve la frecuencia de muestreo empleada para reproducir archivos de audio en el altavoz, o –1 si ocurre un error. 8000 muestras/segundo se consideran como calidad de voz.

- Utilice la función `record_file` para crear ocho archivos de audio con una duración, cada uno, de 10 segundos: `pid1.au`, `pid2.au`, y así sucesivamente. En el archivo `pid1.au` grabe (con su propia voz) el siguiente mensaje: "Soy el proceso 1 transmitiendo a la salida de error estándar." Grabe mensajes similares en los demás archivos. Reproduzca los archivos utilizando la función `play_file`.

- Asegúrese de crear un archivo de encabezado (`audio.h`) con los prototipos de las funciones de la biblioteca de audio. Incluya este archivo en cualquier programa que llame a las funciones de esta biblioteca.

- Vuelva a diseñar la representación del objeto audio y las funciones de acceso de modo que los procesos tengan la opción de abrir por separado para lectura y escritura. Reemplace `audio_fd` con los descriptores `play_fd` y `record_fd`. Modifique `open_audio` de modo que ésta inicialice a `play_fd` y `record_fd` con el valor del descriptor de archivo devuelto por el `open`. Añada las siguientes funciones de acceso al objeto audio del programa 3.4.

  - La función `open_audio_for_record` abre el dispositivo de audio para lectura (`O_RDONLY`). El prototipo es

```
int open_audio_for_record(void);
```

Si tiene éxito la función devuelve 0, o –1 si ocurre un error.

  - La función `open_audio_for_play` que abre el dispositivo de audio para escritura (`O_WRONLY`). El prototipo es

```
int open_audio_for_play(void);
```

La función `open_audio_for_play` devuelve 0 si tiene éxito, o -1 si ocurre un error.

- Grabe con su voz la pronunciación de los dígitos numéricos (del 0 al 9) en diez archivos diferentes. Escriba una función `speak_number` que tome una cadena de caracteres que representan un entero y que dé salida en voz al número que corresponde a la cadena llamando para ello a la función `play_file` para que reproduzca los archivos de dígitos. (¿Cómo será el sonido creado por el programa comparado con los mensajes generados por computadora de la compañía de teléfonos?)

- Reemplace la instrucción `fprintf` que imprime varios ID del programa 2.12 con una llamada a `play_file`. Para el proceso que tiene una `i` con el valor 1, reproduzca el archivo `pid1.au`, y así sucesivamente. Escuche los resultados para diferentes números de procesos cuando el altavoz esté abierto antes del lazo `fork`. ¿Qué sucede cuando el altavoz se abre después del `fork`? Asegúrese de hacer uso de `sprintf` para construir los nombres de los archivos a partir del valor de `i`. No ponga en el código del programa los nombres de los archivos.

- Grabe del siguiente mensaje en el archivo `pid.au`: "Mi ID de proceso es." En lugar de que cada proceso de la parte anterior reproduzca el archivo `pidi.au` que corresponde a su número `i`, utilice `speak_number` para dar salida en voz al ID del proceso. Maneje los ID del padre y el hijo de manera similar.

- Añada las siguientes funciones a la biblioteca de audio:

  - La función `set_play_volume` cambia el volumen con que se reproduce el sonido en el altavoz. El prototipo es

    ```
    int set_play_volume(double volume);
    ```

    Esta función indica el volumen del altavoz. `volume` debe estar entre 0.0 y 1.0. La función devuelve 0 si tiene éxito, o –1 si ocurre un error.

  - `set_record_volume` cambia el volumen del sonido que proviene del micrófono. El prototipo es

    ```
    int set_record_volume(double volume);
    ```

    La función fija el volumen del micrófono. `volume` debe tener un valor entre 0.0 y 1.0. La función devuelve 0 si tiene éxito y –1 si ocurre un error.

3.14 Ejercicio: control de la terminal

Si no tiene acceso a un dispositivo de audio, todavía puede experimentar con el control del dispositivo. POSIX.1 no incluye a `ioctl`. Puesto que el control de la terminal fue considerado como algo esencial, el comité de estándares POSIX.1 incluyó la siguiente biblioteca de funciones para manejar las características de las terminales y los puertos de comunicación asíncronos.

```
#include <termios.h>
int tcgetattr(int fildes, struct termios *termios_p);
int tcsetattr(int fildes, int optional_actions,
      const struct termios *termios_p);
int tcsendbreak(int fildes, int duration);
int tcdrain(int fildes);
int tcflush(int fildes, int queue_selector);
int tcflow(int fildes, int action);
speed_t cfgetospeed(const struct termios *termios_p);
int cfsetospeed(struct termios *termios_p, speed_t speed);
speed_t cfgetispeed(const struct termios *termios_p);
int cfsetispeed(struct termios *termios_p, speed_t speed);
```

La estructura `struct termios` incluye los siguientes miembros:

```
tcflag_t     c_iflag;       /* modos de entrada */
tcflag_t     c_oflag;       /* modos de salida */
tcflag_t     c_cflag;       /* modos de control */
tcflag_t     c_lflag;       /* modos locales */
cc_t         c_cc[NCCS];    /* caracteres de control */
```

Existen varias funciones adicionales que manipulan el grupo del proceso y el ID de sesión de una terminal. Se emplean estas llamadas para el control de trabajos.

SINOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t tcgetpgrp(int fildes);
int tcsetpgrp(int fildes, pid_t pgid);
```

POSIX.1, Spec 1170

SINOPSIS

```
#include <termios.h>

pid_t tcgetsid(int fildes);
```

Spec 1170

Haga lo siguiente para familiarizarse con el control de la terminal.

- Lea la página del manual que corresponde a `termio`.
- Ejecute `stty -a` e intente comprender los diferentes campos.
- Compare los servicios específicos proporcionados por las llamadas específicas de la terminal con los ofrecidos por `ioctl`. Para mayor información, lea sobre `termio` en la sección 7 de las páginas del manual.

3.15 Lecturas adicionales

Una buena fuente de información actualizada sobre herramientas y enfoques que evolucionan bajo UNIX es *USENIX Conference Proceedings. Operating Systems Review* es una publicación informal de SIGOPS, que es la Association for Computing Machinary Special Interest Group on Operating Systems. En ocasiones la *Operating Systems Reviews* presenta artículos sobre desarrollos recientes en el área de sistemas de archivo y administración de dispositivos.

Advanced Programming in the UNIX Environment de Stevens [86] ofrece algunos interesantes estudios de caso sobre el control de dispositivos a nivel de usuario, incluyendo un programa para controlar una impresora postscript, un marcador de número para módem y un programa de administración de pseudoterminal. El *Data Communications Networking Devices* de Held [38] es una referencia general acerca de la administración de dispositivos de red. Finalmente, *SunOS 5.3 Writing Device Drivers* es una guía muy técnica para implantar controladores para dispositivos de bloque y orientados a caracteres bajo el control de Solaris [83].

Capítulo 4

Proyecto: *Anillo de procesos*

Este proyecto explora los entubamientos (*pipes*), las bifurcaciones (*forks*) y el redireccionamiento en el contexto de un anillo de procesos. El anillo permite hacer simulaciones sencillas e interesantes de las topologías de una red de anillo. El capítulo también presenta ideas fundamentales de algoritmos distribuidos, incluye modelos de procesamiento, el funcionamiento con entubamientos y el cómputo en paralelo.

La topología de anillo es una de las configuraciones más simples y baratas para conectar entidades que se comunican entre sí. La figura 4.1 ilustra una estructura de anillo unidireccional. Cada entidad tiene una conexión para entrada y otra para salida. La información circula alrededor del anillo en la dirección del giro de las manecillas del reloj. Los anillos son atractivos debido a que los costos de entubamiento sobre el anillo aumentan de manera lineal, de hecho sólo se necesita una conexión adicional para cada nodo que se agrega al anillo. La latencia se incrementa a medida que aumenta el número de nodos, ya que el tiempo que requiere un mensaje para viajar por el anillo va en aumento. En muchos casos la rapidez con la que los nodos pueden leer la información del anillo o escribirla en él, no cambia al aumentar el tamaño de éste, de modo que el ancho de banda es independiente del tamaño del anillo. Existen varios estándares de red, incluidos el *token ring* (IEEE 802.5), el *token bus* (IEEE 802.4) y el *FDDI*, que están basados en la conectividad de un anillo.

En este capítulo se analiza la implantación de un anillo de procesos que se comunican mediante entubamientos (*pipes*) o interconexiones. Los procesos representan nodos en el anillo. Cada proceso lee información de la entrada estándar, y escribe en la salida estándar. El proceso $n - 1$ redirige su salida estándar a la entrada estándar del proceso n mediante un entubamiento. Una vez establecida la estructura del anillo, el proyecto puede extenderse para simular los estándares de red o implantar algoritmos para exclusión mutua y selección del líder con base en la arquitectura del anillo.

La sección 4.1 presenta el desarrollo paso por paso de un anillo de procesos sencillo conectado por entubamiento (*pipes*). La sección 4.2 pone a prueba la conectividad y operación del

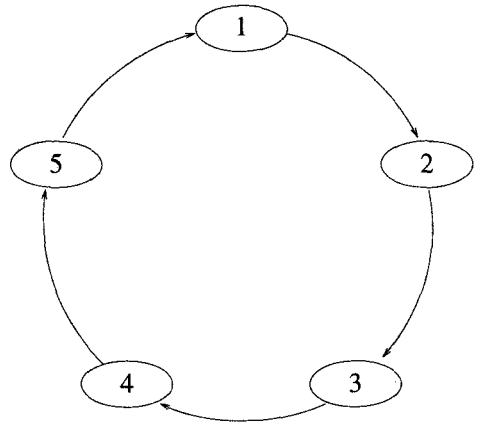


Figura 4.1: Anillo unidireccional con cinco nodos.

anillo al pedir a éste que genere una secuencia de Fibonnaci. Las secciones 4.3 y 4.4 presentan dos enfoques alternativos para proteger las secciones críticas del anillo. Las demás secciones del capítulo describen extensiones que exploran diferentes aspectos de la comunicación por redes, el procesamiento distribuido y los algoritmos paralelos. Las extensiones en una sección particular son independientes de las descritas en otras secciones.

4.1 Formación del anillo

En esta sección se desarrolla un anillo de procesos que comienza con un anillo que contiene un solo proceso. En las figuras se usará un `[0]` para designar la entrada estándar y un `[1]` para indicar la salida estándar. Asegúrese de seguir el estándar POSIX y utilizar `STDIN_FILENO` y `STDOUT_FILENO` cuando haga referencia a estos descriptores de archivo en el código del programa.

Ejemplo 4.1

El siguiente segmento de código conecta la salida estándar de un proceso con su entrada estándar por medio de un entubamiento. Para que resulte más sencillo, se omite el código que verifica la existencia de errores.

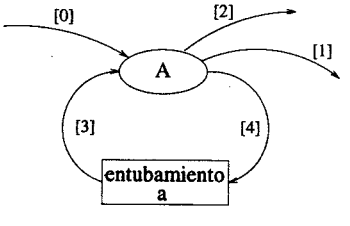
```
#include <unistd.h>
int fd[2];

pipe(fd);
dup2(fd[0], STDIN_FILENO);
dup2(fd[1], STDOUT_FILENO);
close(fd[0];
close(fd[1];
```

Las figuras 4.2 a la 4.4 ilustran el estado del proceso del ejemplo 4.1. Los valores numéricos encerrados entre paréntesis rectangulares (por ejemplo, `[0]`) representan índices de la tabla de descriptores de archivo del proceso. Las entradas en esta tabla son apuntadores dirigidos a entradas en la tabla de archivos del sistema. Por ejemplo, *el entubamiento a escribe* en la entrada `[4]` significa "un apuntador a la entrada de escritura en la tabla de archivos del sistema para *el entubamiento a*" y una *entrada estándar* en la entrada `[0]` significa "un apuntador a la entrada de la tabla de archivos del sistema que corresponde al dispositivo preestablecido para la entrada estándar" usualmente, el teclado.

La figura 4.2 muestra la tabla de descriptores de archivo después de la creación del *entubamiento (pipe) a*. Las entradas `[3]` y `[4]` de descriptores de archivo apuntan a las entradas de la tabla de archivos del sistema que fueron creadas por la llamada `pipe`. En este momento un programa puede escribir al entubamiento mediante el empleo del valor del descriptor de archivo `4` en la llamada `write`.

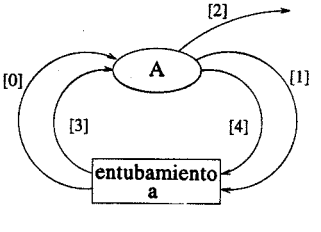
La figura 4.3 muestra el estado de la tabla de descriptores de archivo después de las llamadas a `dup2`. En este momento el programa puede escribir en el entubamiento utilizando `1` o `4` como valor para el descriptor de archivo. La figura 4.4 muestra la configuración después de cerrar los descriptores `3` y `4`.



Proceso A
tabla de descriptores de archivo

[0]	*entrada estándar*
[1]	*salida estándar*
[2]	*error estándar*
[3]	*entubamiento a de lectura*
[4]	*entubamiento a de escritura*

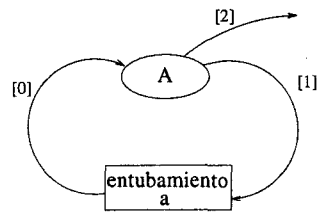
Figura 4.2: Estado del proceso del ejemplo 4.1 después de la ejecución de la instrucción `pipe(fd)`.



Proceso A
tabla de descriptores de archivo

[0]	*entubamiento a de lectura*
[1]	*entubamiento a de escritura*
[2]	*error estándar*
[3]	*entubamiento a de lectura*
[4]	*entubamiento a de escritura*

Figura 4.3: Estado del proceso del ejemplo 4.1 después de la ejecución de las dos instrucciones `dup2`.



Proceso A
tabla de descriptores de archivo

[0]	*entubamiento a de lectura*
[1]	*entubamiento a de escritura*
[2]	*error estándar*

Figura 4.4: Estado del proceso del ejemplo 4.1 después de la ejecución de las instrucciones `close(fd[0])` y `close(fd[1])`.

Ejercicio 4.1

¿Qué sucede si, después de conectar la salida estándar con la entrada estándar por medio de un entubamiento, el proceso del ejemplo 4.1 ejecuta el siguiente fragmento de código?

```
#include <unistd.h>
#include <stdio.h>
int i;
int myint;

for (i = 0; i < 10; i++) {
    write (STDOUT_FILENO, &i, sizeof(i));
    read(STDIN_FILENO, &myint, sizeof(myint));
    fprintf(stderr, "%d\n", myint);
}
```

Respuesta:

El segmento de código genera, como salida en la pantalla, los enteros del 0 al 9 (suponiendo que el dispositivo de error estándar se visualiza en la pantalla).

Ejercicio 4.2

¿Qué sucede con el código del ejercicio 4.1 si se reemplaza con el siguiente código?

```
#include <unistd.h>
#include <stdio.h>
int i;
int myint;

for (i = 0; i < 10; i++) {
    read (STDIN_FILENO, &myint, sizeof(myint));
    write(STDOUT_FILENO, &i, sizeof(i));
    fprintf(stderr, "%d\n", myint);
}
```

Respuesta:

El programa se queda congelado en el primer `read` debido a que aún no se ha escrito nada en el entubamiento.

Ejercicio 4.3

¿Qué sucede si el código del ejercicio 4.1 se reemplaza por el siguiente?

```
#include <unistd.h>
#include <stdio.h>
int i;
int myint;

for (i = 0; i < 10; i++) {
   printf("%d ", i);
   scanf("%d", &myint);
   fprintf(stderr, "%d\n", myint);
```

Respuesta:

El programa se queda congelado en `scanf`, ya que tanto la lectura como la escritura se hacen mediante *buffers*. La función `printf` no escribe nada en el entubamiento hasta que el *buffer* esté lleno. Para obtener la salida es necesario poner una instrucción `fflush(stdout)` después de `printf`.

Ejemplo 4.2

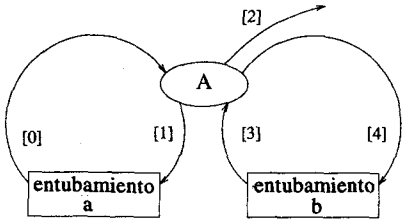
El siguiente fragmento de código crea un anillo con dos procesos.

```
#include <unistd.h>
int fd[2];
pid_t haschild;

pipe(fd);
dup2(fd[0], STDIN_FILENO);
dup2(fd[1], STDOUT_FILENO);
close(fd[0];
close(fd[1];
pipe(fd);
if (haschild = fork())
   dup2(fd[1], STDOUT_FILENO); /* el padre redirige la salida std */
else
   dup2(fd[0], STDIN_FILENO); /* el hijo redirige la entrada std */
close(fd[0]);
close(fd[1]);
```

El proceso padre del ejemplo 4.2 redirige la entrada estándar del primer entubamiento hacia la salida estándar del hijo y redirige la salida estándar a través del segundo entubamiento a la entrada estándar del hijo. Las figuras 4.5 a la 4.8 ilustran el mecanismo de conexión. La figura 4.5 muestra la tabla de descriptores de archivo después de que el proceso padre A crea un segundo entubamiento. La figura 4.6 muestra la situación después de la bifurcación al hijo B.

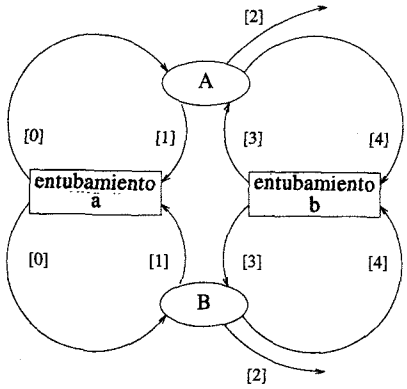
En este momento no se ha ejecutado todavía el `dup2` después del segundo entubamiento (`pipe`). La figura 4.7 muestra la situación después de que el padre y el hijo han ejecutado su último `dup2`. El proceso A ha redirigido su salida estándar para escribir en el entubamiento b, mientras que el proceso B ha redirigido su entrada estándar para leer el entubamiento b. Finalmente, la figura 4.8 muestra el estado de los descriptores de archivo después del cierre de todos los descriptores innecesarios y de la formación de un anillo con dos procesos.



Proceso A
tabla de descriptores de archivo

[0]	*entubamiento a de lectura*
[1]	*entubamiento a de escritura*
[2]	*error estándar*
[3]	*entubamiento b de lectura*
[4]	*entubamiento b de escritura*

Figura 4.5: Conexiones con el proceso padre del ejemplo 4.2 después de la ejecución de la segunda instrucción `pipe(fd)`.



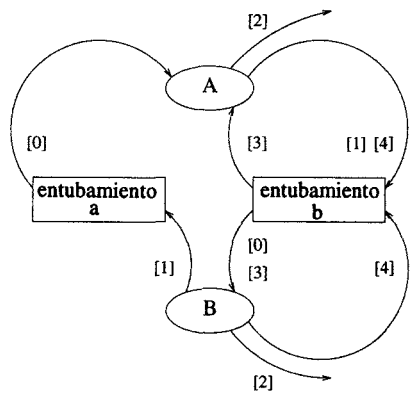
Proceso A
tabla de descriptores de archivo

[0]	*entubamiento a de lectura*
[1]	*entubamiento a de escritura*
[2]	*error estándar*
[3]	*entubamiento b de lectura*
[4]	*entubamiento b de escritura*

Proceso B
tabla de descriptores de archivo

[0]	*entubamiento a de lectura*
[1]	*entubamiento a de escritura*
[2]	*error estándar*
[3]	*entubamiento b de lectura*
[4]	*entubamiento b de escritura*

Figura 4.6: Conexión de los procesos del ejemplo 4.2 después de la ejecución del `fork` pero antes del resto de la proposición `if`. El proceso A es el padre y el B es el hijo.



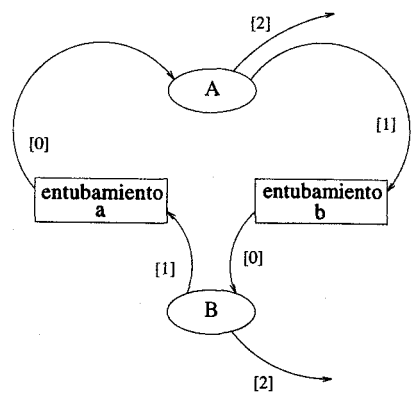
Proceso A
tabla de descriptores de archivo

[0]	*entubamiento a de lectura*
[1]	*entubamiento b de escritura*
[2]	*error estándar*
[3]	*entubamiento b de lectura*
[4]	*entubamiento b de escritura*

Proceso B
tabla de descriptores de archivo

[0]	*entubamiento b de lectura*
[1]	*entubamiento a de escritura*
[2]	*error estándar*
[3]	*entubamiento b de lectura*
[4]	*entubamiento b de escritura*

Figura 4.7: Conexiones de los procesos del ejemplo 4.2 después de la ejecución de la proposición `if`. El proceso A es el padre y el B, el hijo.



Proceso A
tabla de descriptores de archivo

[0]	*entubamiento a de lectura*
[1]	*entubamiento b de escritura*
[2]	*error estándar*

Proceso B
tabla de descriptores

[0]	*entubamiento b de lectura*
[1]	*entubamiento a de escritura*
[2]	*error estándar*

Figura 4.8: Conexiones de los procesos del ejemplo 4.2 después de la ejecución del segmento de código. El proceso A es el padre y el B el hijo.

El código del ejemplo 4.2 para formar un anillo con dos procesos puede extenderse con facilidad para anillos de tamaño arbitrario. El programa 4.1 construye un anillo de n procesos, donde n es un parámetro que se pasa como argumento de la línea comando (y que se convierte en la variable `nprocs`). Se requiere un total de n entubamientos (pipes). Sin embargo, nótese que el programa sólo requiere un arreglo de tamaño 2 más que de uno de $2n$ para guardar todos los descriptores de archivo. (Intente escribir su propio código antes de examinar el programa que crea el anillo).

Programa 4.1: Programa para crear un anillo de procesos.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
/*
 * Programa muestra en C para generar un anillo unidireccional de procesos.
 * Utilice este programa con un argumento de línea comando que indique el
 * número de procesos sobre el anillo. La comunicación se hace a través de
   entubamientos
 * que conectan la salida estándar de un proceso con la entrada estándar
 * de su sucesor en el anillo. Después de la creación del anillo, cada
 * proceso se identifica a sí mismo con su ID de proceso y el ID del proceso
 * de su padre. Después cada proceso termina.
 */

void main(int argc, char *argv[ ])
{
   int  i;          /* número de este proceso (comenzando con 1) */
   int  childpid;   /* indica los procesos que deben generar otros */
   int  nprocs;     /* número total de procesos en el anillo */
   int  fd[2];      /* descriptores de archivo devueltos por un entubamiento */
   int  error;      /* valor devuelto por la llamada a dup2 */
    /* se verifica que la línea comando contenga un número válido de procesos
       a generar */
   if ( (argc != 2) || ((nprocs = atoi (argv[1])) <= 0) ) {
       fprintf (stderr, "Usage: %s nprocs\n", argv[0]);
       exit (1);
   }
                       /* se conecta la entrada estándar con la salida
                          estándar mediante un entubamiento */
   if (pipe (fd) == -1) {
       perror("Could not create pipe");
       exit(1);
   }
   if ((dup2 (fd[0], STDIN_FILENO) == -1) ||
       (dup2 (fd[1], STDOUT_FILENO) == -1)) {
       perror("Could not dup pipes");
       exit(1);
   }
   if ((close(fd[0]) == -1) || (close(fd[1]) == -1)) {
```

```
        perror("Could not close extra descriptors");
        exit(1);
    }
          /* se crean los demás procesos con sus correspondientes entubamientos */
    for (i = 1; i < nprocs; i++) {
        if (pipe (fd) == -1) {
            fprintf (stderr, "Could not create pipe %d: %s\n",
                i, strerror(errno));
            exit (1);
        }
        if ((childpid = fork()) == -1) {
            fprintf (stderr, "Could not create child %d: %s\n",
                i, strerror(errno));
            exit (1);
        }
        if (childpid > 0)      /* para el proceso padre, se reasigna stdout */
            error = dup2 (fd[1], STDOUT_FILENO);
        else
            error = dup2 (fd[0], STDIN_FILENO);
        if (error == -1) {
            fprintf(stderr, "Could not dup pipes for iteration %d: %s\n",
                    i, strerror(errno));
            exit (1);
        }
        if ((close(fd[0]) == -1) || (close(fd[1]) == -1)) {
            fprintf(stderr, "Could not close extra descriptors %d: %s\n",
                  i, strerror(errno));
            exit (1);
        }
        if (childpid)
            break;
    }
                                        /* impresión de mensaje */
    fprintf(stderr, "This is process %d with ID %d and parent id %d\n",
        i, (int)getpid(), (int)getppid());
    exit (0);
}       /* fin del programa principal */
```

Programa 4.1

Los ejercicios siguientes prueban y modifican el programa 4.1. Para cada modificación, haga una copia del programa. Los nombres sugeridos para cada ejecutable aparecen entre paréntesis.

- Ejecute el programa 4.1 (`ring`).
- Cree un *makefile* con las descripciones necesarias para compilar y examinar con la herramienta `lint` el programa. Utilice `make` para compilar el programa. Añada varios objetivos (*targets*) para las demás partes adicionales de este proyecto. (Si está poco familiarizado con `make`, consulte la sección A.3 de este libro.)

- Haga todas las correcciones necesarias para eliminar todos los errores y mensajes de advertencia generados por lint. (Consulte la sección A.6 si está poco familiarizado con la utilería `lint`.)
- Ejecute `ring` para varios valores del argumento de la línea comando y observe lo que sucede a medida que el número de procesos en el anillo varía de 1 a 20.
- Modifique el programa original colocando una llamada `wait` al sistema antes de la proposición final `fprintf` (`ring1`). ¿Qué efecto tiene esto sobre la salida del programa?
- Modifique el programa original colocando una llamada `wait` al sistema después de la proposición final `fprintf` (`ring2`). ¿Qué efecto tiene esto sobre la salida del programa?
- Reemplace la proposición `fprintf` del programa original, `ring`, con llamadas a `sprintf` y `prtastr` (`ring3`). Escriba la función

```
void prtastr(const char *s, int fd, int n);
```

la cual imprime la cadena `s`, un carácter a la vez, en un archivo especificado por el descriptor `fd` mediante el empleo de `write`. Después de dar salida a cada carácter, `prtastr` ejecuta el siguiente ciclo:

```
for (i = 0; i < n; i++);
```

Este ciclo sólo desperdicia cierto tiempo de CPU. Utilice `prtastr` para escribir la cadena en el dispositivo de error estándar. El valor de `n` utilizado por `prtastr` se pasa a `ring3` como un argumento opcional de la línea comando. El valor por omisión para este parámetro es `1`. (Un carácter a la vez brinda a los procesos una mayor oportunidad de entrelazar su salida.) Ejecute el programa con un valor de `n` que provoque un retraso pequeño, pero que pueda notarse, entre la salida de los caracteres.

- Compare los resultados obtenidos al ejecutar el programa modificado `ring3` si
  - * Se inserta un `wait` antes de la llamada a `prtastr` (`ring4`).
  - * Se inserta un `wait` después de la llamada a `prtastr` (`ring5`).

- Modifique `ring1` de la siguiente manera:
  - * Antes del `wait`, cada proceso asigna un arreglo ID de `nprocs` elementos para guardar los ID de todos los procesos del anillo. El proceso pone su propio ID en el elemento cero del arreglo y guarda en su variable `next_ID` su ID de proceso.
  - * Haga lo siguiente para `k` desde 1 hasta `nproc - 1`:
    - *Escriba `next_ID` en la salida estándar.*
    - Lea `next_ID` de la entrada estándar.
    - Inserte `next_ID` en la posición `k` del arreglo de ID.

* Reemplace el `fprintf` después del `wait` con un ciclo que imprima el contenido del arreglo ID en el dispositivo de error estándar con un formato que facilite su lectura. Esta salida prueba la conectividad del anillo puesto que el arreglo ID contiene los procesos en el orden en que aparecen en relación con un proceso dado.

- Modifique `ring1` para que sea un anillo bidireccional (la información puede fluir en cualquier dirección entre dos vecinos del anillo). Pruebe las conexiones acumulando arreglos ID para cada dirección.
- Modifique `ring1` para crear un toro bidireccional de procesos. Acumule arreglos de ID para probar la conectividad. Un toro tiene una estructura bidimensional, similar a la de una malla con la excepción de que los extremos están conectados entre sí. Los n^2 procesos están acomodados en n anillos en cada dimensión. Cada proceso tiene cuatro conexiones (Norte, Sur, Este, Oeste).

4.2 Comunicación simple

En la sección 4.1 se explicó cómo se realizan las conexiones para un anillo de procesos. Esta sección analiza una aplicación sencilla en la que los procesos generan una secuencia de números de Fibonacci sobre el anillo. Una secuencia de números de Fibonacci se genera sumando los dos últimos números de la secuencia para producir el siguiente.

En este proyecto, los procesos pasan información en un formato de cadena de caracteres. El padre original envía la cadena `"1 1"`, la cual representa los dos primeros números de Fibonacci. Los demás procesos decodifican esta cadena, calculan el siguiente número de Fibonacci y envían una cadena que representa los dos últimos números de Fibonacci calculados para el siguiente proceso; visualizan el resultado en el dispositivo de error estándar y terminan. El padre original termina después de recibir una cadena y visualizar los números recibidos.

El siguiente desafío hace el problema más interesante: escribir el programa de modo tal que éste maneje un número muy grande de procesos y aun así calcule todos los números de Fibonacci con exactitud hasta que se presente una condición de desbordamiento (*overflow*). Si ésta se presenta, el proceso debe detectarla, mostrar un mensaje en el dispositivo de error estándar y enviar `"0 0"` al siguiente proceso. Intente calcular el mayor número de Fibonacci posible.

Para ello el lector debe comenzar con la versión `ring` del programa 4.1 y reemplazar la instrucción `fprintf` con código para leer dos enteros de la entrada estándar con el formato de cadena descrito más adelante, calcular el siguiente número entero en la secuencia de Fibonacci y escribir el resultado en la salida estándar.

- Cada cadena es la representación en ASCII de dos enteros separados por un espacio en blanco.
- El padre original escribe la cadena `"1 1"`, la cual representa dos unos, y a continuación lee una cadena.
- Los demás procesos primero leen una cadena y luego escriben otra.

- Los números de Fibonacci satisfacen la fórmula $x_{n+1} = x_n + x_{n-1}$. Cada proceso recibe dos números (por ejemplo, `a` seguido de `b`), calcula `c = a + b`, y escribe `b` seguido de `c` como una cadena con terminación *null*.
- Después de enviar la cadena a la salida estándar, el proceso escribe un mensaje de una línea en el dispositivo de error estándar con el siguiente formato

```
Process i with PID x  and parent PID y received a b and sent b c.
```

- Después de enviar el mensaje al dispositivo de error estándar, el proceso termina.

Intente escribir el programa de modo que éste maneje el mayor número posible de procesos y, aun con esto, calcule de manera correcta los números de Fibonacci. Al momento de la ejecución, puede suceder que se agote el número de procesos y que alguno de ellos genere un desbordamiento numérico cuando calcula el siguiente número. Intente detectar este desbordamiento y envíe la cadena `"0 0"` si es que éste se presenta.

Notas: El programa debe ser capaz de calcular Fib(46) = 1, 836, 311, 903 utilizando 45 procesos o Fib(47) = 2, 971, 215, 073 empleando 46 procesos. Incluso debe ser capaz de calcular Fib(78) = 8, 944, 394, 323, 791, 464 utlizando para ello 77 procesos. Con un poco de trabajo adicional el programa puede calcular valores más grandes. Un enfoque posible para detectar el desbordamiento es duplicar el resultado y comparar dicho valor con el obtenido cuando primero se duplican los valores que van a sumarse y después se hace la adición.

Este programa impone una carga muy fuerte sobre la CPU de la máquina. No intente este proyecto con más de unos cuantos procesos a menos que esté trabajando en una computadora dedicada. Por otra parte, en algunos sistemas, el límite de procesos para un usuario puede interferir con la ejecución del programa para un gran número de procesos.

4.3 Exclusión mutua con fichas (*tokens*)

Todos los procesos en el anillo comparten el dispositivo de error estándar, y la llamada a `ptrastr` descrita en la sección 4.1 es una sección crítica de estos procesos. Existe una estrategia sencilla basada en fichas (*tokens*) para permitir el acceso exclusivo a un dispositivo compartido. La ficha puede ser un carácter solo que pasa por el anillo. Cuando un proceso dado adquiere la ficha (lee el carácter de la entrada estándar), éste tiene entonces el acceso exclusivo al dispositivo compartido. Cuando dicho proceso termina de utilizar el dispositivo compartido, escribe el carácter en la salida estándar de modo que el siguiente proceso pueda adquirir la ficha. El algoritmo para la exclusión mutua es similar a un boleto para hablar (o a la concha [35]) empleado por algunas culturas para poner orden en las reuniones. Sólo la persona que tiene el boleto es la que puede hablar.

La adquisición de la exclusión mutua comienza cuando el primer proceso escribe la ficha (sólo un carácter) en su salida estándar. A partir de aquí, los procesos emplean la siguiente estrategia:

- Leen la ficha de la entrada estándar.
- Tienen acceso al dispositivo compartido.
- Escriben la ficha en la salida estándar.

Si un proceso no desea tener acceso a un dispositivo compartido, entonces deja pasar la ficha.

Al finalizar ¿qué sucede con el algoritmo anterior? Después de que un proceso ha terminado de escribir sus mensajes en el dispositivo de error estándar, debe continuar pasando la ficha hasta que también los demás procesos del anillo hayan terminado de escribir sus mensajes. Una estrategia para detectar que han terminado es reemplazar la ficha por un entero. El símbolo inicial tiene un valor cero. Si un proceso termina el acceso pero desea volver a tener acceso al dispositivo después, entonces sólo pasa la ficha sin cambio. Cuando un proceso ya no desea tener acceso al dispositivo compartido, entonces lleva a cabo el siguiente proceso de apagado (*shutdown*):

- Lee la ficha.
- Incrementa la ficha.
- Escribe la ficha.
- Repite lo siguiente hasta que la ficha tenga un valor igual al número de procesos en el anillo:
    * Lee la ficha.
    * Escribe la ficha.
- Termina.

La sección de repetición del procedimiento de apagado (*shutdown*) tiene el efecto de obligar al proceso a esperar hasta que todos hayan terminado. Esta estrategia requiere el conocimiento del número de procesos que contiene el anillo.

Implante y pruebe la exclusión mutua con fichas, de la siguiente manera:

- Comience con la versión `ring3` del programa ring de la sección 4.1.
- Implemente la exclusión mutua para el dispositivo de error estándar utilizando el método de la ficha entera descrito anteriormente pero sin el procedimiento de apagado. La sección crítica debe incluir la llamada a `prtastr`.
- Pruebe el programa con diferentes valores para los argumentos de la línea comando. ¿En qué orden aparecen los mensajes y por qué?
- Incorpore una variación en las pruebas dejando que cada proceso repita la sección crítica un número aleatorio de veces entre 0 y *r*. Pase *r* como argumento de la línea comando. Antes de cada llamada a `prtastr`, lea la ficha. Después de llamar a `prtastr`, escriba la ficha. Cuando esté terminada toda la salida, ejecute el ciclo que sólo pasa la ficha. (Sugerencia: lea la página del manual que corresponde a `drand48` y las funciones relacionadas con ella. La función `drand48` genera un valor pseudoaleatorio de tipo *double* en el intervalo [0, 1). Si `drand48` genera un valor de `x`, entonces `y = (int) (x*n)` es un entero que satisface $0 \leq y < n$.) Utilice los ID de los procesos como semillas para el generador de modo que cada proceso haga uso de números pseudoaleatorios independientes.
- Los mensajes que cada proceso escribe en el dispositivo de error estándar deben incluir ID del proceso y la hora en que comenzó la operación. Utilice la función `time` para obtener el tiempo en segundos. (Vea la página 206 en el capítulo 6 para una descripción de `time`.)

4.4 Exclusión mutua por votación

Un problema con el método de la ficha es que genera un tráfico continuo (una forma de ocupado espera) incluso cuando ningún proceso desea entrar en su propia sección crítica. Si todos los procesos desean entrar en sus secciones críticas, el acceso se permite de acuerdo con la posición relativa a medida que la ficha recorre el anillo. Existe un enfoque alternativo que hace uso del algoritmo de Chang y Roberts para hallar extremos [19]. Los procesos que desean entrar en sus secciones críticas realizan una votación para ver cuál de ellos es el que obtiene el acceso. Este método sólo genera tráfico cuando un proceso desea el acceso exclusivo y puede modificarse para acomodar una gran variedad de esquemas de prioridad para determinar cuál es el siguiente proceso que obtendrá el acceso.

Cada proceso que está compitiendo en la exclusión mutua genera un mensaje de votación con un ID único formado por dos partes. La primera de ellas recibe el nombre de número de secuencia, el cual está basado en una prioridad. La segunda parte del ID, el ID del proceso, se utiliza para deshacer empates si dos procesos tienen la misma prioridad. Ejemplos de prioridad incluyen números de secuencia basados en el tiempo o en el número de veces que el proceso ha ganado la exclusión mutua en el pasado. En cada una de estas estrategias el menor valor es el que corresponde a la prioridad más grande. Aquí se hará uso de la última estrategia.

Para votar, el proceso escribe en el anillo su mensaje de ID. Cada proceso que no está participando en la votación se encarga sólo de pasar los mensajes de ID que le llegan al siguiente proceso sobre el anillo. Cuando un proceso que está votando recibe un mensaje de ID, éste basa sus acciones en lo siguiente:

- Si el mensaje que llega tiene un ID mayor (menor prioridad) que su propio voto, el proceso desecha el mensaje.
- Si el mensaje que llega tiene un ID menor (mayor prioridad) que su propio voto, entonces el proceso distribuye el mensaje.
- Si el mensaje que llega es su propio mensaje, entonces el proceso ha ganado la exclusión mutua y puede comenzar la ejecución de su sección crítica. (Convénzase de que el ganador es el proceso cuyo mensaje de ID es el menor de todos en dicha votación.)

Un proceso renuncia a la exclusión mutua enviando alrededor del anillo un mensaje de renuncia. Una vez que el proceso detecta el inicio del proceso de votación ya sea por petición o porque ha recibido un mensaje, éste no puede iniciar otra votación hasta que detecte un mensaje de renuncia. Por tanto, de todos los procesos que han decidido participar, el que reciba el acceso la menor cantidad de veces en el pasado es el que gana la elección.

Implante el algoritmo de votación para el acceso exclusivo al dispositivo de error estándar. Incorpore valores aleatorios para el valor de retraso, que es el último argumento de la función `prtastr` definida en la sección 4.1. Diseñe una estrategia para la terminación después de que todos los procesos han completado su salida.

4.5 Elección del líder en un anillo anónimo

Las especificaciones de los algoritmos distribuidos se refieren a las entidades que ejecutan el algoritmo como *procesos* o *procesadores*. Dichos algoritmos a menudo especifican un modelo para el procesador en términos de una máquina de estados finitos. Los modelos del procesador están clasificados por la forma en que se controlan las transiciones de estado (sincronía) y si los procesadores están etiquetados.

En el *modelo de procesador síncrono* los procesadores avanzan por pasos cerrados y las transiciones están controladas por un reloj. En el *modelo de procesador asíncrono*, las transiciones de estado están controladas por mensajes. La recepción de un mensaje en un enlace de comunicación da inicio a un cambio en el estado del procesador. El procesador puede enviar mensajes a sus vecinos, realizar algún cálculo, o detenerse como resultado del mensaje recibido. En cualquier enlace dado entre procesadores, los mensajes llegan en el orden en que éstos se envían. Los mensajes experimentan en un retraso de transmisión finito pero impredecible.

Un sistema de comunicación de procesos UNIX conectados por entubamiento, como el anillo del programa 4.1, es un ejemplo de un sistema asíncrono. Una máquina SIMD (una instrucción, varios datos) como la CM-2 es un ejemplo de un sistema síncrono.

El modelo de un procesador también debe especificar si los procesadores tienen etiquetas o son indistinguibles. En un *sistema anónimo*, los procesadores no tienen ninguna característica que los distinga. En general, los algoritmos donde participan sistemas de procesadores o procesos anónimos son más complejos que los algoritmos correspondientes a sistemas donde existen etiquetas.

La rutina `fork` de UNIX crea una copia del proceso que la llama. Si el padre y el hijo fuesen completamente idénticos, el `fork` no hará nada más allá de lo que puede hacerse con un solo proceso. De hecho, UNIX distingue al padre y al hijo por sus ID de proceso, y el `fork` devuelve valores diferentes para el padre y el hijo de modo que cada uno se dé cuenta de la identidad del otro. En otras palabras, `fork` rompe la simetría entre el padre y el hijo mediante la asignación de ID de proceso diferentes. Los sistemas de procesadores UNIX no son anónimos debido a que los procesos pueden etiquetarse con sus propios ID.

El *rompimiento de simetría* es un problema general en el cómputo distribuido en el que es necesario distinguir procesos (o procesadores) idénticos a fin de lograr un trabajo útil. La asignación de un acceso exclusivo es un ejemplo de rompimiento de simetría. Una manera posible de asignar un acceso exclusivo es darle preferencia al proceso que tiene el mayor ID de proceso. A menudo, lo mejor es emplear un método más equitativo. El algoritmo de votación de la sección 4.4 asigna la exclusión mutua al proceso que lo haya adquirido el menor número de veces en el pasado. El algoritmo emplea el ID del proceso únicamente en caso de empates.

La elección del líder es otro ejemplo de un algoritmo para romper la simetría. Los algoritmos de elección de líder se emplean en algunas redes para designar un procesador particular para dividir la red, regenerar símbolos, o realizar otras operaciones.

Por ejemplo, ¿qué sucede en una red de anillo de fichas si el procesador que retiene la ficha deja de trabajar? Cuando el procesador que dejó de trabajar y reanuda su trabajo, ya no contiene la ficha y la actividad de la red queda suspendida. Uno de los procesadores que no ha fallado debe tomar la iniciativa para generar otra ficha. ¿Cómo decidir qué procesador es el que tiene esta responsabilidad?

No existen algoritmos determinísticos para elegir al líder en un anillo anónimo. En esta sección se estudia la implantación de un algoritmo probabilista para la elección del líder en un anillo anónimo. El algoritmo es una versión asíncrona del algorítmo síncrono propuesto en [42].

Itai y Roteh [42] propusieron un algoritmo para la elección del líder en un anillo síncrono anónimo de tamaño n. La versión síncrona del algoritmo avanza en fases.

- Fase cero:
  - Se hace la variable local `m` igual con n.
  - Se asigna a `active` el valor `TRUE`.
- Fase k:
  - Si `active` es `TRUE`:
    - Se elige un número aleatorio, `x`, entre `1` y `m`.
    - Si el número elegido es 1, se envía un mensaje de un bit alrededor del anillo.
  - Se cuenta el número de mensajes de un bit recibidos en los próximos $n - 1$ pulsos de reloj:
    - Si únicamente un procesador activo fue el que eligió 1, entonces la elección está completa.
    - Si ningún procesador activo eligió el 1, se va a la siguiente fase sin ningún cambio.
    - Si p procesos escogieron el 1, se hace `m` igual con p. Si el proceso está activo y no escogió el 1, se asigna a su variable local `active` el valor `FALSE`.

En resumen, en cada fase los procesos activos seleccionan un número aleatorio entre 1 y el número de procesos activos. Cualquier proceso que tome un 1 se encontrará activo en la siguiente ronda. Si ninguno escoge un 1 en una ronda dada, el proceso activo hace otro intento. La probabilidad de que un proceso en particular tome un 1 aumenta a medida que disminuye el número de procesos activos. En promedio, el algoritmo elimina con rapidez los procesos que participan en la contienda. Itai y Roteh demostraron que el número esperado de fases necesarias para seleccionar al líder en un anillo de tamaño n es menor que $e \approx 2.718$ e independiente de n.

Implemente una simulación de este algoritmo para la elección del líder a fin de medir la distribución de probabilidad $J(n, k)$, que es la probabilidad de que la elección en un anillo de tamaño n se lleve a cabo en k.

La implementación tiene que abordar dos problemas. El primero de ellos es que el algoritmo está dado para un anillo síncrono, pero la implantación se hará en un anillo asíncrono.

Los anillos síncronos son temporizados por los mensajes que reciben (esto es, cada vez que un proceso lee un mensaje, éste actualiza su reloj). Los procesos deben leer los mensajes en el punto correcto del algoritmo, o perderán la sincronización. Con todo esto, los procesos inactivos deberán escribir mensajes de reloj.

La segunda dificultad aparece debido a que la convergencia teórica del algoritmo depende de que los procesos tengan flujos independientes de números aleatorios. En la práctica los procesos utilizan un generador de números pseudoaleatorio con una semilla apropiada. Se supone que los procesos son idénticos, pero si ellos comienzan con la misma semilla, el algoritmo no trabajará. La implantación puede hacer trampa utilizando los ID de los procesos para generar una semilla, pero a fin de cuentas ésta debe incluir un método para generar los números con base en el reloj del sistema u otro *hardware* del sistema. (Las primeras secciones del capítulo 6 discuten las llamadas al sistema para tener acceso al reloj y los temporizadores de éste.)

4.6 Anillo de fichas (*toke ring*) para comunicación

En esta sección se desarrolla una simulación de la comunicación sobre una red de anillo de fichas (*toke ring*). Ahora cada proceso sobre el anillo representa un IMP (procesador de interfaz de mensajes) de un nodo sobre la red. El IMP maneja el paso de mensajes y el control de la red para el huésped del nodo. Cada proceso IMP crea un par de entubamientos para comunicarse con sus procesos huéspedes, como se muestra en la figura 4.9. El huésped está representado por un proceso hijo creado a partir de un IMP.

Cada IMP espera mensajes provenientes de su huésped y de la red. Para que sea más claro, un mensaje está formado por cinco números enteros —un tipo de mensaje, el ID de la IMP fuente, el ID de la IMP destino, un estado y un número de mensaje. Los posibles tipos de mensajes están definidos por el tipo enumeración `msg_t`:

```
typedef enum msg_const {TOKEN, HOST2HOST,
          IMP2HOST, HOST2IMP, IMP2IMP} msg_t;
```

El IMP debe leer un mensaje `TOKEN` del anillo antes de que escriba cualquier mensaje que éste origine en el anillo. Cuando el IMP recibe un acuse de recibo del mensaje que envió, entonces escribe un nuevo mensaje `TOKEN` en el anillo. Los acuses de recibo están indicados en el miembro de estado que es de tipo `msg_status_t` y que está definido por

```
typedef enum msg_const_source{NONE, NEW, ACK} msg_status_t;
```

El IMP espera un mensaje ya sea del huésped o del anillo. Cuando un IMP detecta que el huésped desea enviar un mensaje, lo lee, lo coloca en un *buffer* temporal y activa la bandera `got_msg`. Una vez activada la bandera `got_msg`, el IMP no puede leer ningún mensaje adicional del huésped hasta que `got_msg` sea desactivada.

Cuando el IMP detecta un mensaje proveniente de la red, sus acciones dependen del tipo de mensaje. Si el IMP lee un mensaje `TOKEN` y tiene en espera un mensaje del huésped (`got_msg`

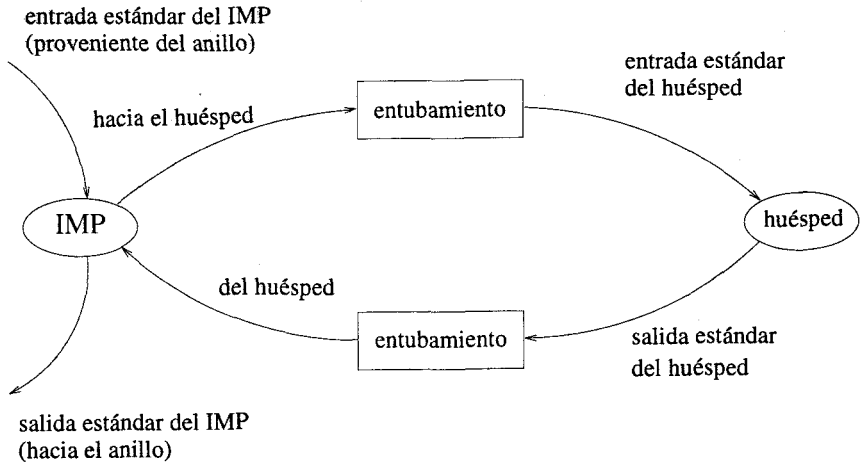


Figura 4.9: Estructura IMP-huésped.

está activa), el IMP escribe el mensaje del huésped en la red. Si el IMP no tiene ningún mensaje que enviar (`got_msg` está inactiva), entonces escribe `TOKEN` en la red.

Si el IMP lee un mensaje diferente del mensaje `TOKEN` del anillo, sus acciones dependen de los ID de la fuente y el destino en el mensaje.

- Si el ID de la fuente del mensaje concuerda con el ID del IMP, entonces el mensaje fue enviado por el mismo IMP. En este caso, el IMP imprime un mensaje en el dispositivo de error estándar indicando si el mensaje fue recibido por el destinatario. En cualquier caso el IMP escribe un mensaje `TOKEN` en el anillo y borra `got_msg`.
- Si el ID de destino del mensaje concuerda con el ID del IMP, el mensaje es para el IMP o para el huésped del IMP. El IMP imprime un mensaje de estado en el dispositivo de error estándar que indica el tipo de mensaje. El IMP cambia el estado del mensaje a `ACK` y escribe el mensaje en el anillo.
- En cualquier otro caso, el IMP escribe el mensaje sin cambio alguno en el anillo.

El protocolo de anillo de fichas (*token ring*) IEEE 802.5 usado en la actualidad es más complicado que esto. En lugar de mensajes de longitud fija, los IMP utilizan un temporizador de retención de la ficha para fijar un valor por omisión cuando comienza la transmisión. Un IMP puede transmitir hasta que el temporizador termine, de modo que los mensajes pueden llegar a ser bastante grandes. Los mensajes también pueden tener un esquema de prioridad [84]. En el protocolo de anillo de fichas (*token ring*) usado actualmente, uno de los IMP es designado como el monitor activo del anillo y éste envía periódicamente marcos de control a las demás estaciones para indicarles que el monitor activo está presente. El monitor activo también detecta la pérdida de la ficha y tiene la responsabilidad de regenerarlas. Todas las demás estaciones envían de manera periódica marcos de control `standby-monitor-present` para detectar interrupciones en el anillo.

Para la primera parte de este proyecto, inicie con el programa 4.1. Modifíquelo de modo que después de la creación del anillo, cada proceso IMP cree dos entubamientos y un proceso huésped hijo como se muestra en la figura 4.9. Redirija la salida y la entrada estándar del huésped hijo como se indica en dicha figura. Escriba y ponga a prueba un programa aparte, `hostgen`, que tenga dos argumentos de línea comando: un número `n` de proceso y un tiempo `s` de suspensión. El programa `hostgen` debe ejecutar el ciclo siguiente:

- Suspender su ejecución un número aleatorio de segundos entre 0 y *s*.
- Escribir un entero aleatorio entre 0 y *n* en el dispositivo de salida estándar.

Para la segunda parte de este proyecto, inicie de nuevo con el programa 4.1. Modifíquelo de modo que el huésped hijo ejecute, con una llamada a `exec`, el programa `hostgen` con los argumentos de línea comando apropiados. El IMP entra en un ciclo infinito para vigilar todas sus posibles entradas utilizando a `select`. Cuando la entrada está disponible, el IMP lleva a cabo el protocolo simple de anillo de fichas o mensajes descrito con anterioridad.

4.7 Preprocesador con estructura de entubamiento

El preprocesador de C, `cpp`, hace un procesamiento previo del código fuente en C de modo que el compilador de C no tenga que preocuparse de ciertas cosas. Por ejemplo, si un programa en C tiene una línea como

```
#define BUFSIZE 250
```

`cpp` reemplaza todas las instancias del símbolo `BUFSIZE` por `250`. El preprocesador de C trata con símbolos, de modo que no reemplazará una ocurrencia de `BUFSIZE1` con `2501`. Es claro que se requiere este comportamiento para el código fuente en C. Es posible hacer que `cpp` entre en un ciclo con algo como lo siguiente

```
#define BUFSIZE BUFSIZE + 1
```

Distintas versiones de `cpp` manejan esta situación de manera diferente.

En otras situaciones el programa no es capaz de tratar con los símbolos o las fichas y puede reemplazar cualquier ocurrencia de una cadena, incluso si ésta es parte de un símbolo o está formada por varios de ellos. Un método para el manejo de los ciclos que puede generarse por recursión es no llevar a cabo ninguna prueba adicional sobre una cadena que ya ha sido reemplazada. Este método falla en situaciones tan sencillas como

```
#define BUFSIZE 250
#define BIGGERBUFSIZE BUFSIZE + 1
```

Una manera ineficiente de manejar la situación anterior es efectuar varias pasadas a través del archivo de entrada, una para cada `#define` y hacer los reemplazos de manera secuencial. El procesamiento puede hacerse con más eficiencia (y posiblemente en paralelo) con una estructura de entubamiento (*pipeline*). La figura 4.10 muestra una estructura de entubamiento

de cuatro etapas. Cada etapa aplica una transformación a su entrada y entonces da salida al resultado, que es la entrada de la siguiente etapa. Una estructura de entubamiento se parece a la línea de ensamblado de un proceso de manufactura.

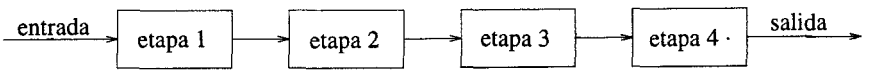


Figura 4.10: Estructura de entubamiento de cuatro etapas.

En esta sección se desarrolla una estructura de entubamiento de preprocesadores basada en el anillo del programa 4.1. Para simplificar la programación, los preprocesadores sólo convertirán caracteres simples en cadenas de caracteres.

- Escriba una función `pre_process_char` que tenga el siguiente prototipo:

```
int pre_process_char(int fdin, int fdout, char inchar,
                     char *outstr);
```

  La función `pre_process_char` lee del descriptor de archivo `fdin` hasta que encuentra una marca de fin de archivo y escribe en el descriptor de archivo `fdout`, traduciendo cualquier ocurrencia del carácter `inchar` en la cadena `outstr`. La función devuelve 0 si tiene éxito o de lo contrario regresa `-1`. Escriba una rutina de control (*driver*) para probar este programa antes de usarlo con el anillo.
- Modifique el programa 4.1 de modo que éste tome ahora cuatro argumentos de línea comando (`ringpp`). Ejecute el programa de la manera siguiente:

```
ringpp n conf.in file.in file.out
```

  El valor del argumento de línea comando `n` especifica el número de etapas en el sistema de entubamiento. Esto corresponde a `nprocs - 2` en el programa 4.1. El padre original es responsable de generar la entrada al sistema de entubamiento mediante la lectura de `file.in` y el último hijo es el responsable de eliminar la salida del sistema de entubamiento y escribirla en `file.out`. Antes de que `ringpp` cree el anillo, el padre original abre el archivo `conf.in`, lee el contenido de éste en `n` líneas que contienen un carácter y una cadena, y después guarda esta información en un arreglo.

  Suponga que `n` siempre es menor o igual que `MAXPROCS` y que cada traducción de cadena contiene cuando mucho `MAXSTR` caracteres, incluyendo el de terminación de la cadena. El programa `ringpp` lee el archivo `conf.in` antes de dar origen a cualquier hijo, de modo que toda la información del arreglo esté disponible para todos los hijos.
- El padre original es responsable de copiar el contenido del archivo `file.in` en su salida estándar. Cuando el padre encuentra el final de `file.in`, el proceso termina. El padre original genera la entrada de la estructura de entubamiento y no lleva a cabo ningún procesamiento interconectado.

- El último hijo es responsable de eliminar la salida de la estructura de entubamiento. El proceso copia los datos de su entrada estándar en `file.out`, pero no lleva a cabo ningún procesamiento de entubamiento. El proceso termina cuando encuentra el fin de archivo en su entrada estándar.
- Para `i` entre `2` y `n + 1`, el proceso hijo `i` utiliza información de la línea `(i-1)` del arreglo de traducción para traducir un carácter en una cadena. Cada proceso hijo actúa como un filtro, leyendo la entrada del dispositivo de entrada estándar, haciendo la sustitución y escribiendo el resultado en el dispositivo de salida estándar. Llame a la función `pre_process_char` para procesar la entrada. Cuando ésta reciba un fin de archivo, cada proceso cierra su archivo de salida estándar y termina su ejecución.
- Después de asegurarse de que el programa trabaja de manera correcta, haga el intento con un archivo grande (varios megaoctetos) y un número moderado de procesos (10 a 20).
- Si es posible, intente ejecutar el programa en una máquina con varios procesadores para medir el aumento en la velocidad de éste. (Véase la sección 4.8 para una definición de aumento de velocidad.)

Cada etapa de la estructura de entubamiento lee de su entrada estándar y escribe en su salida estándar. El problema puede generalizarse al dejar que cada etapa ejecute con `execvp` un proceso arbitrario en lugar de llamar a la misma función. El archivo `conf.in` puede contener las líneas comando para `execvp` en lugar de la tabla de reemplazos de cadenas específica para este problema.

También es posible dejar que el padre original maneje tanto la generación de la entrada a la estructura de entubamiento como la eliminación de su salida. En este caso el padre abre `file.in` y `file.out` después de crear a su hijo. Ahora el proceso debe manejar una entrada que proviene de dos fuentes: `file.in` y su entrada estándar. Es posible hacer uso de `select` para manejar esta situación, pero el problema es más complicado de lo que parece ser. El proceso también debe vigilar su salida estándar con `select` debido a que uno de los entubamientos puede llenarse y bloquear con ello operaciones de escritura adicionales. Si el proceso se bloquea mientras escribe en su salida estándar, entonces no será capaz de eliminar la salida de la etapa final de la estructura de entubamiento. En este caso, la estructura (*pipeline*) puede quedarse bloqueada. El padre original es un candidato perfecto para generar secuencias de ejecución múltiples, tal como se estudia en los capítulos 9 y 10.

4.8 Algoritmos de anillo paralelos

El procesamiento en paralelo se refiere a la división de un problema en partes de modo que éste pueda resolverse en paralelo, reduciendo de este modo el tiempo total de ejecución. Una medida de la efectividad de la división es el aumento en la velocidad, $S(n)$, el cual se define como

$$S(n) = \frac{\text{tiempo de ejecución con un procesador}}{\text{tiempo de ejecución con } n \text{ procesadores}}$$

De manera ideal, la ejecución es inversamente proporcional al número de procesadores, con lo que el aumento en la velocidad $S(n)$ es sólo n. Desafortunadamente, en la práctica es muy raro obtener, por varias razones, un aumento lineal de la velocidad. Siempre existe una parte del

trabajo que no puede hacerse en paralelo, y la versión paralela del algoritmo incurre en cierto costo adicional cuando los procesadores se sincronizan o comunican para intercambiar información.

Los tipos de problemas más adecuados para su paralelización son los que tienen una estructura regular e involucran un intercambio de información que sigue patrones bien definidos. En esta sección se examinan dos algoritmos paralelos para el anillo: el filtrado de imágenes y la multiplicación matricial. El filtrado de imágenes pertenece a una clase de problemas en los que cada procesador lleva a cabo sus cálculos de manera independiente o mediante el intercambio de información con dos de sus vecinos. En la multiplicación matricial, uno de los procesadores debe obtener información de todos los demás procesadores con objeto de completar el cálculo. Sin embargo, la información puede propagarse por un simple desplazamiento. Existen otros algoritmos paralelos que pueden adaptarse para tener una ejecución eficiente en el anillo, pero los patrones de comunicación son más complicados que los de los ejemplos que aparecen en esta sección.

4.8.1 Filtrado de imágenes

Un filtro es una transformación que se aplica a una imagen. El filtrado puede eliminar el ruido, mejorar los detalles de la imagen o modificar las características de la imagen dependiendo del tipo de transformación. En este análisis se considera una imagen digital de escala de grises representada por un arreglo de $n \times n$ octetos. Los *filtros espaciales* comunes reemplazan el valor de cada pixel de una imagen de este tipo por una función del pixel original y sus vecinos. El algoritmo de filtrado especifica la vecindad que contribuye al cálculo mediante una máscara. La figura 4.11 muestra una máscara de 3×3 de vecinos próximos. Esta máscara en particular representa un *filtro lineal* debido a que la función es un promedio ponderado de los pixeles que forman la máscara. (En contraste, no es posible escribir un filtro no lineal como una combinación lineal de los pixeles bajo la máscara. Tomar la mediana de los pixeles vecinos es un ejemplo de un filtro no lineal.)

$$\frac{1}{9} \times \begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

Figura 4.11: Máscara para aplicar a una imagen un filtro de suavizamiento.

Los valores de la máscara son los factores de ponderación que se aplican a cada pixel en el promedio cuando la máscara queda centrada sobre el pixel donde se lleva a cabo la transformación. En la figura 4.11 todos los factores de ponderación tienen el valor 1/9. Si $a_{i,j}$ es el pixel que se encuentra en la posición (i, j) de la imagen original y $b_{i,j}$ es el pixel correspondiente en la imagen filtrada, entonces la máscara de la figura 4.11 representa la transformación del pixel.

$$b_{i,j} = \frac{1}{9}[a_{i-1,j-1} + a_{i,j-1} + a_{i+1,j-1} + a_{i-1,j} + a_{i,j} + a_{i+1,j} + a_{i-1,j+1} + a_{i,j+1} + a_{i+1,j+1}]$$

Esta transformación oculta las aristas y elimina contrastes en la imagen. En la terminología del filtrado la máscara representa un filtro de baja frecuencia debido a que mantiene los componentes que cambian con lentitud o los de baja frecuencia, y elimina los componentes de alta frecuencia. La máscara de la figura 4.12 es un filtro de alta frecuencia que hace más notables las aristas y oscurece el fondo.

$$\frac{1}{9} \times \begin{array}{|c|c|c|} \hline -1 & -1 & -1 \\ \hline -1 & 8 & -1 \\ \hline -1 & -1 & -1 \\ \hline \end{array}$$

Figura 4.12: Máscara para aplicar a una imagen un filtro de diferencias.

El anillo de procesos es una arquitectura natural para paralelizar los tipos de filtros descritos por máscaras como las de las figuras 4.11 y 4.12. Suponga que se tiene un anillo de n procesos para filtrar una imagen de $n \times n$. Cada proceso puede ser responsable de calcular el filtro para un renglón o columna de la imagen. Puesto que C guarda los arreglos en un formato de renglón (esto es, los elementos de un arreglo bidimensional se guardan linealmente en la memoria guardando primero todos los elementos del renglón cero, seguido de todos los elementos del renglón uno, y así sucesivamente), es más conveniente dejar que los procesos manejen cada uno un renglón.

Para llevar a cabo la operación de filtrado en el proceso p, se hace lo siguiente:

- Se obtienen los renglones $p - 1$, p y $p + 1$ de la imagen original, representando los valores de pixel de los tres renglones de la imagen original por un arreglo

  ```
  unsigned char a[n+2][3];
  ```

  Ponga los pixeles de la imagen del renglón $p - 1$ en `a[1][0]...a[n][0]`. Haga `a[0][0]` y `a[n+1][0]` igual a 0 con objeto de calcular el resultado para los pixeles de las orillas sin necesidad de preocuparse de los límites del arreglo. Maneje los renglones p y $p + 1$ de manera similar. Si p es 1, haga `a[0][0]...a[n+1][0]` igual a 0, lo que corresponde al renglón anterior de la imagen. Si p es n, haga `a[0][2]...a[n+1][2]` igual a 0, lo que corresponde al renglón de pixeles debajo de la parte inferior de la imagen.
- Calcule los nuevos valores para los pixeles del renglón p y guárdelos en un arreglo:

  ```
  unsigned char b[n+2];
  ```

Para calcular el valor de `b[i]` utilice la fórmula

```
int sum;
int i;
int j;
int m;

sum = 0;
for (j = 0; j < 3; j++)
    for (m = i - 1; m < i + 2; m++)
        sum += a[m][j];
b[i] = (unsigned char) (sum/9);
```

El valor de `b[i]` es el valor del pixel $b_{p,i}$ en la nueva imagen.

- Inserte `b` en el renglón p de la nueva imagen.

La descripción anterior resulta vaga en cuanto al origen de la imagen y hacia dónde va. Esta parte de E/S es el corazón del problema. El enfoque más simple es dejar que cada proceso lea de un archivo la parte de la imagen de entrada que necesita y que escriba el renglón resultante en otro archivo. En este enfoque los procesos son independientes unos de otros. Suponga que la imagen original se guarda como un archivo binario de octetos en formato de renglón. Utilice la llamada `lseek` al sistema para colocar el desplazamiento dentro del archivo en el sitio apropiado, y la llamada `read` para leer los tres renglones necesarios para hacer los cálculos. Después de calcular la nueva imagen, haga uso de `lseek` y `write` para escribir el renglón en el lugar apropiado en la imagen. Asegúrese de abrir los archivos de entrada y salida de la imagen después de la llamada `fork` de modo que cada proceso sobre el anillo tenga sus desplazamientos de archivo propios.

Anillo bidireccional

Un enfoque alternativo es la comunicación con el vecino más cercano. El proceso p sobre el anillo únicamente lee el renglón p-ésimo. Luego escribe este renglón p en sus vecinos de cualquier lado y lee los renglones $p - 1$ y $p + 1$ de sus vecinos. Este intercambio de información requiere que el anillo sea bidireccional, esto es, que el proceso de un nodo pueda leer o escribir en las ligas de cada dirección. (Como alternativa, cada liga del anillo puede reemplazarse con dos ligas unidireccionales, una en cada dirección.) Es probable que este enfoque resulte excesivo para la implantación del filtro lineal, pero varios problemas relacionados con éste requieren la comunicación con el vecino más cercano. Por ejemplo, el método explícito para resolver la ecuación de calor sobre una rejilla de $n \times n$ utiliza una actualización del vecino más cercano de la forma

$$b_{i,j} = a_{i,j} + D\left[a_{i-1,j} + a_{i+1,j} + a_{i,j-1} + a_{i,j+1}\right]$$

La constante D está relacionada con la rapidez con la que el calor difunde sobre la rejilla. El arreglo $b_{i,j}$ es la nueva distribución de calor sobre la rejilla después de haber transcurrido una unidad de tiempo. Esta distribución se convierte en el arreglo inicial $a_{i,j}$ del siguiente intervalo de tiempo. Es claro que el programa no debe escribir la rejilla en el disco entre cada intervalo de tiempo, de aquí que sea necesario el intercambio con el vecino más cercano.

Cálculo en bloque

Otro aspecto importante en el procesamiento en paralelo es la granularidad del problema y la forma en que ésta se transforma en el número de procesadores. El anillo típicamente tiene menos de 100 procesadores, pero las imágenes de interés pueden tener 1024 × 1024 pixeles. En este caso cada procesador calcula el filtro para un bloque de renglones.

Suponga que el anillo tiene m procesos y la imagen $n \times n$ pixeles, donde $n = qm + r$. Los primeros r procesos son responsables de $q + 1$ renglones, mientras que los procesos restantes se encargan de q renglones. Cada procesador calcula el rango de renglones del que es responsable para q y r. Los parámetros m y n se pasan al proceso original como argumentos de línea comando.

4.8.2 Multiplicación de matrices

Otro problema que conduce a la ejecución en paralelo sobre un anillo es la multiplicación matricial. El resultado de multiplicar dos matrices de $n \times n$, A y B, es otra matriz C cuya entrada (i,j) está dada por

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$

En otras palabras, el elemento (i, j) del resultado es el producto del renglón i de la primera matriz con la columna j de la segunda matriz. Comience suponiendo que existen n procesadores en el anillo. Cada arreglo de entrada está almacenado como un archivo binario con formato de renglón. Los elementos del arreglo son de tipo `int`.

Un enfoque para la multiplicación matricial es que el proceso p lea el renglón p del archivo de entrada A y la columna p del archivo de entrada B. El procesador p acumula el renglón p de la matriz C. Éste multiplica el renglón p de A por la columna p de B y pone en $c_{p,p}$ el valor resultante. A continuación escribe la columna p de la matriz B en el anillo y lee la columna $p - 1$ de su vecino. El proceso p entonces calcula el elemento $c_{p,p-1}$, y así sucesivamente.

El enfoque renglón-columna es muy eficiente una vez que los procesadores han leído las columnas de B, pero ya que B está guardada en forma de renglón, los accesos al archivo son ineficientes si el proceso accesa una columna de B, puesto que debe leer y buscar cada elemento. Además es probable que la multiplicación matricial sea un paso intermedio de un cálculo más grande que puede tener distribuidos a los arreglos A y B distribuidos en los procesos con un formato de renglón. El siguiente algoritmo lleva a cabo la multiplicación matricial cuando el proceso p comienza con el renglón o de la matriz A y el renglón p de la matriz B.

El proceso p se encarga de calcular el renglón p del resultado. En cada iteración, un renglón de B contribuye con un término a la suma necesaria para calcular cada elemento del renglón p de la matriz producto. Eventualmente, cada proceso necesitará todas las entradas de la matriz B y recibirá los renglones de B, uno a la vez, de sus vecinos.

Utilice los arreglos siguientes:

```
int a[n+1];       /* retiene el renglón pth de A */
int b[n+1];       /* inicia con el renglón pth de B */
int c[n+1];       /* guarda el renglón de pth C */
```

Inicia los elementos de `a[]` y `b[]` a partir de sus respectivos archivos. Inicia `c[]` empleando

```
for (k = 1, k < n+1; k++)
   c[k] = a[p] * b[k];
```

En el proceso p lo anterior representa la contribución del renglón p de la matriz B al renglón p de la matriz C. En notación matricial, $c_{p,k} = a_{p,p}b_{p,k}$. El proceso p hace lo siguiente:

```
m = p;
write(STDOUT_FILENO, &b[1], n*sizeof(int));
read(STDIN_FILENO, &b[1], n*sizeof(int));
for (k = 1; k < n+1; k++) {
   if ((m -= 1) == 0)
      m = n;
   c[k] += a[m]*b[k];
}
```

El `read` llena el arreglo `b[]` con los valores del renglón B retenidos inicialmente por el procesador que está inmediatamente antes del proceso p en el anillo. Una ejecución del ciclo `for` sumará la contribución del renglón $p-1$ de B al renglón p del resultado correspondiente a $c_{p,k} = c_{p,k} + a_{p,p-1}b_{p-1,k}$. Ejecute este código $n-1$ veces para multiplicar todo el arreglo. Escriba el arreglo resultante c como el renglón p del archivo de salida que representa a C.

4.9 Anillo flexible

El anillo flexible es un anillo en el que pueden añadirse y eliminarse nodos. La flexibilidad es útil para la recuperación de fallas y el mantenimiento de la red.

- Modifique la versión `ring` del programa 4.1 para hacer uso de entubamientos con nombre o FIFOs en lugar de los entubamientos sin nombre. Diseñe un esquema apropiado para asignar nombres a los entubamientos.
- Proponga e implante un esquema para añadir un nodo después del nodo i del anillo.
- Proponga e implante un esquema para eliminar el nodo i del anillo. Pase i como argumento de línea comando.

Después de probar las estrategias para insertar y eliminar nodos, convierta la implantación de anillo de paquetes de la sección 4.6 en una que utilice entubamiento con nombre. Desarrolle un protocolo de modo que cualquier nodo pueda iniciar una solicitud para añadir o eliminar un nodo. Implemente el protocolo.

Este proyecto deja abierta la mayor parte de la especificación. Imagine lo que significa insertar o borrar un nodo.

4.10 Lecturas adicionales

Las primeras versiones del proyecto de anillo descrito en este capítulo pueden encontrarse en [72]. *Local and Metropolitan Area Networks*, cuarta edición, de Stallings [84] presenta un buen análisis de los estándares de red *token ring*, *token bus* y FDDI. Cada una de estas redes está basada en la arquitectura de anillo. Stallings también presenta los métodos de elección utilizados por estas arquitecturas para la regeneración de fichas y la reconfiguración. El artículo "A resilient mutual exclusion algorithm for computer networks" por Nishio *et al.* [51], analiza el problema de regenerar fichas extraviadas en redes de computadoras.

Los textos teóricos sobre algoritmos distribuidos son abundantes. Los algoritmos de la sección 4.4 están basados en un artículo de Chang y Roberts [19], mientras que los algoritmos de la sección 4.5 aparecen en Itai y Roteh [42]. Un buen artículo teórico sobre anillos anónimos es "Computing on an anonymous ring" de Attiya *et al.* [5]. El libro *Introduction to Parallel Computing: Design and Analysis of Algorithms*, por Kumar *et al.* [45], presenta un buen panorama de algoritmos paralelos y cómo traducirlos de acuerdo con la arquitectura de la máquina en particular.

Parte II

Eventos asíncronos

Capítulo 5

Señales

Pocas personas aprecian la naturaleza engañosa de los eventos asíncronos hasta que encuentran un problema irreproducible. Este capítulo estudia las señales y los efectos de éstas sobre los procesos. El capítulo también abarca el nuevo manejo de señales de POSIX.1b, así como los medios para llevar a cabo operaciones de E/S en forma asíncrona. El nuevo manejo de señales es particularmente útil, ya que permite la formación de colas de señales y la transmisión de información adicional para el manejador de la señal.

Una *señal* es una notificación por *software* para un proceso de la ocurrencia de cierto evento. Una señal es generada cuando ocurre el evento que causa la señal. Se dice que la señal es *depositada* cuando el proceso emprende una acción con base en ella. El *tiempo de vida* de una señal es el intervalo entre la generación y el depósito de ésta. Se dice que una señal está *pendiente* si es generada pero todavía no es depositada. Puede existir un tiempo considerable entre la generación de la señal y el depósito de ésta. Por otra parte, el proceso debe estar ejecutándose sobre un procesador en el momento en que se emprende la acción.

Un proceso *atrapa* una señal si éste ejecuta el *manejador de señal* cuando se deposita la señal. Un programa instala el *manejador de señal* mediante una llamada a `sigaction` con el nombre de la función escrita por el usuario o `SIG_DFL` o `SIG_IGN`. `SIG_DFL` significa que debe realizarse una acción preestablecida, mientras que `SIG_IGN` indica que la señal debe ignorarse. Si el proceso está configurado para *ignorar* una señal, ésta es descartada al momento de depositarla y no tiene ningún efecto sobre el proceso.

La acción que se emprende cuando se genera la señal depende del manejador en uso para dicha señal y de la *máscara de señal* del proceso. La máscara de señal contiene una lista de las señales que en determinado momento están *bloqueadas*. Es fácil confundir el bloqueo de una señal con el hecho de ignorarla. Las señales bloqueadas no son desechadas como sucede con las señales ignoradas. Si una señal pendiente está bloqueada, será depositada cuando el proceso la libere. Los programas bloquean una señal cambiando con `sigprocmask` la máscara de

señal del proceso. Los programas ignoran una señal especificando como manejador de señal a `SIG_IGN` con una llamada a `sigaction`.

Este capítulo presenta todos los aspectos de las señales en POSIX. La sección 5.1 analiza las señales y presenta ejemplos de cómo generarlas. La sección 5.2 estudia la máscara de señal y el bloqueo de señales, mientras que la sección 5.3 contempla el proceso de atrapar e ignorar señales. La sección 5.4 muestra la manera en que un proceso debe hacer el bloqueo mientras espera el depósito de una señal, y la sección 5.5 ilustra sobre el manejo básico de señales a través de un programa `biff` sencillo. Las demás secciones del capítulo presentan aspectos más avanzados sobre el manejo de señales. La sección 5.6 discute las interacciones entre las llamadas al sistema y el manejo de señales. La sección 5.7 cubre la llamada `siglongjmp`, mientras que la sección 5.8 presenta la extensión POSIX.1b para señales de tiempo real. Finalmente, la sección 5.9 introduce la E/S asíncrona de POSIX.1b.

5.1 Envío de señales

Cualquier señal tiene un nombre simbólico que comienza con `SIG`. Los nombres de las señales están definidos en el archivo `signal.h`, el cual debe incluir cualquier programa en C que haga uso de las señales. Los nombres de éstas representan enteros pequeños mayores que 0. La tabla 5.1 contiene una lista de las señales requeridas por POSIX.1. La acción por omisión asociada con todas las señales requeridas es terminar el proceso de manera anormal.

Símbolo	Significado
`SIGABRT`	terminación anormal como la iniciada por `abort`
`SIGALRM`	señal de espera como la iniciada por `alarm`
`SIGFPE`	error en operación aritmética como en la división por cero
`SIGHUP`	colgado (muerte) de la terminal de control (proceso)
`SIGILL`	instrucción de *hardware* no válida
`SIGINT`	señal de atención interactiva
`SIGKILL`	terminación (no se puede atrapar o ignorar)
`SIGPIPE`	escritura en un entubamiento sin lectores
`SIGQUIT`	terminación interactiva
`SIGSEGV`	referencia no válida a memoria
`SIGTERM`	terminación
`SIGUSR1`	señal 1 definida por el usuario
`SIGUSR2`	señal 2 definida por el usuario

Tabla 5.1: Señales requeridas por POSIX.1.

POSIX.1 también define un grupo opcional de señales para control de trabajos. Los intérpretes utilizan estas señales para controlar la interacción entre los procesos de primero y se-

gundo planos. Si una implantación soporta el control de trabajos (_POSIX_JOB_CONTROL está definido), entonces también debe dar soporte a las señales mostradas en la tabla 5.2. La acción predeterminada para SIGCHLD es ignorar. Cuando SIGCONT se genera, siempre continúa de inmediato un proceso suspendido (incluso si SIGCONT está bloqueada o ha sido ignorada). La acción por omisión para las demás señales de control es detener el proceso.

Señales de control de trabajo de POSIX.1

Símbolo	Significado
SIGCHLD	indica terminación o suspensión del proceso hijo
SIGCONT	continuar si está detenido (efectuado cuando se genera)
SIGSTOP	señal de alto (no se puede atrapar o ignorar)
SIGTSTP	señal de alto interactiva
SIGTTIN	proceso de segundo plano que intenta leer la terminal de control
SIGTTOU	proceso de segundo plano que intenta escribir en la terminal de control

Tabla 5.2: Señales de control de trabajos para procesos de primero y segundo planos.

Algunas señales como SIGFPE o SIGSEGV se generan cuando ocurren ciertos errores. Otras señales se generan mediante llamadas específicas. Un usuario puede enviar una señal sólo a los procesos que posee. En otras palabras, es necesario que los ID del usuario real o efectivo de los procesos que envían y reciben señales sean los mismos.

Las señales se generan desde el intérprete con el comando `kill`. Si bien `kill` puede parecer un nombre extraño (asesinar) para un comando que envía señales, no es tanto, ya que muchas de las señales tienen como acción por omisión la terminación de procesos.

SINOPSIS

```
kill -s signal pid...
kill -l [exit_status]
kill [-signal] pid...
```

POSIX.2, Spec 1170

El comando `kill` tradicional aparece en la sinopsis anterior. Esta forma toma dos argumentos línea comando, un número de señal precedido por un signo más o menos y un ID de proceso. La forma listada es obsoleta en los estándares POSIX.2 y Spec 1170 debido a que no sigue los lineamientos para argumentos presentados en la sección A.1.2. En estos comandos utilice un nombre simbólico para el parámetro `signal` omitiendo el prefijo SIG del correspondiente nombre de la señal en POSIX. Existen dos señales, SIGUSR1 y SIGUSR2, disponibles para los usuarios y que no tienen un uso preasignado.

Ejemplo 5.1

El siguiente comando envía la señal `SIGUSR1` *al proceso* 3423.

```
kill -USR1 3423
```

Ejemplo 5.2

El comando `kill -l` *proporciona una lista de los nombres de señal simbólicos del sistema. Un sistema que se ejecuta bajo el control de Sun Solaris 2 produjo la siguiente muestra de salida.*

```
% kill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE
KILL BUS SEGV SYS PIPE ALRM TERM USR1
USR2 CLD PWR WINCH URG POLL STOP TSTP
CONT TTIN TTOU VTALRM PROF XCPU XFSZ WAITING
LWP FREEZE THAW RTMIN RTMIN+1 RTMIN+2 RTMIN+3 RTMAX-3
RTMAX-2 RTMAX-1 RTMAX
```

Utilice la llamada `kill` al sistema desde un programa en C para enviar una señal a un proceso. Esta llamada toma como parámetros un ID de proceso y un número de señal.

SINOPSIS

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

POSIX.1, Spec 1170

Si el parámetro `pid` es positivo, `kill` envía la señal especificada al proceso con ese ID de proceso. Si el parámetro `pid` es negativo, `kill` envía la señal al grupo de procesos que tienen un ID de grupo igual con |`pid`|. Si `pid` es 0, `kill` envía la señal a los miembros del grupo de procesos al que pertenece el proceso que lo llama. (En el capítulo 7 se estudian los grupos de procesos.) Si la llamada a `kill` tiene éxito, entonces ésta devuelve 0. `kill` puede fallar si quien lo invoca no tiene el permiso para enviar la señal a cualquiera de los procesos especificados por `pid`. En este caso, `kill` devuelve –1 y activa a `errno`. Para muchas señales, `kill` determina los permisos al comparar los ID de usuario de quien lo invoca y del receptor. `SIGCONT` es una excepción. Para `SIGCONT` no se verifican los ID de usuario si el `kill` se envía a un proceso que se encuentra ejecutándose en la misma sesión. (La sección 7.5 estudia las sesiones.)

Ejemplo 5.3

El siguiente segmento de código envía a `SIGUSR1` *al proceso* 3423.

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>

if (kill(3423, SIGUSR1) == -1)
   perror("Could not send signal");
```

Normalmente un programa no debe hacer uso de una ID de proceso específico, tal como el `3423` del ejemplo 5.3. La manera más común de encontrar los ID de los procesos relevantes es con `getpid`, `getppid`, `getgpid` o guardando el valor devuelto por `fork`.

Ejemplo 5.4

El siguiente escenario parece siniestro, pero un proceso hijo puede asesinar a su padre mediante la ejecución del siguiente segmento de código.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

if (kill(getppid(), SIGTERM) == -1)
   perror ("Error in kill");
```

Un proceso puede enviar una señal con la función `raise`. Esta función `raise` sólo necesita un parámetro: el número de la señal.

SINOPSIS

```
#include <signal.h>

int raise(int sig);
```

ISO C, Spec 1170

Ejemplo 5.5

La siguiente instrucción en C hace que un proceso envíe la señal `SIGUSR1` *a sí mismo.*

```
#include <signal.h>

raise(SIGUSR1);
```

Cada vez que se presiona una tecla del teclado, se genera una interrupción por *hardware* que es manejada por el controlador de dispositivo del teclado. Este controlador y sus módulos asociados pueden llevar a cabo operaciones de almacenamiento temporal y edición sobre la entrada proveniente del teclado. Existen dos caracteres, `INTR` y `QUIT`, que hacen que el controlador de dispositivo envíe una señal a los procesos de primer plano. Un usuario puede enviar la señal `SIGINT` a los procesos de primer plano mediante la introducción de carácter `INTR`. Este carácter especificado por el usuario es a menudo `ctrl-c`. De manera similar, el carácter `QUIT` definido por el usuario es a menudo `ctrl-|`, y sirve para enviar la señal `SIGQUIT`.

Ejemplo 5.6

El comando `stty -a` *informa las características del dispositivo asociado con la entrada estándar, incluyendo los valores de los caracteres que generan señales. Un sistema que se ejecuta bajo el control de Sun Solaris 2 generó la siguiente salida.*

```
% stty -a
speed 9600 baud;
rows = 57; columns = 103; ypixels = 0; xpixels = 0;
eucw 1:0:0:0, scrw 1:0:0:0
intr = ^c; quit = ^|; erase = ^?; kill = ^u;
eof = ^d; eol = <undef>; eol2 = <undef>; swtch = <undef>;
start = ^q; stop = ^s; susp = ^z; dsusp = ^y;
rprnt = ^r; flush = ^o; werase = ^w; lnext = ^v;
-parenb -parodd cs8 -cstopb hupcl cread -clocal -loblk -crtscts
-parext -ignbrk brkint ignpar -parmrk -inpck -istrip -inlcr -igncr
icrnl -iuclc ixon -ixany -ixoff imaxbel
isig icanon -xcase echo echoe echok -echonl -noflsh
-tostop echoctl -echoprt echoke -defecho -flusho -pendin iexten
opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel
```

La terminal del ejemplo 5.6 interpreta a `ctrl-c` como el carácter `INTR`. La introducción de `ctrl-c` genera la señal `SIGINT` para los procesos de primer plano. El carácter `QUIT` (`ctrl-|`) genera una señal `SIGQUIT`. El carácter `SUSP` (`ctrl-z`) genera un `SIGSTOP`, mientras que el carácter `DSUSP` (`ctrl-y`) genera un `SIGCONT`.

La función `alarm` hace que se envíe una señal `SIGALRM` al proceso que la llama después de un tiempo específico de segundos.

SINOPSIS

```
#include <unistd.h>

unsigned int alarm(unsigned int seconds);
```

POSIX.1, Spec 1170

Las peticiones hechas a `alarm` no se apilan, de modo que si el programa llama a `alarm` antes de que el primer intervalo de tiempo expire, entonces la `alarm` será inicializada con el nuevo valor. La función `alarm` devuelve el número de segundos que quedan en la `alarm` antes de la llamada que reinicializa el valor. La llamada a `alarm` con un valor cero para `seconds` cancela la solicitud previa de alarma.

Ejemplo 5.7

Puesto que la acción por omisión para `SIGALRM` *es terminar el proceso, el siguiente fragmento de código se ejecuta aproximadamente durante diez segundos.*

```
#include <unistd.h>

void main(void)
{
   alarm(10);
   for ( ; ; ) {
   }
}
```

5.2 Máscara de señal y conjunto de señales

Un proceso puede impedir de manera temporal el depósito de una señal mediante el bloqueo de ésta. Las señales bloqueadas no afectan el comportamiento del proceso hasta que ellas son depositadas. La *máscara de señal* de proceso proporciona el conjunto de señales que serán bloqueadas. La máscara de señal es de tipo `sigset_t`.

El bloquear una señal es diferente de ignorarla. Cuando un proceso bloquea una señal, la aparición de ésta queda congelada hasta que el proceso desbloquea la señal. Un proceso bloquea una señal mediante la modificación con `sigprocmask` de su máscara de señal. Cuando un proceso ignora una señal, ésta es depositada y el proceso la maneja deshaciéndose de ella. El proceso especifica la señal por ignorar llamando a `sigaction` con un manejador de `SIG_IGN`, como se describe en la sección 5.3.

Las operaciones específicas (como el bloqueo o desbloqueo) sobre grupos de señales utilizan conjuntos de señal de tipo `sigset_t` con las funciones siguientes para conjuntos de señales.

SINOPSIS

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

POSIX.1, Spec 1170

La función `sigemptyset` inicia un conjunto de señal de modo que no contenga señales, mientras que la función `sigfillset` inicia un conjunto de señal de modo que éste contenga todas las señales. Antes de utilizar un conjunto de señal, éste debe ser iniciado llamando ya sea a `sigemptyset` o `sigfillset`.

La función `sigaddset` pone una señal específica en el conjunto. La función `sigdelset` quita una señal del conjunto. `sigismember` devuelve 1 si la señal especificada es miembro del conjunto, o 0 si no lo es. Las demás funciones para conjuntos de señales devuelven 0 si tienen éxito o –1 si hay un error. El acceso a los conjuntos de señales debe hacerse siempre a través de las funciones para conjuntos de señales, de modo que el código sea independiente de la implantación.

Ejemplo 5.8

El siguiente segmento de código inicia un conjunto de señal `twosigs` *de modo que contenga exactamente dos señales,* `SIGINT` *y* `SIGQUIT`.

```
#include <signal.h>
sigset_t twosigs;

sigemptyset(&twosigs);
sigaddset(&twosigs, SIGINT);
sigaddset(&twosigs, SIGQUIT);
```

Un proceso puede examinar o modificar su máscara de proceso de señal con la función `sigprocmask`.

SINOPSIS

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
```

POSIX.1, Spec 1170

El parámetro `how` es un entero que indica la manera en que debe modificarse la máscara de señal. Las tres posibilidades para este parámetro de la función `sigprocmask` son

`SIG_BLOCK`:	añade una colección de señales de entre las que se encuentran bloqueadas en ese momento.
`SIG_UNBLOCK`:	borra una colección de señales de entre las que se encuentran bloqueadas en ese momento.
`SIG_SETMASK`:	especifica de una colección dada la colección de señales que serán bloqueadas.

El parámetro `set` de la función `sigprocmask` apunta al conjunto de señales que serán usadas para la modificación, mientras que el parámetro `oset` es la dirección de una variable de tipo `sigset_t` que guarda el conjunto de señales que estaban bloqueadas antes de la llamada a `sigprocmask`. Los parámetros segundo o tercero pueden ser `NULL`. Si el segundo parámetro es `NULL`, el tercero proporciona la máscara que se está utilizando sin ninguna modificación. Si el tercer parámetro es `NULL`, `sigprocmask` no devuelve en `*oset` el valor anterior de la máscara de señal del proceso. Si ocurre un error, `sigprocmask` devuelve `-1` y activa `errno`. Si la llamada tiene éxito, `sigprocmask` devuelve 0.

Ejemplo 5.9

El siguiente segmento de código añade `SIGINT` *al conjunto de señales que el proceso tiene bloqueadas.*

```
#include <stdio.h>
#include <signal.h>
sigset_t  newsigset;

sigemptyset(&newsigset);          /* se inicia con un conjunto nuevo */
sigaddset(&newsigset, SIGINT);       /* se añade SIGINT al conjunto */
if (sigprocmask(SIG_BLOCK, &newsigset, NULL) < 0)
   perror("Could not block the signal");
```

Si `SIGINT` ya está bloqueada, entonces la llamada a `sigprocmask` del ejemplo 5.9 no tiene ningún efecto.

Ejemplo 5.10

El siguiente programa presenta un mensaje, bloquea la señal `SIGINT` *mientras realiza un trabajo que no tiene mucha utilidad, desbloquea la señal y sigue realizando trabajo sin mucha utilidad. El programa repite esta secuencia de manera continua en un ciclo.*

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <signal.h>

void main(int argc,  char * argv[])
{
   double y;
   sigset_t intmask;
   int i, repeat_factor;

   if (argc != 2) {
      fprintf(stderr,"Usage: %s repeat_factor\n", argv[0]);
      exit(1);
   }
   repeat_factor = atoi(argv[1]);
   sigemptyset(&intmask);
   sigaddset(&intmask, SIGINT);
   for( ; ; ) {
      sigprocmask(SIG_BLOCK, &intmask, NULL);
      fprintf(stderr, "SIGINT signal blocked\n");
      for (i = 0; i < repeat_factor; i++) y = sin((double)i);
      fprintf(stderr, "Blocked calculation is finished\n");
      sigprocmask(SIG_UNBLOCK, &intmask, NULL);
      fprintf(stderr, "SIGINT signal unblocked\n");
      for (i = 0; i < repeat_factor; i++) y = sin((double)i);
      fprintf(stderr, "Unblocked calculation is finished\n");
   }
}
```

Si un usuario introduce `ctrl-c` mientras la señal `SIGINT` está bloqueada, el programa del ejemplo 5.10 finalizará el cálculo antes de terminar. Si el usuario introduce `ctrl-c` cuando `SIGINT` no está bloqueada, la ejecución del programa termina de inmediato.

Ejemplo 5.11

La función `makepair` *toma como argumentos el nombre de dos rutas de acceso y crea dos entubamientos con dichos nombres. La función devuelve* 0 *si tiene éxito y* `-1` *si hay un error.*

```
#include <unistd.h>
#include <sys/stat.h>
#include <signal.h>
#include <errno.h>
#define R_MODE S_IRUSR | S_IRGRP | S_IROTH
#define W_MODE S_IWUSR | S_IWGRP | S_IWOTH
#define RW_MODE R_MODE | W_MODE
```

```
int makepair(char *pipe1, char *pipe2)
{
   sigset_t oldmask;
   sigset_t blockmask;
   int returncode = 0;

   sigfillset(&blockmask);
   sigprocmask(SIG_SETMASK, &blockmask, &oldmask);
   if ( ((mkfifo(pipe1, RW_MODE) == -1)
            && (errno != EEXIST)) ||
      ((mkfifo(pipe2, RW_MODE) == -1)
            && (errno != EEXIST)) ) {
       unlink(pipe1);
       unlink(pipe2);
       returncode = -1;
   }
   sigprocmask(SIG_SETMASK, &oldmask, NULL);
   return returncode;
}
```

Si existe un error al establecer cualquiera de los dos entubamientos, la función `makepair` del ejemplo 5.11 desconecta los dos entubamientos y devuelve un error. La función bloquea todas las señales durante la creación de los entubamientos con nombre para asegurarse de que pueda liberar la memoria asignada a los dos entubamientos en caso de que exista un error. Antes de regresar el control a quien la llamó, la función restaura la máscara de señal original. La proposición `if` depende de la evaluación condicional de izquierda a derecha de `&&` y `||`.

Ejercicio 5.1

¿Cuál es el error en la función `makepair` del ejemplo 5.11?

Respuesta:

Si uno de los archivos ya existe, `mkfifo` devuelve `-1` y hace `errno` igual a `EEXIST`. Esta función no determina si el archivo era un FIFO o un archivo ordinario. Es posible que `makepair` indique que tuvo éxito incluso sí el archivo previamente existente no es un FIFO.

Ejemplo 5.12

El siguiente segmento de código bloquea señales durante la creación de un hijo. Tanto el padre como el hijo restauran el valor original de la máscara de señal después del `fork`.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

sigset_t oldmask, blockmask;
pid_t mychild;
```

```
sigfillset(&blockmask);
if (sigprocmask(SIG_SETMASK, &blockmask, &oldmask) == -1) {
   perror("Could not block all signals");
   exit(1);
}
if ((mychild = fork()) == -1) {
   perror("Could not fork child");
   exit(1);
} else if (mychild == 0) {  /* el código del hijo va aquí */
   if (sigprocmask(SIG_SETMASK, &oldmask, NULL) == -1){
      perror("Child could not restore signal mask");
      exit(1);
   }
   /*.....resto del código del hijo..... */
} else {                    /* el código del padre va aquí */
   if (sigprocmask(SIG_SETMASK, &oldmask, NULL) == -1){
      perror("Parent could not restore signal mask");
      exit(1);
   }
   /*.....resto del código del padre..... */
}
```

Los procesos heredan la máscara de señal, tanto después del `fork` como del `exec`. El hijo creado por el `fork` del ejemplo 5.12 tiene una copia de la máscara de señal original guardada en `oldmask`. Un `exec` sobreescribe todas las variables del programa, de modo que un proceso generado por este mecanismo no puede restaurar la máscara de señal original una vez que el `exec` se lleva a cabo.

5.3 Atrapar e ignorar señales —sigaction

La función `sigaction` instala los manejadores de señales de un proceso. La información sobre el manejador está contenida en una estructura de datos de tipo `struct sigaction`. La llamada `sigaction` al sistema tiene tres parámetros: el número de la señal, un apuntador a la estructura (de tipo `struct sigaction`) del nuevo manejador y un apuntador (también de tipo `struct sigaction`) a la estructura del anterior manejador.

Cuando se hace uso de `sigaction` debe llenarse a `*act` con la información del manejador. La función `sigaction` llena la estructura `*oact` con los valores que corresponden al estado anterior de la señal. Si el primer parámetro de `sigaction` es un apuntador `NULL`, entonces nada cambia. Si el programa no necesita el valor del manejador de señal previo, entonces se pasa un apuntador `NULL` para `oact`. La llamada a `sigaction` puede parecer confusa debido a que tiene el mismo nombre que la estructura utilizada para pasar la información de su manejador. Cuando se haga referencia a la estructura, debe hacerse siempre uso de la palabra reservada `struct`.

SINOPSIS

```
#include <signal.h>

int sigaction(in signo const struct sigaction *act,
                          struct sigaction *oact);
struct sigaction {
   void (*sa_handler)(); /* SIG_DFL, SIG_IGN o
                            un apuntador a una función */
   sigset_t sa_mask;     /* las señales adicionales serán bloqueadas
                            durante la ejecución del manejador */
   int sa_flags;         /* banderas especiales y opciones */
};
```

POSIX.1, Spec 1170

La estructura `struct sigaction` anterior muestra tres miembros. POSIX.1b especifica un miembro adicional para señales de tiempo real (sección 5.8). Esta estructura no debe definirse en un programa sino que debe emplearse la definición dada para ella en `signal.h`.

Ejemplo 5.13

El siguiente segmento de código hace que el manejador de señal para `SIGINT` *sea* `mysighand`.

```
#include <signal.h>
#include <stdio.h>
struct sigaction newact;

newact.sa_handler = mysighand;    */ se establece el nuevo manejador */
sigemptyset(&newac.sa_mask);      */ no más señales bloqueadas */
newact.sa_flags = 0;             */ sin opciones especiales */
if (sigaction(SIGINT, &newact, NULL) == -1)
   perror("Could not install SIGINT signal handler");
```

El manejador de señal es una función ordinaria con un valor de regreso de tipo `void` y que tiene un parámetro entero. Cuando el sistema operativo deposita la señal, éste pone en el parámetro el número de la señal que fue depositada. Muchos manejadores de señal ignoran este valor, pero es posible tener un solo manejador para muchas señales. La utilidad de los manejadores de señal está limitada por la incapacidad para pasarles valores. Esta característica se ha añadido a las señales de tiempo real estudiadas en la sección 5.8.

Existen dos valores especiales del miembro `sa_handler` de la estructura `struct sigaction` que son `SIG_DFL` y `SIG_IGN`. El valor `SIG_DFL` especifica que `sigaction` deberá incluir el manejador por omisión para la señal. Los valores `SIG_IGN` indican que el proceso deberá manejar la señal ignorándola.

Ejemplo 5.14

El siguiente fragmento de código hace que el proceso ignore `SIGINT` *si está utilizando el manejador por omisión para esta señal.*

```
#include <signal.h>
#include <stdio.h>
struct sigaction act;
                          */ Se encuentra el manejador de señal original */
if (sigaction(SIGINT, NULL, &act) == -1)
   perror("Could not get old handler for SIGINT");
else if (act.sa_handler == SIG_DFL) {    */ se ignora a SIGINT */
   act.sa_handler == SIG_ING; */ establece el nuevo manejador para
                                 ignorar */
   if (sigaction(SIGINT, &act, NULL) == -1)
      perror("Could not ignore SIGINT");
}
```

Ejemplo 5.15

El siguiente segmento de código configura un manejador de señal que atrapa la señal `SIGINT` *generada por* `ctrl-c`.

```
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

char handmsg[] = "I found ^c\n";
void catch_ctrl_c(int signo)
{
   write(STDERR_FILENO, handmsg, strlen(handmsg));
}
...
struct sigaction act;
...
act.sa_handler = catch_ctrl_c;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
if (sigaction(SIGINT, &act, NULL) < 0)
   /* manejo del error aquí */
...
```

POSIX.1 garantiza que `write` es segura para señales asíncronas, lo que significa que puede invocarse de manera segura desde el interior de un manejador de señal. No existen garantías similares para `fprintf`, pero en algunas implantaciones `fprintf` tal vez sea segura para señales asíncronas. La tabla 5.3 de la página 191 contiene una lista de funciones seguras para señales asíncronas en POSIX y Spec 1170.

Ejemplo 5.16

El siguiente segmento de código hace que la acción asociada con `SIGINT` *sea por la omisión.*

```
#include <stdio.h>
#include <signal.h>
struct sigaction  newact;
```

```
newact.sa_handler = SIG_DFL;    */ el nuevo manejador se establece como el
                                   manejador por omisión */
sigemptyset(&newac.sa_mask);     */ no más señales bloqueadas */
newact.sa_flags = 0;            */ sin opciones especiales */
if (sigaction(SIGINT, &newact, NULL) == -1)
   perror("Could not set SIGINT handler to default action");
```

Ejemplo 5.17

La siguiente función toma como parámetro un número de señal y devuelve 1 *si se ignora dicha señal, o* 0 *en cualquier otro caso.*

```
#include <signal.h>

int test_signal_ignored(int signo)
{
   int returnvalue = 0;
   struct sigaction act;
   if (sigaction(signo, NULL, &act) == -1)
      returnvalue = 0;
   else if (act.sa_handler == SIG_IGN)
      returnvalue = 1;
   return returnvalue;
}
```

5.4 Espera de señales —`pause` y `sigsuspend`

Una de las razones para el empleo de señales es evitar el problema del ocupado en espera. El problema del tiempo ocupado en espera implica emplear ciclos de CPU en espera de que ocurra un evento. Lo común es que el programa haga esto examinando el valor de una variable dentro de un ciclo. Un enfoque más eficiente es suspender el proceso hasta que se presente el evento esperado, de modo que los demás procesos puedan hacer uso de CPU de manera productiva. UNIX proporciona dos funciones que permiten suspender los procesos hasta que ocurra una señal: `pause` y `sigsuspend`.

La llamada a `pause` suspende el proceso que la invoca hasta que sea depositada en éste una señal que no es ignorada por él. Si el proceso atrapa la señal, `pause` regresa después de que el manejador de señal regresa.

SINOPSIS

```
#include <unistd.h>

int pause(void);
```

POSIX.1, Spec 1170

La función `pause` siempre devuelve `-1`, de modo que este valor no tiene ningún significado.

Para esperar una señal en particular hágase uso de `pause`, y verifíquese qué señal es la que hizo que `pause` regresara. Esta información no está disponible de manera directa, de modo que el manejador de señal debe activar una bandera para que el programa la verifique después del regreso de `pause`.

Ejemplo 5.18

El siguiente fragmento de código hace que un proceso espere con `pause` *una señal en particular, dejando que el manejador de señal haga que la variable* `signal_received` *sea* 1. *Al inicio el valor de* `signal_received` *es* 0.

```
#include <unistd.h>
int signal_received = 0;        /* variable estática externa */

...
while(signal_received == 0)
   pause();
```

El ejemplo 5.18 requiere de un lazo, ya que `pause` regresa cuando se deposita en el proceso cualquier señal. Si la señal no es la deseada, `signal_received` tendrá el valor 0 y el ciclo llamará de nuevo a `pause`. La implantación del ejemplo 5.18 supone que la señal no ocurre entre la prueba en el ciclo `while` y el `pause`.

Ejercicio 5.2

¿Qué sucede si la señal del ejemplo 5.18 se deposita entre la prueba de `signal_received` y `pause`?

Respuesta:

La `pause` no regresa hasta que alguna otra señal o cualquier otra aparición de la misma señal sea depositada en el proceso. Una solución que funcione debe probar el valor de `signal_received` mientras la señal se encuentre bloqueada.

Ejemplo 5.19

Lo siguiente es un intento incorrecto de impedir que una señal sea depositada entre la prueba de `signal_received` *y la ejecución de* `pause` *en el ejemplo 5.18.*

```
#include <unistd.h>
#include <signal.h>
int signal_received = 0;        /* variable estática externa */

...
sigset_t sigset;
int signum;

sigemptyset(&sigset);
sigaddset(&sigset, signum);
sigprocmask(SIG_BLOCK, &sigset, NULL);
while(signal_received == 0)
   pause();
```

Desafortunadamente el programa del ejemplo 5.19 ejecuta el `pause` mientras la señal está bloqueada. Como consecuencia de esto, el programa nunca recibe la señal y `pause` nunca regresa.

Si el programa desbloquea la señal antes de ejecutar el `pause`, entonces puede recibir la señal entre el desbloqueo y la ejecución del `pause`. Este evento es más probable de lo que parece. Si se genera una señal mientras el proceso la tiene bloqueada, éste recibe la señal justo después de desbloquearla.

El depósito de una señal antes de `pause` fue uno de los problemas importantes con las señales originales de UNIX, y no había una manera sencilla y confiable de resolver el problema. El programa debe hacer dos operaciones "al mismo tiempo": desbloquear la señal y dar inicio a `pause`. Otra manera de decir lo anterior es que las dos operaciones juntas deben ser atómicas (esto es, el programa no puede ser interrumpido de manera lógica entre la ejecución de una y otra operación). La función `sigsuspend` proporciona un método para hacerlo.

SINOPSIS

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

POSIX.1, Spec 1170

La `sigsuspend` hace que la máscara de señal sea igual a la que apunta `sigmask` y suspende el proceso hasta que se deposite una señal en él. La `sigmask` puede emplearse para desbloquear la señal que el programa está esperando. Cuando `sigsuspend` regresa, se restablece el valor que la máscara de señal tenía antes de `sigsuspend`. La `sigsuspend` siempre devuelve `-1`, de modo que el valor regresado no tiene significado alguno.

Ejemplo 5.20

El siguiente fragmento de código suspende un proceso hasta que ocurra la señal dada por el valor de `signum`. *Después restaura el valor original de la máscara de señal.*

```
#include <signal.h>
int signal_received = 0;        /* variable estática externa */
...
sigset_t sigset;
sigset_t sigoldmask;
int signum;

sigprocmask(SIG_SETMASK, NULL, &sigoldmask);
sigprocmask(SIG_SETMASK, NULL, &sigset);
sigaddset(&sigset, signum);
sigprocmask(SIG_BLOCK, &sigset, NULL);
sigdelset(&sigset, signum);
while(signal_received == 0)
    sigsuspend(&sigset);
sigprocmask(SIG_SETMASK, &sigoldmask, NULL);
```

El segmento de código del ejemplo 5.20 supone que el valor inicial de `signal_received` es 0 y que el manejador para `signum` cambia a 1 el valor de `signal_received`. Es impor-

tante que la señal sea bloqueada mientras el `while` hace la prueba de `signal_received`. De otro modo, la señal puede ser depositada entre la prueba de `signal_received` y la llamada a `sigsuspend`. En este caso el proceso queda bloqueado hasta que otra señal haga que `sigsuspend` regrese.

Ejercicio 5.3

Suponga que el `sigsuspend` del ejemplo 5.20 regresa por causa de una señal diferente. ¿Queda bloqueada la señal `signum` cuando el `while` prueba otra vez la variable `signal_received`?

Respuesta:

Sí, ya que después del regreso de `sigsuspend`, la máscara de señal vuelve a tener el estado que tenía antes de `sigsuspend`. La llamada a `sigprocmask` antes del `while` garantiza que esta señal quede bloqueada.

5.5 Ejemplo —biff

En la sección 2.8 se presentó un programa sencillo para notificar a un usuario cuando tiene correo. El programa 5.1 muestra una versión más compleja de este programa, que hace uso de `stat` para determinar el tamaño de archivo de correo con objeto de notificar al usuario cuando éste cambia de tamaño, lo que indica que ha llegado correo nuevo. El programa también tiene un mecanismo para habilitar o deshabilitar la notificación sin dar por terminado el proceso. Estas características hacen que el programa pueda emplearse como un verdadero demonio que corre todo el tiempo pero que sólo se activa cuando el usuario así lo desea.

El programa hace uso de dos señales de usuario, `SIGUSR1` y `SIGUSR2`, para habilitar o deshabilitar la notificación, e ilustra varios aspectos importantes con respecto al empleo de señales. Los dos manejadores de señales cambian la variable global `notifyflag`. Es importante que ningún manejador de señal sea interrumpido por otra señal. Durante la ejecución de un manejador de señal, la señal que provocó la ejecución de éste queda bloqueada de manera predeterminada. La `sigaction` permite que el programa bloquee también otras señales. En este programa las dos señales quedan bloqueadas cuando la ejecución entra en cualquiera de los manejadores. El miembro `sa_mask` de la estructura `sigaction` especifica las señales que deben ser bloqueadas durante la ejecución del manejador.

El programa también ilustra la manera correcta de esperar que la variable global cambie de valor. Es necesario bloquear las señales de usuario antes de probar `notifyflag`.

Ejercicio 5.4

¿Cuál es el error en el siguiente fragmento de código que espera a que `notifyflag` cambie de valor?

```
#include <unistd.h>
int notifyflag = 0;   /* variable estática externa */
...
while (notifyflag == 0)
   pause();
```

Respuesta:

Suponga que `notifyflag` es 0 y que se deposita la señal `SIGUSR1` en el proceso después de examinar `notifyflag` pero antes de ejecutar `pause()`. La `pause()` queda bloqueada hasta que se deposite la *siguiente* señal.

Ejercicio 5.5

Puesto que `sigsuspend` desbloquea las señales ¿cuál es la finalidad de la segunda llamada a `sigprocmask` en el programa 5.1?

Respuesta:

No se llama a `sigsuspend` si `notifyflag` es 1. Por otra parte, cuando `sigsuspend` regresa, restaura la máscara de señal al valor que ésta tenía antes de la llamada.

Programa 5.1: Programa `biff` que utiliza señales.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <pwd.h>
#define MAILDIR "/var/mail/"
#define SLEEPTIME 10
int notifyflag = 1;

 /* Si no hay error, se devuelve el tamaño del archivo. De lo contrario,
  * si el archivo no existe se devuelve 0, y -1 en cualquier otro caso.
  */
int get_file_size(const char *filename)
{
   struct stat buf;

   if (stat(filename, &buf) == -1) {
      if (errno == ENOENT) return 0;
      else return -1;
   }
   return (long)buf.st_size;
}

/* Se notifica al usuario la llegada de correo
void send_mail_notification()
{
   fprintf(stderr, "Mail has arrived\007\n");
}
```

```
/* Se habilita la notificación */
void turn_notify_on(int s)
{
   notifyflag = 1;
}

/* Se inhabilita la notificación */
void turn_notify_off(int s)
{
   notifyflag = 0;
}

/* Se verifica de manera continua la llegada del correo y se notifica al
   usuario que hay correo nuevo */
int notify_of_mail(const char *filename)
{
   long old_mail_size;
   long new_mail_size;
   sigset_t blockset;
   sigset_t emptyset;

   sigemptyset(&emptyset);
   sigemptyset(&blockset);
   sigaddset(&blockset, SIGUSR1);
   sigaddset(&blockset, SIGUSR2);
   old_mail_size = get_file_size(filename);
   if (old_mail_size < 0) return 1;
   if (old_mail_size > 0) send_mail_notification();
   sleep(SLEEPTIME);
   for( ; ; ) {
      if (sigprocmask(SIG_BLOCK, &blockset, NULL) < 0)
         return 1;
      while (notifyflag == 0)
         sigsuspend(&emptyset);
      if (sigprocmask(SIG_SETMASK, &emptyset, NULL) < 0)
         return 1;
      new_mail_size = get_file_size(filename);
      if (new_mail_size > old_mail_size)
         send_mail_notification();
      old_mail_size = new_mail_size;
      sleep(SLEEPTIME);
   }
}

void main(void)
{
   char mailfile[80];
   struct sigaction newact;
   struct passwd *pw;
```

```
    if ((pw = getpwuid(getuid())) == NULL) {
        perror("Could not determine login name");
        exit(1);
    }
    strcpy(mailfile, MAILDIR);
    strcat(mailfile, pw->pw_name);
    newact.sa_handler = turn_notify_on;
    newact.sa_flags = 0;
    sigemptyset(&newact.sa_mask);
    sigaddset(&newact.sa_mask, SIGUSR1);
    sigaddset(&newact.sa_mask, SIGUSR2);
    if (sigaction(SIGUSR1, &newact, NULL) < 0)
        perror("Could not set signal to turn on notification");
    newact.sa_handler = turn_notify_off;
    if (sigaction(SIGUSR2, &newact, NULL) < 0)
        perror("Could not set signal to turn off notification");
    notify_of_mail(mailfile);
    fprintf(stderr, "Fatal error, terminating program\n");
    exit(1);
}
```

Programa 5.1

Ejercicio 5.6

¿Bajo qué circunstancias falla el programa 5.1 en la notificación de la llegada de correo?

Respuesta:

Si llega un mensaje breve justo después de que el usuario ha leído un mensaje grande, es posible que `biff` no lo detecte.

5.6 Llamadas al sistema y señales

Existen dos dificultades que pueden presentarse cuando las señales interaccionan con las llamadas al sistema. La primera de ellas tiene que ver con el hecho de determinar si deben reiniciarse las llamadas al sistema que fueron interrumpidas por señales. El otro problema se presenta con las llamadas al sistema no reentrantes que están en los manejadores de señal.

¿Qué sucede cuando un proceso atrapa una señal mientras está ejecutando una llamada al sistema? La respuesta depende del *tipo de llamada al sistema*. Algunas llamadas al sistema, como las que hacen operaciones de E/S, pueden bloquear el proceso por un lapso indeterminado. No hay ningún límite al tiempo que debe transcurrir para obtener el valor de una tecla del teclado o para hacer la lectura de un entubamiento. Este tipo de llamadas se conocen como llamadas "lentas" al sistema. Existen otras, como las que hacen operaciones de E/S en disco, que pueden bloquear el proceso por poco tiempo. Incluso existen otras, como `getpid()`, que no realizan ningún bloqueo. Ninguna de estas últimas es considerada como "lenta". Se necesita la capacidad de interrupción de las llamadas lentas al sistema, para que el sistema operativo pueda depositar señales a los programas durante esperas muy largas de operaciones de E/S.

En POSIX.1, las llamadas al sistema lentas que son interrumpidas devuelven –1 y hacen que `errno` sea `EINTR`. El programa debe manejar este error explícitamente y reiniciar la llamada al sistema si así se desea. Puesto que la E/S puede redireccionarse, el compilador tal vez no sea capaz de resolver durante la compilación si una llamada de E/S pertenece a la categoría de llamadas lentas. Lo mejor que puede hacerse por el lado seguro es insertar el código necesario para el reinicio cada vez que esto sea relevante.

Ejemplo 5.21

El siguiente segmento de código reinicia la llamada al sistema `read` *si ésta es interrumpida por una señal.*

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include <errno.h>
int fd;
int retval;
int  size;
char *buf;

while (retval = read(fd, buf, size),
                retval == -1 && errno == EINTR)
   ;
if (retval == -1)
    /* manejo de errores aquí */
```

Nótese el uso del operador coma en el ejemplo 5.21 y del punto y coma después del `while`. Al regresar, `read` establece el valor de `retval`. El `while` continúa siempre y cuando el segundo parámetro del operador coma sea verdadero, esto es, siempre y cuando el `read` falle debido a que fue interrumpido por una señal. En este caso, el `while` vuelve a iniciar el `read`.

Bajo Spec 1170 el programa puede especificar en la llamada a `sigaction` que las llamadas lentas al sistema deban reiniciarse. El reinicio automático no es parte de POSIX.1, aunque el estándar no prohíbe esta extensión. La especificación de reinicio se logra mediante el empleo de la opción `SA_RESTART` en el miembro `sa_flags` de la estructura `struct sigaction` del manejador. Si se escriben bibliotecas de funciones, procúrese no depender de que el programa que hace la llamada configure los manejadores para que éstos reinicien las llamadas al sistema. Algunas veces la interrupción de una llamada al sistema mediante una señal es conveniente. El siguiente ejemplo utiliza una señal para evitar que una llamada al sistema quede bloqueada indefinidamente.

Ejemplo 5.22

El siguiente código maneja la lectura de un entubamiento en un descriptor de archivo `fd` *con un tiempo de espera de 10 segundos.*

```
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>
```

```
#include <errno.h>
int fd;
int size;
char *buf;

alarm(10);
if (read(fd, buf, size) < 0) {
   if (errno == EINTR)
      /* handle timeout here */
   else
      /* manejo de otros errores aquí */
}
alarm(0);
```

La llamada `alarm` al sistema instruye al reloj de alarma de los procesos que la llaman a que envíe una señal `SIGALRM` al proceso después de un número específico de segundos. La acción por omisión para `SIGALRM` es terminar el proceso. El ejemplo 5.22 supone que el programa ha instalado un manejador de señal para atrapar esta señal sin el reinicio automático de las llamadas al sistema. Finalmente, `alarm(0)` apaga el temporizador de la alarma. Un enfoque alternativo es emplear `select` con un tiempo de espera para E/S.

Recuérdese que una función es *segura con respecto de señales asíncronas* si puede invocarse de manera segura desde el interior de un manejador de señal. Muchas de las llamadas al sistema de UNIX así como de funciones de bibliotecas no son seguras en relación con las señales asíncronas debido a que se utilizan estructuras de datos estáticas, llamadas a `malloc` o `free`, o utilizan estructuras de datos globales de manera no reentrante. En consecuencia, un solo proceso no puede ejecutar de manera correcta dos instancias concurrentes de estas llamadas.

Normalmente lo anterior no es ningún problema, pero las señales añaden concurrencia al programa. Puesto que las señales se presentan de manera asíncrona, es posible que un proceso atrape una señal mientras está ejecutando una llamada al sistema o una función contenida en una biblioteca. (Por ejemplo, ¿qué sucede si el programa interrumpe a `strtok` con otro `strtok` y luego intenta continuar la primera llamada?) Por tanto, debe tenerse cuidado cuando se ejecutan llamadas al sistema dentro de los manejadores de señales. Cada estándar UNIX tiene una lista de llamadas al sistema que pueden utilizarse de manera segura dentro de los manejadores de señales. La tabla 5.3 contiene una lista con las funciones que POSIX.1 y Spec 1170 garantizan que pueden llamarse de manera segura desde un manejador de señal. Además, Spec 1170 requiere que `fpathconf`, `raise` y `signal` sean seguras con respecto a señales asíncronas.

El manejo de señales es complicado; a continuación se proporcionan unas cuantas reglas útiles.

- Cuando haya duda en un programa, reinice explícitamente las llamadas al sistema.
- Para hacer seguro un manejador de señal, compruebe que cada llamada al sistema o función de biblioteca empleada en él se encuentre en la lista de llamadas *seguras en cuanto a señales asíncronas.*
- Analice con cuidado las posibles interacciones entre un manejador de señal que cambia una variable externa y otro código del programa que tenga acceso a dicha variable. Bloquee las señales para evitar interacciones no deseadas.

_exit()	fstat()	read()	sysconf()
access()	getegid()	rename()	tcdrain()
alarm()	geteuid()	rmdir()	tcflow()
cfgetispeed()	getgid()	setgid()	tcflush()
cfgetospeed()	getgroups()	setpgid()	tcgetattr()
cfsetispeed()	getpgrp()	setsid()	tcgetpgrp()
cfsetospeed()	getpid()	setuid()	tcsendbreak()
chdir()	getppid()	sigaction()	tcsetattr()
chmod()	getuid()	sigaddset()	tcsetpgrp()
chown()	kill()	sigdelset()	time()
close()	link()	sigemptyset()	times()
creat()	lseek()	sigfillset()	umask()
dup2()	mkdir()	sigismember()	uname()
dup()	mkfifo()	sigpending()	unlink()
execle()	open()	sigprocmask()	utime()
execve()	pathconf()	sigsuspend()	wait()
fcntl()	pause()	sleep()	waitpid()
fork()	pipe()	stat()	write()

Tabla 5.3: Funciones que POSIX.1 y Spec 1170 especifican como seguras en cuanto a señales asíncronas.

5.7 siglongjmp y sigsetjmp

En ocasiones los programas emplean señales para manejar errores que no son fatales pero que pueden presentarse en muchas partes de un programa. Por ejemplo, es posible que un usuario desee abortar un cálculo largo o una operación de E/S que ha quedado bloqueada por mucho tiempo pero sin terminar el programa. La respuesta de un programa a `ctrl-c` debe ser el comenzar otra vez desde el principio (o en algún otro lugar especificado). El programa puede hacer uso de las señales de manera indirecta o directa para manejar esta situación.

En el enfoque indirecto, el manejador de señal para `SIGINT` establece el valor de una bandera. El programa prueba la bandera en lugares estratégicos y regresa al ciclo principal si la bandera tiene el valor apropiado. El enfoque indirecto es complicado puesto que el programa tal vez tenga que regresar a través de varias capas de funciones. En cada retorno a una capa, el programa prueba el valor devuelto para este caso especial.

En el enfoque directo, el manejador de señal salta de regreso directamente al programa principal. El salto requiere desenmarañar la pila del programa. Existe un par de funciones que proporcionan esta capacidad: `sigsetjmp` y `siglongjmp`. `sigsetjmp` es análogo a una instrucción de etiqueta, mientras que `sigsetjmp` es el análogo a un `goto`. La diferencia principal es que `sigsetjmp` y `sigsetjmp` proporcionan un mecanismo para limpiar la pila y los estados de la señal, además de llevar a cabo el salto.

SINOPSIS

```
#include <setjmp.h>

int sigsetjmp(sigjmp_buf env, int savemask);
void siglongjmp(sigjmp_buf env, int val);
```

POSIX.1, Spec 1170

La llamada a `sigsetjmp` debe hacerse en el punto del programa donde ésta debe regresar. `sigsetjmp` proporciona un marcador en el programa similar a la proposición de etiqueta. El programa que invoca la función debe proporcionar un *buffer* de tipo `sigjmp_buf`, el cual inicia `sigsetjmp` para recopilar la información necesaria cuando se haga el salto de regreso al marcador. Si `savemask` es distinto de cero, el estado que tiene la máscara de señal se guarda en el *buffer* `env`. Cuando el programa llama a `sigsetjmp` directamente, la función devuelve 0. Para saltar de regreso al punto `sigsetjmp` desde un manejador de señal, se ejecuta a `sigsetjmp` con la misma variable `sigjmp_buf`. La llamada hace que parezca que el programa regresa de `sigsetjmp` con un valor de regreso dado por `val`.

La biblioteca estándar de C proporciona las funciones `setjmp` y `longjmp` para los tipos de saltos mencionados en los párrafos anteriores. La razón para utilizar `sigsetjmp` y `sigsetjmp` más que `setjmp` y `longjmp` es que la acción del último par sobre la máscara de señal depende del sistema. `sigsetjmp` permite que el programa especifique si la máscara de señal debe restablecerse cuando se haga el salto fuera del manejador de señal.

El programa 5.2 muestra cómo establecer un manejador para `SIGINT` que haga que el programa regrese al ciclo principal cuando se teclee la combinación `ctrl-c`. Es importante ejecutar `sigsetjmp` antes de llamar a `siglongjmp` para poder establecer un punto de regreso. La llamada a `sigaction` debe aparecer antes de `sigsetjmp` de modo que ésta sea llamada sólo una vez. Para evitar que el manejador de señal llame a `siglongjmp` antes de que el programa ejecute `sigsetjmp`, el programa 5.2 utiliza la bandera `jumpok`. El manejador de señal examina esta bandera antes de llamar a `siglongjmp`. El calificador `volatile` impide que esta variable sea colocada en un registro, y el tipo `sig_atomic_t` indica que la escritura en ella no podrá ser interrumpida por una señal.

Programa 5.2: Código para establecer un manejador de señal que regrese al ciclo principal cuando se escriba `ctrl-c`.

```
#include <signal.h>
#include <stdio.h>
#include <setjmp.h>

static volatile sig_atomic_t jumpok = 0;
static sigjmp_buf jmpbuf;
...
void int_handler(int errno)
{
```

```
    if (jumpok == 0) return;
    siglongjmp(jmpbuf, 1);
}
...
void main(void)
{
    struct sigaction act;
          ...
    act.sa_handler = int_handler;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    if (sigaction(SIGINT, &act, NULL) < 0) {
       perror("Error setting up SIGINT handler");
       exit(1);
    }
          ...
    if (sigsetjmp(jmpbuf, 1))
       fprintf(stderr, "Returned to main loop due to ^c\n");
    jumpok = 1;
          ...   /* inicio del ciclo principal */
}
```

Programa 5.2

5.8 Señales de tiempo real

En POSIX.1 un manejador de señal es una función con un solo parámetro que representa el número de señal de la señal generadora. POSIX.1b introduce varias mejoras a POSIX.1 en cuanto a las capacidades para el manejo de señales, incluyendo la puesta en cola de las mismas y el paso de información a los manejadores de señal. El estándar extiende la estructura `sigaction` con el fin de permitir parámetros adicionales para el manejador de señal. Si `_POSIX_REALTIME_SIGNALS` está definida, entonces la implantación soporta estas capacidades extendidas, lo que se conoce como RealTime Signal Extension.

SINOPSIS

```
#include <signal.h>

struct sigaction {
    void (*sa_handler)(); /* SIG_DFL, SIG_IGN, o
                             un apuntador a una función */
    void (*sa_handler)(int, signfo_t *, void *);
    sigset_t sa_mask;    /* señales adicionales que serán bloqueadas
                            durante la ejecución del manejador */
    int sa_flags;        /* banderas especiales y opciones */
};
```

POSIX.1b, Spec 1170

El nuevo miembro `sa_sigaction` especifica un tipo alternativo para el manejador de señal. El nuevo tipo del manejador se emplea si `sa_flags && SA_SIGINFO` es verdadero. La forma del nuevo manejador es

```
void func(int signo, siginfo_t *info, void *context);
```

El parámetro `signo` es el mismo que antes y se le asigna un valor igual al número de la señal que se desea atrapar. En este momento `context` no está definida en el estándar POSIX. La estructura `*info` contiene al menos los siguientes miembros:

```
int si_signo ;              /* número de señal */
int si_code;                /* causa de la señal */
union sigval si_value;      /* valor de la señal */
```

El `si_signo` contiene el número de la señal. Este valor es el mismo que está almacenado en `signo`. El `si_code` indica la causa de la señal. Los valores posibles son `SI_USER`, `SI_QUEUE`, `SI_TIMER`, `SI_ASYNCHIO` y `SI_MESGQ`. Un valor `SI_USER` indica que la señal fue generada por `kill`, `raise` o `abort`. En estas situaciones no hay manera de generar un `si_value`, de modo que éste no se encuentra definido. `SI_QUEUE` indica que la función `sigqueue` generó la señal. `SI_TIMER` señala que el origen de la señal fue un temporizador. Las secciones 6.2 y 6.9 presentan estos temporizadores. Un valor `SI_ASYNCIO` indica el término de una operación de E/S asíncrona (sección 5.9) y uno `SI_MESGQ` señala la llegada de un mensaje a una cola de mensajes vacía.

La unión `sigval` debe tener al menos los siguientes miembros:

```
int sival_int;
void *sival_ptr;
```

para permitir la transmisión de señal al manejador, ya sea de un entero o de un apuntador.

Como ya se mencionó, el estándar POSIX.1 no especifica lo que sucede cuando están pendientes varias instancias de una señal. POSIX garantiza el depósito de al menos una de las instancias cuando la señal sea desbloqueada, aunque es posible que las demás se pierdan. La función `sigqueue` es una extensión de `kill` que permite que las señales sean colocadas en una cola.

SINOPSIS

```
#include <signal.h>

int sigqueue(pid_t pid, int signo, const union sigval value);
```

POSIX.1b

El parámetro adicional especifica el valor que será recibido por el manejador de señal en el miembro `si_value` del parámetro `info` de éste.

Para garantizar que varias instancias de una señal en particular sean colocadas en una cola, el programa debe utilizar la bandera `SA_SIGINFO` en `sa_flags` cuando se establezca el

manejador con `sigaction`. Las distintas señales del mismo número de señal generadas por `sigqueue` son colocadas en una cola hasta un máximo dado por `SIGQUEUE_MAX`, que por lo general tiene el valor de 32.

La garantía de que las señales sean puestas en una cola está determinada por el método empleado para generar las señales y no por la forma en que se establece el manejador de señal. Es posible que las diversas instancias de una señal generadas por `kill` no sean puestas en una cola, aun si esto ocurre con las instancias de la misma señal generadas por `sigqueue`.

El programa 5.3 muestra un programa que envía señales puestas en una cola a otros procesos. El programa se comporta como el comando `kill` pero utiliza `sigqueue` en lugar de la llamada al sistema `kill`. El programa permite el envío de señales en una cola así como de un valor entero para cada una de ellas. El programa 5.4 muestra un programa que imprime su ID de proceso, establece un manejador de señal para `SIGUSR1`, bloquea la señal y espera una entrada antes de terminar su ejecución. El manejador de señal sólo presenta los valores que recibe a partir de sus parámetros.

Programa 5.3: Programa que envía una señal en una cola a un proceso.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void main(int argc, char *argv[])
{
   union sigval qval;
   int val;
   int pid;
   int signo;

   if (argc != 4) {
      fprintf(stderr, "Usage: %s pid signal value\n", argv[0]);
      exit(1);
   }

   pid = atoi(argv[1]);
   signo = atoi(argv[2]);
   val = atoi(argv[3]);

   fprintf(stderr, "Sending signal %d with value %d to process %d\n",
      signo, val, pid);
   qval.sival_int = val;
   sigqueue(pid, signo, qval);
}
```

Programa 5.3

Programa 5.4: Programa que recibe señales `SIGUSR1`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void my_handler(int signo, siginfo_t* info, void *context)
{
   fprintf(stderr, "Signal handler entered for signal number %d\n",
      signo);
   fprintf(stderr, "   si_signo = %3d\n", info->si_signo);
   fprintf(stderr, "   si_code  = ");

   if (info->si_code == SI_USER) fprintf(stderr, "USER\n");
   else if (info->si_code == SI_QUEUE) fprintf(stderr, "QUEUE\n");
   else if (info->si_code == SI_TIMER) fprintf(stderr, "TIMER\n");
   else if (info->si_code == SI_ASYNCIO) fprintf(stderr, "ASYNCIO\n");
   else if (info->si_code == SI_MESGQ) fprintf(stderr, "MESGQ\n");
   else fprintf(stderr, "%d\n", info->si_code);
   fprintf(stderr,"   si_value = %3d\n", info->si_value.sival_int);
}

void main(void)
{
   struct sigaction act;
   sigset_t sigset;

   fprintf(stderr, "Process ID is %ld\n", (long)getpid());

   sigemptyset(&sigset);
   sigaddset(&sigset, SIGALRM);
   sigprocmask(SIG_BLOCK, &sigset, NULL);
   sigemptyset(&act.sa_mask);
   act.sa_flags = SA_SIGINFO;
   act.sa_sigaction = my_handler;
   if (sigaction(SIGUSR1, &act, NULL) < 0) {
      perror("Sigaction failed");
      exit(1);
   }
   fprintf(stderr, "Signal SIGUSR1 = %d ready\n", SIGUSR1);

   for( ; ; )
      pause();
}
```

Programa 5.4

5.9 E/S asíncrona

Cuando un proceso lleva a cabo una operación de lectura o escritura normalmente, se bloquea hasta que la operación de E/S termine. Algunas veces el proceso debe más bien iniciar la petición y continuar con su ejecución sin tener que esperar a que la E/S esté completa. El primer tipo de solicitud se conoce como *solicitud con bloqueo*, mientras que el segundo recibe el nombre de *solicitud sin bloqueo*.

Ejemplo 5.23

El siguiente segmento de código abre el FIFO existente `myfifo` *para lectura sin bloqueo.*

```
#include <stdio.h>
#include <fcntl.h>
int fd;

if ((fd = open("myfifo", O_RDONLY | O_NONBLOCK)) == -1)
   perror("Could not open myfifo");
```

Si no hay ningún dato disponible en el FIFO del ejemplo 5.23, el `read` de `fd` devuelve -1 con un `errno` igual a `EAGAIN`.

En una operación de E/S asíncrona, el proceso emite una operación de E/S sin bloqueo, y el sistema operativo le notifica el momento en que la E/S está completa. Spec 1170 emplea a `SIGPOLL` para notificación de E/S asíncrona, mientras que el viejo BSD 4.3 hace uso de `SIGIO`. La E/S asíncrona no es parte del estándar POSIX.1 original, pero POSIX.1b tiene una extensión E/S asíncrona descrita más adelante.

Ejemplo 5.24

El siguiente segmento de código muestra cómo establecer un descriptor de archivo para operaciones de E/S asíncronas, como lo describe para Spec 1170.

```
#include <unistd.h>
#include <stropts.h>
#include <fcntl.h>

if ((fd = open(pathname, O_RDONLY | O_NONBLOCK)) == -1)
    /* manejador de errores aquí */
if (ioctl(fd, I_SETSIG, S_RDNORM) == -1)
    /* no fue posible establecer el descriptor de archivo para lectura
       asíncrona */
```

El parámetro `S_RDNORM` del ejemplo 5.24 indica que `SIGPOLL` deberá generarse cuando llegue a `fd` un mensaje ordinario. El `fd` debe ser un flujo de datos, como en la sección 12.6. Los archivos, los entubamientos, los FIFO y la E/S en red se incluyen en esta categoría. Una vez establecido el descriptor de archivo para E/S asíncrona, el controlador de dispositivo genera una señal `SIGPOLL` si hay E/S (en el ejemplo 5.24, ésta es un mensaje ordinario) disponi-

ble para dicho descriptor de archivo. Es evidente que primero se debe asegurar el establecimiento de un manejador para SIGPOLL antes de establecer un descriptor de archivo para su uso en operaciones de E/S asíncronas. La acción predeterminada para SIGPOLL es terminar el proceso. La sección 9.1.2 proporciona un ejemplo completo de una E/S asíncrona utilizando a SIGPOLL.

El POSIX.1b Realtime Extension define la E/S asíncrona con base en aio_read y aio_write. aio_read permite que un proceso ponga en una cola la solicitud de lectura para un descriptor de archivo abierto. El papel de aio_write es similar. aio_return devuelve el estado de la operación de E/S asíncrona, y aio_error devuelve el estado de error de la operación de E/S asíncrona.

SINOPSIS

```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
int aio_write(struct aiocb *aiocbp);
ssize_t aio_return(struct aiocb *aiocbp);
int aio_error(const struct aiocb *aiocbp);
```

POSIX.1b

La estructura struct aiobc tiene al menos los siguientes miembros:

```
int             aio_fildes;     /* descriptor de archivo */
volatile void   *aio_buf;       /* localización del buffer */
size_t          aio_nbytes;     /* longitud de la transferencia */
off_t           aio_offset;     /* desplazamiento de archivo */
int             aio_reqprio;    /* solicitud de desplazamiento de prioridad */
struct sigevent aio_sigevent;   /* número de señal y desplazamiento */
int             aio_lio_opcode; /* operación listio */
```

Los tres primeros miembros de esta estructura son similares a los parámetros de un read o write ordinario. El aio_offset especifica la posición de inicio para la operación de E/S. Si la implantación soporta la planificación por parte del usuario (_POSIX_PRIORITIZED_IO y _POSIX_PRIORITY_SCHEDULING están definidas), aio_reqprio reduce la prioridad de la solicitud. El aio_sigevent especifica la forma en que se notificará al proceso el término de la operación. Si aio_sigevent.sigev_notify tiene el valor SIGEV_NONE, el sistema operativo no envía una señal al término de la operación de E/S. Si aio_sigevent.sigev_notify tiene el valor SIGEV_SIGNAL, el sistema operativo genera la señal especificada en aio_sigevent.sigev_signo. El aio_lio_opcode lo utiliza la función lio_listio para enviar varias solicitudes de E/S.

El programa 5.5 ilustra el uso de la E/S asíncrona en POSIX. El programa vigila dos dispositivos de entrada lentos cuyos nombres se pasan como argumentos de la línea comando uno y dos. El programa 5.5 abre los dos descriptores para entrada. El programa pone en una cola la solicitud de lectura de fd_1 al llamar a aio_read. El sistema operativo deposita una señal SIGRTMAX cuando la operación está completa. De manera similar, el programa pone en la cola la primera solicitud de lectura para fd_2 especificando la notificación con la señal SIGRTMAX-1.

El manejador para cada una de estas señales es `my_aio_handler`. El manejador hace uso de su segundo parámetro para determinar cuál fue la señal depositada. Después escribe la información que recibió como entrada en el dispositivo de error estándar e inicia otra lectura asíncrona para dicho descriptor. La E/S asíncrona de POSIX.1b no estaba soportada al momento de la publicación de este libro en ninguno de los sistemas a los que los autores tienen acceso, así que se advierte al lector que el código del programa 5.5 no ha sido probado.

Programa 5.5: Programa para vigilar dos descriptores de archivo utilizando E/S asíncrona.

```
#include <aio.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stropts.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define BLKSIZE 1024

volatile char buf2[BLKSIZE];
volatile char buf1[BLKSIZE];
int fd_1 = 0;
int fd_2 = 0;
int fd1_error = 0;
int fd2_error = 0;
struct aiocb my_aiocb1;
struct aiocb my_aiocb2;

void my_aio_handler(int signo, siginfo_t *info, void *context)
{
   int  my_errno;
   int  my_status;
   struct aiocb *my_aiocbp;
   int *errorp;

   my_aiocbp = info->si_value.sival_ptr;
   if (signo == SIGRTMAX)
      errorp = &fd1_error;
   else
      errorp = &fd2_error;
   if ((my_errno = aio_error(my_aiocbp)) != EINPROGRESS) {
      my_status = aio_return(my_aiocbp);
      if (my_status >= 0) {
         write(STDERR_FILENO, (char *)my_aiocbp->aio_buf, my_status);
         *errorp = aio_read(my_aiocbp);
```

```
        }
        else
            *errorp = 1;
    }
}

void main(int argc, char *argv[])
{
    sigset_t oldmask;
    struct sigaction newact;
                            /* apertura de los descriptores de archivo para E/S */
    if (argc != 3) {
        fprintf(stderr, "Usage: %s filename1 filename2\n", argv[0]);
        exit(1);
    }
    if ((fd_1 = open(argv[1], O_RDONLY)) == -1) {
        fprintf(stderr,"Could not open %s: %s\n",
            argv[1], strerror(errno));
        exit(1);
    }
    if ((fd_2 = open(argv[2], O_RDONLY)) == -1) {
        fprintf(stderr,"Could not open %s: %s\n",
            argv[2], strerror(errno));
        exit(1);
    }
                /* Establecimiento de manejadores para SIGRTMAX y SIGRTMAX-1 */
    sigemptyset(&newact.sa_mask);
    sigaddset(&newact.sa_mask, SIGRTMAX);
    sigaddset(&newact.sa_mask, SIGRTMAX-1);
    if (sigprocmask(SIG_BLOCK, &newact.sa_mask, &oldmask) == -1) {
        perror("Could not block SIGRTMAX or SIGRTMAX-1");
        exit(1);
    }
    newact.sa_sigaction = my_aio_handler;
    newact.sa_flags = SA_SIGINFO;
    if (sigaction(SIGRTMAX, &newact, NULL) == -1) {
        perror("Could not set SIGRTMAX handler");
        exit(1);
    }
    if (sigaction(SIGRTMAX-1, &newact, NULL) == -1) {
        perror("Could not set SIGRTMAX-1 handler");
        exit(1);
    }
                                            /* Desbloqueo de señales */
    if (sigprocmask(SIG_UNBLOCK, &newact.sa_mask, NULL) == -1) {
        perror("Could not unblock SIGRTMAX or SIGRTMAX-1");
        exit(1);
    }
                /* Inicio de la primera operación de E/S sobre fd_1 */
```

```
    my_aiocb1.aio_fildes = fd_1;
    my_aiocb1.aio_offset = 0;
    my_aiocb1.aio_buf = (void *)buf1;
    my_aiocb1.aio_nbytes = BLKSIZE;
    my_aiocb1.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    my_aiocb1.aio_sigevent.sigev_signo = SIGRTMAX;
    my_aiocb1.aio_sigevent.sigev_value.sival_ptr = &my_aiocb1;
    fd1_error = aio_read(&my_aiocb1);
    if (fd1_error == -1) {
        if (errno == ENOSYS)
           fprintf(stderr,"                    !!!!! Not supported yet\n");
        else
           perror("The aio_read failed");
        exit(1);
    }

                    /* Inicio de la primera operación de E/S sobre fd_2 */
    my_aiocb2.aio_fildes = fd_2;
    my_aiocb2.aio_offset = 0;
    my_aiocb2.aio_buf = (void *)buf2;
    my_aiocb2.aio_nbytes = BLKSIZE;
    my_aiocb2.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    my_aiocb2.aio_sigevent.sigev_signo = SIGRTMAX-1;
    my_aiocb2.aio_sigevent.sigev_value.sival_ptr = &my_aiocb2;
    fd2_error = aio_read(&my_aiocb2);
    if (fd1_error == -1) {
        perror("The aio_read failed");
        exit(1);
    }

                          /* se procede a traslapar las operaciones */
    while(!fd1_error || !fd2_error)
        /* hacen lo que le parezca */ ;
    exit(0);
}
```

Programa 5.5

5.10 Ejercicio: Vaciado de estadísticas

La función `atexit` que aparece en la rutina `show_times` del programa 2.11 también trabaja como un manejador de señal para notificar el tiempo de CPU utilizado. Implante un manejador de señal para `SIGUSR1` que imprima esta información en el dispositivo de error estándar. Lea con cuidado las páginas del manual para determinar si todas las llamadas son seguras en relación con las señales asíncronas. Convierta el manejador en uno que emplee exclusivamente llamadas seguras con respecto de señales asíncronas. (Nota: En POSIX.1 o Spec 1170 no se garantiza que `fprintf` sea segura con respecto de señales asíncronas.)

5.11 Ejercicio: proc Filesystem II

Este ejercicio está basado en los ejercicios de la sección 3.12. Escriba un programa `examps` que tenga la página de manual que aparece en la figura 5.1. Implante `examps` de la siguiente manera:

- Escriba una primera versión de `examps` que analice la línea comando. Si se proporciona la opción `-s`, `examps` envía una señal `SIGSTOP` al proceso, imprime un mensaje en el dispositivo de salida estándar, y luego envía una señal `SIGCONT`. Si no se indica la opción `-s`, `examps` sólo imprime un mensaje en el dispositivo de salida estándar con el ID del proceso.
- Implante la función `get_prstatus` de la sección 3.12.
- Implante el programa `examps` completo descrito en la página de manual de la figura 5.1.

5.12 Ejercicio: Operación en línea de un dispositivo lento

Este ejercicio hace uso de la E/S asíncrona para traslapar el manejo de operaciones de E/S de un dispositivo lento con otros cálculos del programa. Un ejemplo de lo anterior es la impresión o la realización de una transferencia de archivo sobre un módem lento. Otro ejemplo es un programa que reproduce un archivo de audio en segundo plano mientras hace cualquier otra cosa. En los dos ejemplos, el programa lee un archivo en disco y lo escribe en un dispositivo lento.

Escriba un programa que haga uso de la E/S asíncrona para escribir en un dispositivo lento. La fuente de información es un archivo en disco. Utilice las señales de POSIX.1b con `aio_write` si es que éstas se encuentran disponibles; si no es el caso, entonces emplee la versión Spec 1170 con `SIGPOLL`. Pase el nombre del archivo de entrada y del dispositivo de salida como argumentos de línea comando. Pruebe esta fase con un archivo de entrada que es un archivo en disco y con el archivo de salida igual al dispositivo de error estándar. Si tiene disponible una estación de trabajo con soporte para dispositivo de audio, utilice un archivo de audio en disco como entrada y `"/dev/audio"` como dispositivo de salida.

El manejador de señal debe leer otro bloque de datos de un archivo de entrada y efectuar una operación `write` asíncrona en el archivo de salida. El manejador debe establecer una bandera externa en caso de que encuentre un error de E/S.

El programa principal abre los descriptores de entrada y salida y establece el manejador de señal apropiado. Si emplea el sistema Spec 1170, entonces haga uso de `ioctl` con el comando `I_SETSIG` para establecer el descriptor de archivo para operaciones de E/S asíncronas. El programa principal lleva a cabo un `read` del descriptor de entrada, seguido de la escritura en el descriptor de salida. A continuación el programa principal entra en un ciclo hasta que el manejador establece una bandera externa que indique la existencia de un error. Haga que el programa realice un cálculo dentro del ciclo. Este programa es particularmente interesante cuando la salida va al dispositivo de audio. Es posible escuchar plática de fondo o música traslapada con el cálculo. Estime el porcentaje de tiempo invertido en el manejo de la E/S comparado con el tiempo de cálculo.

```
NOMBRE
       examps - examina un proceso

SINOPSIS
       examps [-s] pid


DESCRIPCIÓN

La utilería examps examina el archivo /proc/pid y genera como salida la
siguiente información en el dispositivo de salida estándar:
       ID del proceso,
       ID del proceso padre,
       ID del grupo del proceso,
       ID de sesión,
       tiempo de CPU del usuario del proceso,
       tiempo de CPU del sistema del proceso,
       nombre de la clase de planificación,
       instrucción en ejecución,
       nombres de las señales pendientes,
       señal activa,
       inicio del heap,
       tamaño del heap del proceso en bytes,
       inicio de la pila,
       tamaño de la pila del proceso en bytes,
       número de llamada al sistema (si se tiene un syscall),
       número de parámetros a la llamada al sistema.

El proceso debe ser propiedad del usuario que ejecuta examps. Si se proporciona
la opción -s, el proceso se detiene y vuelve a comenzar. Si no se proporciona
esta opción, el proceso no se detiene y los valores tal vez sean modificados
mientras se lleva a cabo la lectura.

OPCIONES
     -s    Envía la señal SIGSTOP al proceso antes de obtener la información.
           Después envía la señal SIGCONT una vez obtenida la información.

EJEMPLOS
     El siguiente ejemplo examina el estado del proceso 3421:
          examps 3421
     El proceso 3421 no es detenido por examps. Tal vez sea detenido por
     otra fuente.

ARCHIVOS
     /proc

VÉASE TAMBIÉN
     proc(4)
```

Figura 5.1: Página de manual para `examps`.

5.13 Lecturas adicionales

En *Advanced Programming in the UNIX Environment* de Stevens [86] aparece un buen panorama histórico de las señales. Véase también *POSIX.4: Programming for the Real World* por Gallmeister [31] para una presentación completa del nuevo POSIX.1b Realtime Extension.

Capítulo 6

Proyecto: *Temporizadores*

Los sistemas operativos emplean temporizadores para tareas como la planificación de procesos, los tiempos de espera para protocolos de red y las actualizaciones periódicas de las estadísticas del sistema. Este capítulo estudia en detalle los temporizadores, pero también hace hincapié en la prueba y el manejo cuidadoso de las señales. El capítulo abarca tanto los temporizadores de Spec 1170 como los temporizadores y relojes de tiempo real de POSIX.1b, además de comparar las características de ambos.

Un *temporizador* mantiene un registro del paso del tiempo. Los temporizadores más sencillos miden el *tiempo transcurrido* y proporcionan esta información cuando es solicitada. Los *temporizadores de intervalo* generan una interrupción después de cierto intervalo de tiempo específico. Los sistemas operativos utilizan los temporizadores de intervalo de muchas maneras; éstos pueden generar una interrupción periódica, durante la cual el sistema operativo incrementa el contenido de un contador. Este contador puede mantener el tiempo transcurrido desde el arranque del sistema. Tradicionalmente, los sistemas UNIX mantienen la fecha del sistema como el número de segundos transcurridos desde el 1 de enero de 1970. Si el temporizador de intervalo genera una interrupción cada 100 microsegundos y reinicia cuando este tiempo termina, entonces la rutina de servicio de la interrupción puede mantener un contador local para medir el número de segundos transcurridos desde el 1 de enero de 1970, incrementando el contenido de este contador local de cada 10 000 ciclos del temporizador de intervalo.

Los sistemas operativos de tiempo compartido también emplean temporizadores de intervalo para la planificación de procesos. Cuando el sistema operativo planifica un proceso, da inicio a un temporizador de intervalo para un intervalo de tiempo denominado *quantum de planificación*. Si este temporizador expira y el proceso aún sigue en ejecución, el planificador moverá el proceso a una cola de procesos listos para la ejecución, de modo que pueda ejecutarse otro proceso. Los sistemas con varios procesadores necesitan uno de estos temporizadores de intervalo para cada procesador.

Muchos algoritmos de planificación tienen un mecanismo para incrementar la prioridad de los procesos que han estado en espera por mucho tiempo. El planificador puede utilizar un temporizador de intervalo para administrar la prioridad. Cada vez que el temporizador expira, el planificador aumenta la prioridad de los procesos que todavía no se ejecutan.

La biblioteca estándar de C contiene una función denominada `sleep`.

SINOPSIS

```
#include <unistd.h>

unsigned sleep(unsigned seconds);
```

POSIX.1, Spec 1170

Cuando un proceso ejecuta a `sleep`, ésta lo bloquea el número de segundos especificados en la llamada. Una implantación de `sleep` bien podría utilizar un temporizador de intervalo para cada proceso.

Por lo general, un sistema de cómputo tiene pocos temporizadores de intervalo implantados en *hardware*, y el sistema operativo implanta muchos temporizadores por *software* utilizando los temporizadores de *hardware*. En la sección 6.1 se estudian diversas representaciones de tiempo en UNIX, y en la sección 6.2 se examinan varios tipos de temporizadores de intervalo, incluyendo las nuevas características del temporizador que son parte de POSIX.1b Realtime Extension. Las siguientes secciones desarrollan el proyecto central del capítulo, que es la implantación de varios temporizadores de intervalo utilizando un temporizador de intervalo programable. Finalmente, la sección 6.9 explora aspectos más avanzados, como la deriva del temporizador (timer drift).

6.1 Tiempo en UNIX

La hora del sistema se mantiene como el número de segundos transcurridos desde la Época, la cual (está definida como) 00:00 (medianoche) del 1 de enero de 1970, Coordinated Universal Time (también denominado UCT, tiempo del meridiano de Greenwich o GMT). Cualquier programa puede tener acceso a la hora del sistema mediante la llamada a la función `time`.

SINOPSIS

```
#include <time.h>

time_t time(time_t *tloc);
```

POSIX.1, Spec 1170

La función `time` proporciona la hora en segundos transcurridos desde la Época. Si `tloc` no es `NULL`, la función `time` también guarda la hora en `*tloc`. Si se presenta un error, entonces `time` devuelve `-1` y pone un valor en `errno`.

Ejercicio 6.1

El tipo `time_t` por lo general se implanta como un `long`. Si un `long` es de 32 bits, ¿en qué fecha aproximadamente ocurrirá un desbordamiento (overflow) en `time_t`? Recuérdese que se emplea un bit para el signo. ¿Qué fecha causará un desbordamiento

si se emplea un `unsigned long`? ¿Qué fecha será la causa de un desbordamiento si el `long` es de 64 bits?

Respuesta:

Se necesitará 68 años a partir del 1 de enero de 1970 para llegar a la fecha que provoque un desbordamiento con un `long` de 32 bits, de modo que UNIX será seguro hasta el año 2038. Para un valor `time_t` que sea `unsigned long`, el desbordamiento ocurrirá en el año 2106. Para un `long` de 64 bits, UNIX será seguro aproximadamente hasta el año 292 mil millones, mucho después que el Sol se haya extinguido (`:-` ()).

El tipo `time_t` es conveniente para cálculos que requieren la diferencia entre fechas, pero es poco adecuado para la impresión de fechas. La función `ctime` de la biblioteca de C convierte una fecha en una cadena ASCII adecuada para su impresión.

SINOPSIS

```
#include <time.h>

char *ctime(const time_t *clock);
```

ISO C, POSIX.1, Spec 1170

La función `ctime` toma un parámetro, que es un apuntador a una variable de tipo `time_t` y devuelve un apuntador a una cadena de 26 caracteres. La función `ctime` toma en cuenta tanto el uso horario como las fechas de cambio de horario por ahorro. Cada uno de los campos en la cadena de caracteres tiene un ancho constante. La cadena puede verse como la siguiente:

```
Sun Oct 06 02:21:35 1986\n\0
```

La función `time` mide el tiempo *real* o la hora del reloj de pared. En un ambiente de multiprogramación existen muchos procesos que comparten la CPU, de modo que el tiempo real no es una medida exacta del tiempo de ejecución. El *tiempo virtual* de un proceso es la cantidad de tiempo que el proceso pasa en el estado de ejecución. Los tiempos de ejecución se expresan en tiempo virtual. La función `times` devuelve información sobre los tiempos de ejecución de un proceso y de sus hijos.

SINOPSIS

```
#include <sys/times.h>

clock_t times(struct tms *buffer);
```

POSIX.1, Spec 1170

El tipo `clock_t` guarda el número de impulsos (tics) del reloj. La estructura `struct tms` contiene al menos los siguientes miembros:

```
clock_t   tms_utime;  /* tiempo de CPU del usuario */
clock_t   tms_stime;  /* tiempo de CPU del sistema */
clock_t   tms_cutime  /* tiempo de CPU de un hijo terminado del usuario */
clock_t   tms_cstime; /* tiempo de CPU de un hijo terminado del sistema */
```

La función `times` devuelve el tiempo transcurrido en impulsos (tics) de reloj a partir de una fecha pasada cualquiera. La función `times` devuelve `-1` y pone un valor en `errno` si no tiene éxito. (El programa 2.11 muestra cómo utilizar `sysconf` para determinar el número de tics por segundo de un sistema.)

Ejemplo 6.1

El siguiente fragmento de código calcula la fracción de tiempo durante la que un proceso se ejecuta en el procesador mientras hace un cálculo.

```
#include <sys/times.h>
#include <limits.h>
#include <stdio.h>

clock_t real_start;
clock_t real_end;
clock_t ticks_used;
struct tms process_start;
struct tms process_end;

if ((real_start = times(&process_start)) == -1)
   perror("Could not get starting times");
else {
   /* perform calculation to be timed */
   if ((real_end = times(&process_end)) == -1)
      perror("Could not get ending times");
   else {
      ticks_used = process_end.tms_utime + process_end.tms_cstime -
         process_start.tms_utime - process_start.tms_cstime;
      printf("Fraction of time running = %f\n",
         (double)(ticks_used)/(real_end - real_start));
   }
}
```

Una escala de tiempo de segundos es demasiado gruesa para temporizar programas o controlar eventos de programas. Tradicionalmente los sistemas UNIX y el Spec 1170 utilizan la estructura `struct timeval` para expresar el tiempo en una escala más fina. La estructura `struct timeval` incluye los siguientes miembros:

```
long   tv_sec;   /* segundos desde Ene. 1, 1970 */
long   tv_usec;  /* y microsegundos */
```

La POSIX.1b Realtime Extension define un temporizador con una granularidad más fina con base en la estructura `struct timespec`, la cual tiene al menos los siguientes miembros:

```
time_t  tv_sec;  /* segundos */
long    tv_nsec; /* nanosegundos */
```

El miembro `tv_nsec` sólo es válido si es mayor que o igual con 0 y menor que 10^9. La estructura `struct timespec` especifica la hora tanto para relojes como temporizadores en POSIX.1b. Una implantación de UNIX que soporta estos relojes y temporizadores es aquella donde está definida `_POSIX_TIMERS`. Al momento de escribir el presente libro, en muchos sistemas aún no han sido implantados los temporizadores de POSIX.1b, así que en este capítulo se estudiará los temporizadores de Spec 1170 y POSIX.

Un programa puede aprovechar las facilidades ofrecidas por Spec 1170 para codificar la hora al recuperar ésta antes y después con la función `gettimeofday`.

SINOPSIS

```
#include <sys/time.h>

int gettimeofday(struct timeval *tp, void *tzp);
```

Spec 1170

La función `gettimeofday` llena con la hora la estructura `struct timeval` a la que apunta `tp`. Si la función tiene éxito devuelve 0, de lo contrario devuelve `-1` y pone un valor en `errno`. El apuntador `tzp` debe ser `NULL`. La inclusión de este segundo parámetro se hace por razones históricas.

La función `gettimeofday` no es parte de POSIX y existen muchas versiones de ella en uso, algunas de las cuales sólo toman un parámetro. El lector deberá consultar las páginas del manual antes de hacer uso de `gettimeofday`. El comité de POSIX no pudo decidir qué versión de `gettimeofday` adoptar, así que POSIX no define una función que devuelva la hora con una exactitud mayor que un segundo. La función `time` puede emplearse si esta exactitud es suficiente.

Ejemplo 6.2

El siguiente fragmento de código utiliza a `gettimeofday` *para medir el tiempo de ejecución de la función* `function_to_time`.

```
#include <stdio.h>
#include <sys/time.h>
#define MILLION 1000000

struct timeval tpstart;
struct timeval tpend;
long timedif;

gettimeofday(&tpstart,NULL);
function_to_time();            /* el código temporizado va aquí */
gettimeofday(&tpend,NULL);
timedif = MILLION*(tpend.tv_sec - tpstart.tv_sec) +
                  tpend.tv_usec - tpstart.tv_usec;
fprintf(stderr, "It took %ld microseconds\n", timedif);
```

El POSIX.1b Realtime Extension también contiene relojes. Un *reloj* es un contador que se incrementa a intervalos fijos conocidos como *resolución del reloj*. POSIX.1b proporciona una función para poner la hora (`clock_settime`), recuperarla (`clock_gettime`) y determinar la resolución del reloj (`clock_getres`).

SINOPSIS

```
#include <time.h>

int clock_settime(clockid_t clock_id,
                  const struct timespec *tp);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_getres(clockid_t clock_id, struct timespec *res);
```

POSIX.1b

Estas funciones devuelven 0 si tienen éxito, de lo contrario devuelven `-1` y ponen un valor en `errno`. Los relojes de POSIX.1b pueden ser visibles en todo el sistema o sólo dentro de un proceso. Todas las implantaciones deben dar soporte a un reloj que abarca todo el sistema, `CLOCK_REALTIME`, el cual corresponde al reloj de tiempo real de sistema. Sólo los usuarios privilegiados pueden poner la hora de este reloj, pero cualquier usuario puede leerla.

Ejemplo 6.3

El siguiente segmento de código mide el tiempo de ejecución de `function_to_time` *empleando los relojes de POSIX.1b.*

```
#include <stdio.h>
#include <time.h>
#define MILLION 1000000

struct timespec tpstart;
struct timespec tpend;
long timedif;

clock_gettime(CLOCK_REALTIME, &tpstart);
function_to_time();            /* el código temporizado va aquí */
clock_gettime(CLOCK_REALTIME, &tpend);
timedif = MILLION*(tpend.tv_sec - tpstart.tv_sec) +
          (tpend.tv_nsec - tpstart.tv_nsec)/1000;
fprintf(stderr,"It took %ld microseconds\n", timedif);
```

6.2 Temporizadores de intervalo

Los temporizadores de intervalo disponibles en Spec 1170 y POSIX.1b son similares, pero difieren en ciertos aspectos significativos. Los temporizadores de Spec 1170 están basados en los tradicionales temporizadores de UNIX y se encuentran disponibles en muchos sistemas. Los temporizadores de intervalo de POSIX.1b proporcionan una exactitud y flexibilidad mayores.

6.2.1 Temporizadores de intervalo en Spec 1170

Muchos sistemas operativos asignan varios temporizadores de usuario a cada proceso. Una implantación que cumpla con Spec 1170 debe proporcionar a cada proceso los tres siguientes temporizadores de intervalo:

`ITIMER_REAL:`	decrece en tiempo real y genera una señal `SIGALRM` cuando termina.
`ITIMER_VIRTUAL:`	decrece en tiempo virtual (tiempo utilizado por el proceso) y genera una señal `SIGVTALRM` cuando termina.
`ITIMER_PROF:`	decrece en tiempo virtual y en tiempo del sistema para el proceso, y genera una señal `SIGPROF` cuando termina.

Los temporizadores de intervalo de Spec 1170 utilizan una estructura `struct itimerval` que contiene los siguientes miembros:

```
struct timeval it_value;    /* tiempo hasta la próxima expiración */
struct timeval it_interval; /* valor que debe volverse a poner
                               en el temporizador */
```

En este caso, `it_value` guarda el tiempo que resta antes de que el temporizador expire, e `it_interval` guarda el intervalo de tiempo que será utilizado para reiniciar el temporizador una vez que éste expire.

Spec 1170 proporciona la función `setitimer` para iniciar y detener los temporizadores de intervalo del usuario.

SINOPSIS

```
#include <sys/time.h>

int setitimer (int which, const struct itimerval *value,
                          struct itimerval *ovalue);
```

Spec 1170

La función `setitimer` devuelve 0 si tiene éxito, de lo contrario devuelve `-1` y pone un valor en `errno`. El parámetro `which` especifica el temporizador (esto es, `ITIMER_REAL`, `ITIMER_VIRTUAL`, o `ITIMER_PROF`). El programa que invoca esta función especifica el intervalo de tiempo para configurar al temporizador en `*value`. Aunque la función invocante pasa un apuntador a la estructura `struct itimerval`, también debe proporcionar los valores de los miembros de ésta. La función `setitimer` no cambia a `*value`. Si el miembro `it_interval` de `*value` no es 0, el temporizador reiniciará con este valor cuando termine. Si el miembro `it_interval` de `*value` es 0, el temporizador no volverá a reiniciar una vez que termine. Si el valor del miembro `it_value` de `*value` es 0, `setitimer` detendrá al temporizador si éste se encuentra en ejecución.

La función `setitimer` llena los miembros de la estructura a la que apunta `ovalue` con los valores de la hora. Si el temporizador ya estaba en ejecución, el miembro `it_value` de `*ovalue` es distinto de cero y contiene el tiempo que queda antes de que el temporizador

expire. El apuntador `ovalue` puede ser `NULL`, en cuyo caso no se devuelve ninguna información.

La función `setup_interval_timer` del programa 6.1 hace que el proceso imprima un asterisco cada dos segundos de tiempo de la CPU utilizado. El temporizador `ITIMER_PROF` genera una señal `SIGPROF` por cada dos segundos de tiempo de la CPU utilizado por un proceso. El proceso atrapa la señal `SIGPROF` y la maneja con `myhandler`.

Programa 6.1: Programa que imprime un asterisco por cada dos segundos de tiempo de CPU utilizado.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>

char astbuf[] = "*";
static void myhandler(int s)
{
   write(STDERR_FILENO, astbuf, sizeof(char));
}

/* configuración del manejador myhandler para la señal SIGPROF */
void init_timer_interrupt(void)
{
   struct sigaction newact;
   newact.sa_handler = myhandler;
   newact.sa_flags = SA_RESTART;
   sigemptyset(&newact.sa_mask);
   sigaction(SIGPROF, &newact, NULL);
}
/*configuración del temporizador de intervalo ITIMER_PROF para intervalos
de 2 segundos*/
void setup_interval_timer(void)
{
   struct itimerval value;
   value.it_interval.tv_sec = 2;
   value.it_interval.tv_usec = 0;
   value.it_value = value.it_interval;
   setitimer(ITIMER_PROF, &value, NULL);
}

void main(int argc, char *argv[])
{
   init_timer_interrupt();
   setup_interval_timer();
   /* la ejecución del resto del programa va aquí */
   exit(0);
}
```

Programa 6.1

Ejercicio 6.2

Escriba un programa que configure el temporizador `ITIMER_REAL` para que expire en dos segundos y luego duerma por otros diez. ¿Cuánto tiempo le llevará al programa terminar? ¿Por qué?

Respuesta:

El programa terminará en dos segundos debido a que la acción predeterminada de la señal `SIGALRM` es terminar el proceso.

Ejercicio 6.3

Modifique el programa del ejercicio 6.2 para atrapar la señal `SIGALRM` antes de que éste configure el temporizador. El manejador de señal deberá imprimir un mensaje de error en el dispositivo de error estándar y regresar. Ahora, ¿cuánto tiempo le llevará al programa terminar su ejecución? Reemplace el `sleep` por un lazo infinito. ¿Qué sucede?

Respuesta:

Si el programa hace uso de `sleep`, éste imprime un mensaje y termina en dos segundos debido a que `sleep` termina de manera prematura cuando el proceso recibe la señal `SIGALRM`. Si el programa emplea un lazo infinito, entonces nunca termina.

Para determinar la cantidad de tiempo que resta en un temporizador Spec 1170 utilice la función `getitimer`.

SINOPSIS

```
#include <sys/time.h>

int getitimer (int which, struct itimerval *value);
```

Spec 1170

`getitimer` pone en la estructura `*value` el tiempo que queda hasta que el temporizador `which` expire. La función `getitimer` devuelve 0 si tiene éxito, de lo contrario devuelve `-1` y pone un valor en `errno`.

Ejercicio 6.4

¿Qué error hay en el siguiente fragmento de código, el cual debería imprimir el número de segundos que quedan en el temporizador de intervalo `ITIMER_VIRTUAL`?

```
#include <sys/time.h>
#include <stdio.h>
struct itimerval *value;

getitimer(ITIMER_VIRTUAL, value);
fprintf(stderr, "Time left is %ld seconds\n",
   value->it_value.tv_sec);
```

Respuesta:

La variable `value` no está iniciada y está declarada como un apuntador a una estructura `struct itimerval`, pero no apunta a nada. Esto es, no existe ninguna declaración de la estructura `struct itimerval` a la que pueda apuntar `value`.

El programa 6.2 utiliza el temporizador de intervalo `ITIMER_VIRTUAL` para medir el tiempo de ejecución de la función `function_to_time`. Este ejemplo, a diferencia del ejemplo 6.2, utiliza el tiempo virtual. Recuérdese que el valor devuelto por `getitimer` es el tiempo que queda, de modo que éste es una cantidad decreciente.

Programa 6.2: Fragmento de programa que hace uso de un temporizador de intervalo Spec 1170 para medir el tiempo de ejecución de una función.

```
#include <stdio.h>
#include <sys/time.h>
#define MILLION 1000000

   struct itimerval value;
   struct itimerval ovalue;
   long timedif;

   value.it_interval.tv_sec = 0;
   value.it_interval.tv_usec = 0;
   value.it_value.tv_sec = MILLION;      /* un número grande */
   value.it_value.tv_usec = 0;
   setitimer(ITIMER_VIRTUAL, &value, NULL);
   getitimer(ITIMER_VIRTUAL, &ovalue);
   function_to_time();                   /* el código temporizado va aquí */
   getitimer(ITIMER_VIRTUAL, &value);
   timedif = MILLION*(ovalue.it_value.tv_sec - value.it_value.tv_sec) +
             ovalue.it_value.tv_usec - value.it_value.tv_usec;
   printf("It took %ld microseconds\n", timedif);
```

Programa 6.2

6.2.2 Temporizadores de intervalo en POSIX

En Spec 1170 cada proceso tiene un número pequeño de temporizadores de intervalo, uno de cada tipo: `ITIMER_REAL`, `ITIMER_VIRTUAL`, `ITIMER_PROF`, y así sucesivamente. En POSIX.1b existe un número pequeño de relojes tales como `CLOCK_REALTIME`, y los procesos pueden crear muchos temporizadores independientes para cada reloj.

Los temporizadores de POSIX.1b están basados en la estructura `struct itimerspec`, la cual tiene definidos los siguientes miembros:

```
struct timespec  it_interval;  /* periodo del temporizador */
struct timespec  it_value;     /* expiración del temporizador */
```

Al igual que con los temporizadores de Spec 1170, `it_interval` es el tiempo empleado para reinicializar el temporizador una vez que éste expira. El miembro `it_value` guarda el tiempo que queda para que el temporizador expire.

Un proceso puede crear temporizadores específicos llamando a la función `timer_create`. Los temporizadores son temporizadores por proceso y no son heredados por las llamadas a `fork`.

SINOPSIS

```
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clock_id, struct sigevent *evp,
      timer_t *timerid);

struct sigevent {
      int             sigev_notify    /* tipo de notificación */
      int             sigev_signo;    /* número de señal */
      union sigval    sigev_value;    /* valor de la señal */
};

union sigval {
      int             sival_int;      /* valor entero */
      void            *sival_ptr;     /* apuntador */  */
};
```

POSIX.1b

`clock_id` especifica el reloj en que se basará el temporizador, y `*timerid` guarda el ID del temporizador creado. `timer_create` devuelve 0 si tiene éxito, de lo contrario devuelve `-1` y pone un valor en `errno`.

Los miembros de la estructura `sigevent` y la unión `sigval` mostrados en la sinopsis son necesarios en el estándar POSIX.1b. El estándar no prohíbe que una implantación incluya miembros adicionales.

El parámetro `*evp` de `timer_create` especifica la señal que debe enviarse al proceso cuando el temporizador expire. Si `evp` es `NULL`, el temporizador genera la señal predeterminada cuando expire. Para `CLOCK_REALTIME`, la señal predeterminada es `SIGALRM`. Para que se genere una señal diferente de la señal predeterminada al expirar el temporizador, el programa debe poner en `evp->sigev_signo` el número de la señal deseada. El miembro `evp->sigev_notify` de la estructura `struct sigevent` indica la acción que debe emprenderse cuando el temporizador expire. Normalmente este miembro es `SIGEV_SIGNAL`, lo que indica que la expiración del temporizador genera una señal. El programa puede impedir que la expiración del temporizador genere una señal haciendo que el miembro de `evp->sigev_notify` sea `SIGEV_NONE`.

Si existen varios temporizadores que generan la misma señal, el manejador puede hacer uso de `evp->sigev_value` para distinguir el temporizador que generó la señal. Para hacer esto, el programa debe utilizar la bandera `SA_SIGINFO` en el miembro `sa_flags` de `struct sigaction` cuando instale el manejador para la señal. (Véase el programa 6.7 de la página 243 para observar un ejemplo de cómo hacer esto.)

Las tres siguientes funciones manipulan los temporizadores por proceso de POSIX.1b.

SINOPSIS

```
#include <time.h>

int timer_settime(timer_t timerid, int flags,
     const struct itimerspec *value,
     struct itimerspec *ovalue);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_getoverrun(timer_t timerid);
```

POSIX.1b

La función `timer_settime` arranca o detiene un temporizador creado con una llamada a `timer_create`. El parámetro `flags` indica si el temporizador emplea tiempo absoluto o relativo. El tiempo relativo es similar al de los temporizadores Spec 1170, mientras que el tiempo absoluto permite una exactitud y control mayores de la deriva del temporizador. Este aspecto será estudiado con mayor detalle en la sección 6.9. Los dos últimos parámetros tienen el mismo significado que en `setitimer`. Las funciones `timer_settime` y `timer_gettime` devuelven 0 si tienen éxito, mientras que `timer_getoverrun` regresa el número de veces que el temporizador queda estancado. Si estas funciones fallan entonces devuelven `-1` y ponen un valor en `errno`.

Para obtener el tiempo que queda en un temporizador activo, utilice `timer_gettime` de manera similar a `getitimer`. Es posible que un temporizador expire mientras se encuentra pendiente una señal proveniente de una expiración previa del mismo temporizador. En este caso es posible perder una de las señales. Esta situación se conoce como estancamiento del temporizador. Un programa puede determinar el número de tales estancamientos para un temporizador en particular mediante una llamada a la función `timer_getoverrun`. Los estancamientos sólo se presentan para señales generadas por el mismo temporizador. Las señales generadas por muchos temporizadores, incluso de aquellos que emplean el mismo reloj y señal, se ponen en una cola y no se pierden.

El programa 6.3 crea un temporizador de POSIX.1b para medir el tiempo de ejecución de la función `function_to_time`. Este programa es muy similar al 6.2, pero hace uso de tiempo real en lugar de tiempo virtual.

Programa 6.3: Segmento de programa que hace uso de un temporizador de intervalo POSIX.1b para medir el tiempo de ejecución de una función.

```
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>
#define MILLION 1000000
#define THOUSAND 1000
```

```
timer_t time_ID;
struct itimerspec value;
struct itimerspec ovalue;
long timedif;

if (timer_create(CLOCK_REALTIME, NULL, &time_ID) < 0) {
   perror("Could not create a timer based on CLOCK_REALTIME");
   exit(1);
}
value.it_interval.tv_sec = 0;
value.it_interval.tv_nsec = 0;
value.it_value.tv_sec = MILLION;      /* un número grande */
value.it_value.tv_nsec = 0;
timer_settime(time_ID, 0, &value, NULL);
timer_gettime(time_ID, &ovalue);
function_to_time();                   /* el código temporizado va aquí */
timer_gettime(time_ID, &value);
timedif = MILLION*(ovalue.it_value.tv_sec - value.it_value.tv_sec) +
   (ovalue.it_value.tv_nsec - value.it_value.tv_nsec)/THOUSAND;
printf("It took %ld microseconds\n", timedif);
```

Programa 6.3

La sección 6.9 contiene información adicional sobre temporizadores POSIX.1b. El proyecto de temporización está descrito en términos de temporizadores Spec 1170 debido a que los temporizadores POSIX.1b aún no están disponibles en muchas implantaciones.

6.3 Panorama del proyecto

El proyecto de este capítulo desarrolla una implantación de varios temporizadores en términos del temporizador del sistema operativo. El proyecto está formado por cinco módulos semindependientes. Tres de ellos se crean como objetos; los otros dos son programas principales.

La figura 6.1 muestra los cinco módulos y las relaciones que existen entre ellos. La línea punteada muestra cómo ocurre la comunicación a través de un entubamiento. La salida estándar del programa `testtime` se envía a la entrada estándar del programa `timermain`. Las líneas (sólidas) representan la llamada de funciones en los objetos. El programa `timermain` llama sólo a las funciones del objeto `mytimers`. El objeto `mytimers` llama a las funciones de los objetos `hardware_timer` y `showall`. El objeto `hardware_timer` llama a las funciones del objeto `showall`. El objeto `showall`, cuyo fin sólo es depurar, llama a funciones de los objetos `mytimers` y `hardware_timer`.

En el nivel más bajo se encuentra el objeto `hardware_timer`. Este objeto consta de un solo temporizador del sistema operativo, el cual genera una señal cuando expira. El objeto temporizador de interés puede ser ya sea un temporizador Spec 1170 o bien, POSIX.1b. Aunque éste no es un verdadero temporizador por *hardware*, será tomado como tal. El objeto proporciona funciones de interfaz que ocultan a la vista de los usuarios externos el temporizador

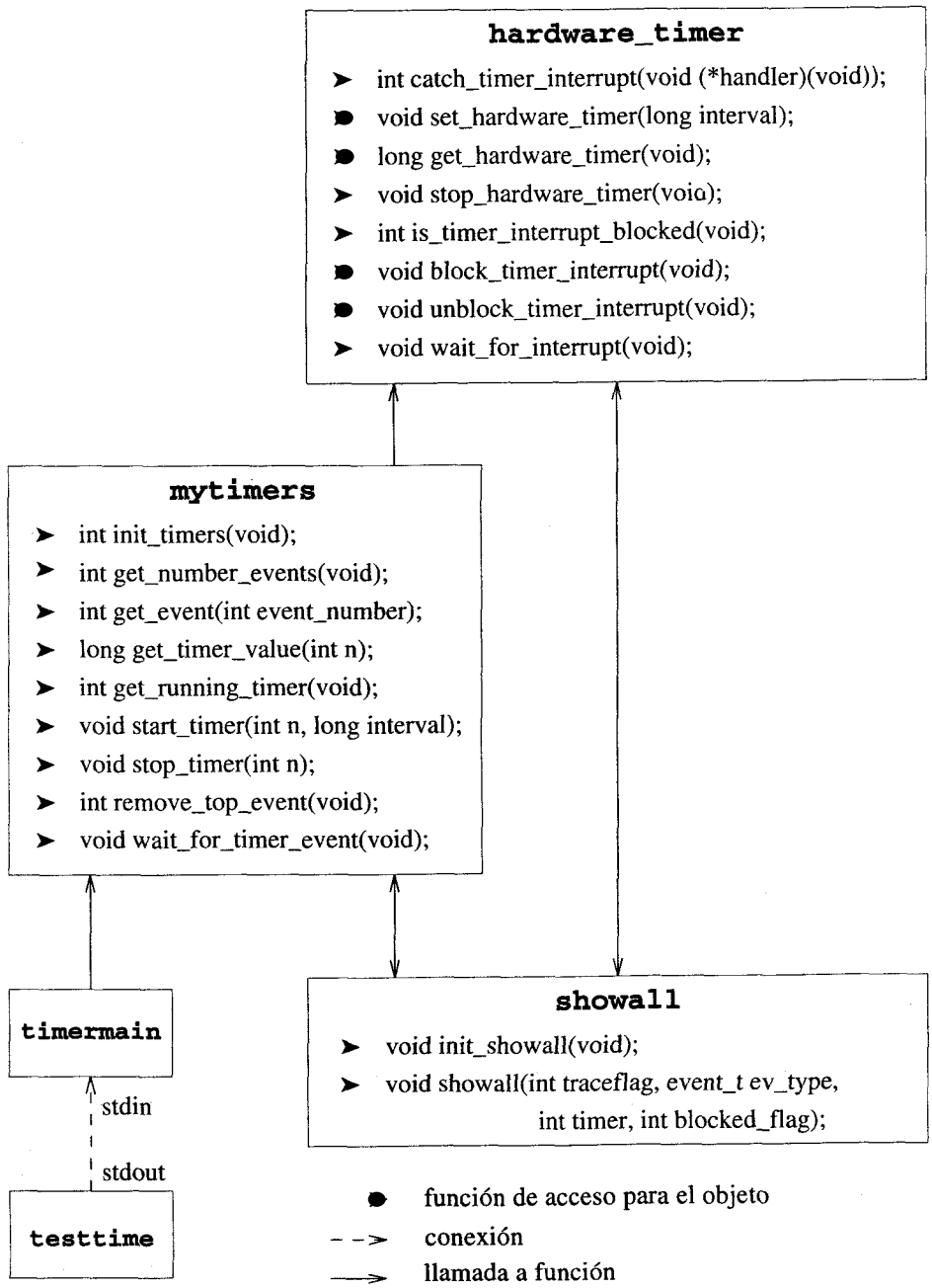


Figura 6.1: Módulos temporizadores que serán creados en este proyecto.

del sistema, y en este sentido se comporta como un temporizador por *hardware*. En teoría, si el programa tiene acceso a un temporizador por *hardware* real, el objeto puede ser este temporizador y la interfaz seguirá siendo la misma. Las funciones de interfaz manipulan un solo temporizador que genera una señal cuando éste expira.

En el siguiente nivel se encuentra el objeto de temporizadores múltiples, `mytimers`. Este objeto tiene funciones de interfaz que manipulan los temporizadores llamando a las funciones del módulo `hardware_timer`.

El tercer objeto, `showall`, sirve para depurar y es llamado por las funciones de los otros dos objetos para visualizar una bitácora de operaciones del temporizador. También llama a las funciones de los otros módulos para obtener información sobre el estado de los temporizadores.

Cada uno de los objetos tiene un archivo de encabezado con el mismo nombre y una extensión `.h` que contiene los prototipos de las funciones a las que puede tenerse acceso desde el exterior. El objeto y cualquier módulo que haga uso de él debe incluir este archivo. Cada objeto tiene también un archivo de encabezado local que contiene definiciones y prototipos de los ítems que serán empleados internamente por el objeto, pero que quedarán ocultos para el exterior.

Para probar los objetos de temporizador se hace uso de dos programas principales. El primero de ellos, `timermain`, recibe la entrada del dispositivo de entrada estándar para llamar a las funciones del objeto `mytimers`. Podría, por ejemplo, poner en marcha un temporizador para que expire después de un intervalo dado. El temporizador arranca cuando `timermain` recibe esta entrada.

Para el proceso de depuración es muy importante que los experimentos produzcan resultados incorrectos que puedan reproducirse con precisión. Entonces, cuando se encuentra un error, el programador puede corregir el código y repetir el mismo experimento con el código modificado. Si los experimentos dependen de la temporización de la entrada del teclado, entonces es casi imposible repetirlos. Para resolver este problema, el programa `testtime` pone en entubamientos los datos para `timertime`, a intervalos muy precisos. El programa `testtime` lee líneas provenientes del dispositivo de entrada estándar e interpreta el primer entero de la línea como un tiempo de retraso. A continuación `testtime` lee la siguiente línea de entrada y continúa su ejecución. Esta configuración permite que `testtime` lea la entrada de un archivo y simule la entrada desde el teclado para `timermain` con un tiempo preciso.

Este proyecto se implanta por etapas. La sección 6.4 introduce las estructuras de datos y proporciona ejemplos sobre la configuración de un solo temporizador. La sección 6.5 presenta los tres objetos y especifica cómo manejar la configuración de un solo temporizador. La sección 6.7 analiza algunas de las condiciones de contención que pueden presentarse con varios temporizadores y la forma de evitarlas, mientras que en la sección 6.8 se estudia una aplicación sencilla del temporizador. La sección 6.9 presenta aspectos avanzados del temporizador de acuerdo con los lineamientos de POSIX y la forma en que éste puede implantarlo.

6.4 Temporizadores sencillos

A menudo, los sistemas operativos implantan varios temporizadores por *software* con base en un solo temporizador por *hardware*. El temporizador por *software* puede representarse con un

número de temporizador y una indicación del momento en que éste expira. La implantación depende del tipo de temporizador de *hardware* disponible.

Supóngase que el temporizador por *hardware* genera interrupciones en intervalos de tiempo cortos y regulares. El intervalo de tiempo por lo general se conoce como el tiempo normal *tic* del reloj (cloc tic time). La rutina de servicio de interrupción del temporizador mantiene un registro del tiempo que queda en cada temporizador (en términos de *tics* del reloj) y disminuye este tiempo para cada *tic* del reloj. Cuando este registro llega a 0, el programa emprende la acción apropiada. Este enfoque es ineficiente si el número de temporizadores es grande o si el tiempo que corresponde al *tic* del reloj es pequeño.

Como alternativa, el programa puede mantener la información del temporizador en una lista ordenada de acuerdo con el tiempo de expiración. Cada entrada contiene un número de temporizador y el tiempo de expiración. La primera entrada en la lista contiene el primer temporizador que va a expirar y el tiempo hasta el momento de la expiración (en *tics* de reloj). La segunda entrada contiene el siguiente temporizador que expirará y el tiempo de expiración relativo al tiempo del primer temporizador que ha expirado, y así sucesivamente. Con esta representación, la rutina de servicio de interrupción disminuye sólo un contador con cada *tic* del reloj, pero existe un costo adicional implícito cuando el programa pone en marcha un temporizador, y es que éste debe insertar en una lista de temporizadores ordenada el tiempo del temporizador que expira inmediatamente después del nuevo.

Ejercicio 6.5

En cada uno de los dos casos descritos anteriormente, ¿cuál es la complejidad en el tiempo del manejador de interrupciones y de la función de arranque del temporizador en términos del número de temporizadores?

Respuesta:

Supóngase que existen n temporizadores. Para el primer método, el manejador de interrupciones es de orden $O(n)$ puesto que es necesario decrementar el valor de todos los temporizadores. La función de arranque del temporizador es $O(1)$ puesto que la puesta en marcha de un temporizador puede hacerse de manera independiente de los demás. Para el segundo método, el manejador de interrupciones por lo general es de orden $O(1)$ puesto que sólo es necesario decrementar el valor del primer temporizador. Sin embargo, cuando esto hace que el temporizador expire, es necesario examinar la siguiente entrada para asegurarse de que no expire al mismo tiempo. Esto puede degenerar en un orden $O(n)$ en el peor de los casos, pero en la práctica el peor de los casos es una situación poco probable. La función de puesta en marcha del temporizador es de orden $O(n)$ en lo que respecta a insertar el temporizador en la lista ordenada o menos si se emplea una estructura de datos más compleja, como un *heap*.

Si el sistema tiene un temporizador de intervalo por *hardware* en lugar de un reloj simple, entonces el programa puede configurar el temporizador de intervalo para que expire en un tiempo correspondiente al temporizador por *software* cuya fecha de expiración sea la más próxima. En este caso no existe costo alguno a menos que un temporizador expire, otro arranque o uno se detenga. Los temporizadores de intervalo son ineficientes cuando los intervalos son largos.

Ejercicio 6.6

Analice el manejador de interrupción y la función de arranque del temporizador para un temporizador de intervalo.

Respuesta:

La complejidad depende de la forma en que se mantengan los temporizadores, en una lista ordenada o no. El manejador del interruptor tiene el mismo orden que el temporizador del *tic* del reloj. La función de arranque del temporizador es de orden *O* (*n*) si el temporizador se mantiene en una lista ordenada, debido a que tal vez sea necesario tener acceso a todas las entradas en la estructura que define al temporizador.

El proyecto utiliza un temporizador de intervalo para implantar varios temporizadores, reemplazando el temporizador de *hardware* por el `ITIMER_REAL`. Cuando `ITIMER_REAL` expira, genera una señal `SIGALRM` y el manejador de ésta coloca una entrada en la lista de eventos.

La figura 6.2 muestra una implantación sencilla de cinco temporizadores por *software* (designados del `0` al `4`), los cuales están representados por variables de tipo `long` en el arreglo `active`. En el arreglo se emplea una entrada `-1` para indicar un temporizador que no se encuentra activo. El arreglo `event` mantiene una lista de temporizadores que han expirado, y `num_events` guarda el número de eventos sin manejar.

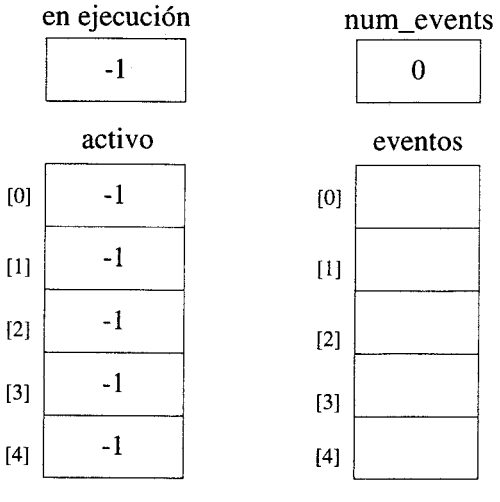


Figura 6.2: Estructura de datos `timers` donde no hay temporizadores activos.

La puesta en marcha de un temporizador se hace especificando un número de temporizador y un intervalo en microsegundos. La figura 6.3 muestra la estructura de datos después de la puesta en marcha del temporizador `2` por cinco segundos (5,000,000 microsegundos). Ningún temporizador ha expirado aún, así que la lista de eventos está vacía.

La sola escritura de información en el arreglo `active` de la figura 6.2 no es suficiente para implantar un temporizador. El programa debe configurar el temporizador `ITIMER_REAL`

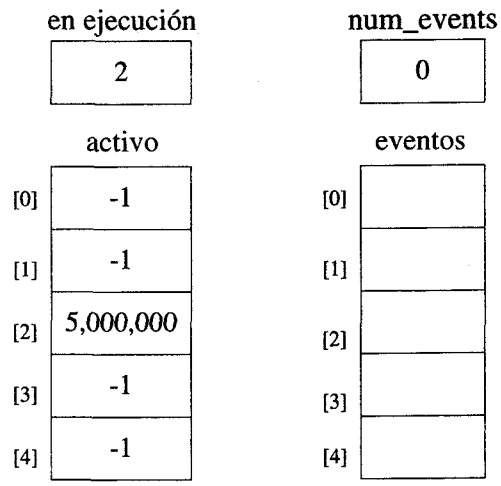


Figura 6.3: Estructura de datos `timers` después de haber configurado el temporizador `2` con un tiempo de cinco segundos.

para 5,000,000 de microsegundos. Cuando se deposita `SIGALRM`, el programa debe borrar la entrada del arreglo `active` y marcar una entrada en el arreglo `event`. La figura 6.4 muestra la estructura de datos del temporizador después de que haya expirado `ITIMER_REAL`.

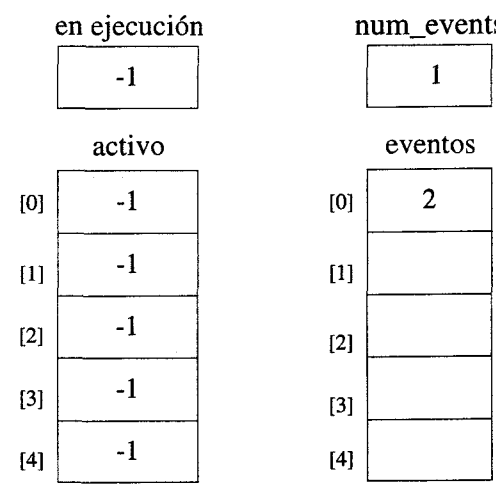


Figura 6.4: Estructura de datos `timers` después de la expiración del temporizador `2`.

6.5 Configuración de uno de cinco temporizadores

Esta sección describe una implantación para configurar uno de cinco posibles temporizadores por *software* utilizando el temporizador de intervalo subyacente del proceso `ITIMER_REAL`. El programa principal toma el número de temporizador y el intervalo de temporización (en microsegundos) como argumentos de línea comando y luego llama a la función `start_timer`. A continuación, el programa principal espera a que el temporizador expire, luego imprime el mensaje de que el temporizador ha expirado y termina su ejecución.

6.5.1 Objeto `mytimers`

Los temporizadores por *software* se implantan en un objeto denominado `mytimers`. Para ello se utiliza la variable estática `timers` de tipo `timer_data_t` para guardar el dato del temporizador interno para el objeto. La estructura `timer_data_t` es

```
#define MAXTIMERS 5
typedef struct timer_data_t {
   int running;
   long active[MAXTIMERS];
   int num_events;
   int events[MAXTIMERS];
} timer_data_t;
```

Los nombres de los miembros de la estructura `timer_data_t` significan lo siguiente:

`ejecutar`	es el número de temporizador que está bajo ejecución o `-1` si ninguno se encuentra activo. El temporizador en ejecución es aquel que será el siguiente en expirar. Es el que su expiración de tiempo origina que el temporizador real (indicado por la variable `set_hardware_timer` especificada en la página 226) genere una señal.
`active`	tiene una entrada para cada temporizador que indica el tiempo de expiración (en μs) en relación con el tiempo de inicio del temporizador en ejecución. Un valor negativo indica que el temporizador no está activo. (En esta parte sólo se encontrará activo un temporizador.)
`num_events`	es el número de entradas en el arreglo `events`.
`events`	es una lista de los temporizadores que han expirado en el orden en que lo han hecho. (En esta sección existe cuando mucho un temporizador en la lista.)

La representación con enteros de los intervalos de tiempo simplifica el código, pero limita la longitud de los intervalos aproximadamente a 2000 segundos (un poco más de media hora) para enteros de 32 bits. Esto debe ser más que suficiente para probar los algoritmos.

Ponga la estructura de datos `timers` en `mytimers.c` junto con las siguientes funciones, las cuales deben ser llamadas desde fuera del objeto:

- `init_timers` que inicia la estructura de datos `timers` de la figura 6.2. La función también llama a las funciones de iniciación de los objetos `hardware_timer` y `showall`. La función `init_timers` devuelve 0 si tiene éxito, de lo contrario regresa `-1`. El prototipo es

```
int init_timers(void);
```

- `get_number_events` devuelve el valor de `num_events`. Su prototipo es

```
int get_number_events(void);
```

- `get_event` devuelve el número de temporizador asociado con una entrada particular del arreglo `events`. El prototipo es

```
int get_event(int event_number);
```

  El parámetro `event_number` especifica la posición en el arreglo `events`, el cual está indexado desde 0. La función `get_event` devuelve `-1` si `event_number` es negativo, mayor o igual que `num_events`.

- `get_timer_value` devuelve el valor en uso de un temporizador específico a partir del arreglo active o `-1` si ningún temporizador está en ejecución o si el número es no válido. El prototipo es

```
long get_timer_value(int n);
```

  El parámetro `n` es el número del temporizador.

- `get_running_timer` devuelve el número del temporizador que está ejecutándose o `-1` si no hay un temporizador en ejecución. El prototipo es

```
int get_running_timer(void);
```

- `start_timer` pone en marcha un temporizador con el intervalo de tiempo dado en microsegundos. Para esta parte suponga que no hay otros temporizadores o incluso que ninguno está en ejecución. El prototipo es

```
void start_timer(int n, long interval);
```

  El parámetro `n` indica qué temporizador debe ponerse en marcha, e `interval` es el número de microsegundos después de los cuales el temporizador debe expirar. Para poner en marcha el temporizador `n`

  * Elimínese el temporizador `n` de la lista de eventos si es que se encuentra en ella.
  * Ejecútese igual a `n`.
  * Póngase en `active[n]` el valor de tiempo apropiado.
  * Póngase en marcha el temporizador de intervalo llamando a la función `set_hardware_timer` del objeto `hardware_timer`.

- `stop_timer` detiene un temporizador si éste se encuentra en ejecución y lo elimina de la lista de eventos si es que se encuentra en ella. El prototipo es

```
void stop_timer(int n);
```

Esta función se necesitará más adelante cuando se manejen varios temporizadores.

- `remove_top_event` elimina el evento del tope de la lista de eventos y devuelve el número de temporizador o -1 si la lista está vacía. El prototipo es

```
int remove_top_event(void);
```

Esta función se necesitará más adelante cuando se manejen varios temporizadores.

- `wait_for_timer_event` espera hasta que exista un evento en la lista de eventos y luego regresa sin modificar la lista. El prototipo es

```
void wait_for_timer_event(void);
```

El objeto `mytimers` también contiene las siguientes funciones, a las cuales no se puede tener acceso desde el exterior. Entre ellas se incluye

- `myhandler` que maneja la señal del temporizador. Esta función la llama el manejador de señal en uso para mantener la estructura `timers` cuando el temporizador de *hardware* real expira. La función debe hacer lo siguiente:

  * Añadir el temporizador en ejecución a la lista de eventos.
  * Desactivar el temporizador en ejecución.
  * Actualizar la estructura de datos `timers`.
  * Reiniciar el temporizador de intervalo si existe un temporizador activo. (Esto no sucede en el caso de un solo temporizador.)

  El prototipo es

```
static void myhandler(void);
```

- `put_on_event_list_and_deactivate` desactiva un temporizador, lo retira de la lista de eventos si se encuentra en ella y lo pone al final de esta lista. El prototipo es

```
static void put_on_event_list_and_deactivate(int n);
```

  El parámetro `n` es el número del temporizador por desactivar. La función `put_on_event_list_and_deactivate` no emprende ninguna acción si `n` es negativo, mayor o igual que `MAXTIMERS`.

Puesto que el objeto `hardware_timer` maneja las señales, debe contener al manejador de señal en uso. El prototipo del manejador de señal puede depender de la implantación y no debe ser parte del objeto `mytimers`. Puesto que es necesario manipular los temporizadores cuando se atrapa una señal, este trabajo deberá hacerse en el objeto `mytimers`. El manejador de señal utilizado llama a `myhandler` para hacer esto. Puesto que `myhandler` se declara para tener vinculación interna, éste se pasa a la función `catch_timer_interrupt` del objeto `hardware_timer`.

6.5.2 Objeto `hardware_timer`

El objeto `hardware_timer` contiene el código para manejar un solo temporizador de "*hardware*". Las funciones a las que se tiene acceso desde fuera del objeto son

- `catch_timer_interrupt`, que hace uso de `sigaction` a fin de establecer un manejador de señal para atrapar la señal `SIGALRM`. La función devuelve 0 si tiene éxito o `-1` si hay un error. El prototipo es

  ```
  int catch_timer_interrupt(void (*handler) (void));
  ```

  El parámetro `handler` es el nombre de la función que hará el trabajo de manejar la señal. El manejador de señal en uso en `hardware_timer` sólo llamará a la función `handler`. El objeto `mytimers` llama a la función `catch_timer_interrupt` para establecer el manejo de la señal.

- `set_hardware_timer` pone en marcha el temporizador `ITIMER_REAL` con el intervalo dado en microsegundos. La llamada a `set_hardware_timer` deberá hacerse únicamente cuando la interrupción del temporizador quede bloqueada o cuando sea detenido el temporizador de intervalo. El prototipo es

  ```
  void set_hardware_timer(long interval);
  ```

  El parámetro `interval` especifica el intervalo de duración del temporizador en microsegundos. Utilícese `setitimer` para implantar esta función.

- `get_hardware_timer` devuelve el tiempo que queda en el temporizador de *hardware* si éste se encuentra en ejecución, o 0 si no lo está. Hágase uso de `getitimer` para implantar esta función. El prototipo es

  ```
  long get_hardware_timer(void);
  ```

- `stop_hardware_timer` detiene el temporizador de *hardware* si éste se encuentra en ejecución. El prototipo es

  ```
  void stop_hardware_timer(void);
  ```

- `is_timer_interrupt_blocked` devuelve 1 si `SIGALRM` está bloqueada o de lo contrario 0. El prototipo es

  ```
  int is_timer_interrupt_blocked(void);
  ```

- `block_timer_interrupt` bloquea la señal `SIGALRM`. El prototipo es

  ```
  void block_timer_interrupt(void)
  ```

- `unblock_timer_interrupt` desbloquea la señal `SIGALRM`. El prototipo es

  ```
  void unblock_timer_interrupt(void);
  ```

- `wait_for_interrupt` llama a `sigsuspend` para esperar hasta que sea atrapada una señal. Esto no garantiza que la señal provenga de la expiración de un temporizador. Normalmente se entra a esta función con la señal del temporizador blo-

queada. La señal utilizada por `sigsuspend` no debe desbloquear ninguna de las señales que ya estaban bloqueadas y que son distintas de las utilizadas por los temporizadores. Si el programa principal ha bloqueado `SIGINT`, entonces el programa no debe terminar si se presiona la combinación `ctrl-c`. El prototipo de `wait_for_interrupt` es

```
void wait_for_interrupt(void);
```

Algunas de estas funciones no se necesitarán hasta más adelante. La interfaz del temporizador de *hardware* está aislada en este archivo, de modo que el uso de temporizadores POSIX.1b o de un temporizador subyacente distinto de `ITIMER_REAL` sólo requerirá la modificación de estas funciones. Para ello defínase un archivo de encabezado con el nombre `hardware_timer.h` que contenga los prototipos de las funciones del objeto `hardware_timers`.

6.5.3 Implantación del programa principal

Escríbase un programa principal que haga lo siguiente:

- Iniciar todas las variables y estructuras llamando a `init_timers`.
- Llamar a `start_timer` con los valores recibidos a través de la línea comando.
- Detectar el momento en que el temporizador ha expirado llamando a `remove_top_event()` en un lazo hasta que el evento sea retirado. Esta forma de espera ocupada se conoce como *escrutinio* (polling) y es muy ineficiente.
- Imprimir un mensaje informativo en la salida estándar.
- Terminar.

Una vez que el programa esté trabajando, llamar a `wait_for_timer_event` en lugar de hacer el lazo.

6.5.4 Instrumentación del código del temporizador, el objeto `showall`

El código con manejadores de señal y temporizadores es difícil de probar debido a la naturaleza impredecible de los eventos que controlan al programa. Una temporización particular de eventos que sea causa de un error podría presentarse muy rara vez o ser difícilmente reproducible. Por otra parte, el comportamiento del programa depende no sólo de los valores de la entrada, sino también de la rapidez con la que se generan los datos.

Esta sección describe cómo instrumentar el código que llama a la función `showall` como un paso preliminar en la prueba. Esta instrumentación es importante para depurar las siguientes partes del proyecto. Para ello se proporciona el código de la función `showall`. Esta subsección explica lo que hace `showall` y cómo utilizarla en el programa.

El prototipo para `showall` es

```
void showall(int traceflag, event_t ev_type, int timer,
             int blocked_flag);
```

Si `traceflag` es `1`, la función `showall` muestra el mensaje correspondiente al valor de `ev_type` y visualiza la estructura de datos `timers` en un formato de una línea. También presenta el valor del parámetro `timer` si tiene sentido hacerlo para el evento. El programa 6.4 muestra el archivo `showall.h` que debe incluirse en los programas que llaman a `showall`.

Programa 6.4: Archivo de encabezado `showall.h`.

```
typedef enum ev_t {TIMER_INITIALIZE,
        TIMER_INTERRUPT_ENTER, TIMER_INTERRUPT_EXIT,
        TIMER_START_ENTER, TIMER_START_EXIT,
        TIMER_STOP_ENTER, TIMER_STOP_EXIT,
        TIMER_REMOVE_EVENT_ENTER, TIMER_REMOVE_EVENT_EXIT,
        TIMER_REMOVE_EVENT_NONE, TIMER_REMOVE_EVENT_OK,
        TIMER_START_NONE_RUNNING, TIMER_START_THIS_NOT_RUNNING,
        TIMER_START_THIS_RUNNING, TIMER_STOP_RUNNING,
        TIMER_STOP_EXIT_NOT_ACTIVE, TIMER_STOP_EXIT_NOT_RUNNING,
        TIMER_WAIT_INPUT, TIMER_GOT_INPUT} event_t;

void init_showall(void);
void showall(int traceflag, event_t ev_type, int timer,
             int blocked_flag);
```

Programa 6.4

Ejemplo 6.4

La siguiente proposición muestra la forma en que debe llamarse a `showall`. *Colóquese esta llamada cerca de la función* `myhandler` *del objeto* `mytimers`.

```
showall(traceflag, TIMER_INTERRUPT_ENTER, this_timer, 1);
```

La llamada a `showall` del ejemplo 6.4 aparece al principio de `myhandler`. `myhandler` obtiene el valor de `this_timer` de `timers.running`. El último parámetro indica que `showall` puede suponer que la señal del temporizador está bloqueada. Si `traceflag` es `1`, entonces cada vez que se entra al manejador de interrupciones, `showall` hará que se muestre una línea similar a la siguiente:

```
**** 6.39043: Timer Interrupt Enter 1 B(1,1.00) A(1,1:0) (0E)
```

La interpretación de la salida de `showall` es

- Los cuatro asteriscos (`****`) identifican el mensaje como una salida proveniente de `showall`.
- El primer campo numérico es el tiempo en segundos desde el inicio de la ejecución del programa.

- A continuación aparece una cadena de caracteres que indica el evento que hizo que este mensaje apareciera. En el ejemplo 6.4 `showall` fue llamada con el tipo de evento `TIMER_INTERRUPT_ENTER`.
- A continuación aparece el número del temporizador que estaba en ejecución cuando la señal fue atrapada.
- La letra `B` indica que la interrupción del temporizador fue bloqueada cuando se entró a `showall`. Esta información se obtiene del último parámetro de `showall`. Después de esta información aparecen tres asteriscos más si el parámetro no concuerda con el estado de bloqueo real de la señal del temporizador como es notificado por `is_timer_interrupt_blocked`, lo cual indica un error en una de las funciones.
- Los valores que siguen entre paréntesis son el temporizador `running` y el tiempo `active[running]`, mostrados en segundos.
- El siguiente campo comienza con A: y contiene una lista de todos los temporizadores con sus entradas activas. En el caso anterior, sólo el temporizador `1` se encuentra activo.
- Finalmente, `(0E)` indica que no hay eventos en la lista de eventos. Si existen eventos, el número de éstos aparecerá antes de la `E`, y los eventos después de ésta.

El programa 6.4 muestra los posibles valores de `event_t`. Los siguientes son los valores de `event_t` importantes para la función `start_timer`. En cada caso, el parámetro `timer` es el número del temporizador que se pone en marcha.

`TIMER_START_ENTER:`	primera proposición de `start_timer`
`TIMER_START_EXIT:`	antes de cada proposición `return`
`TIMER_START_THIS_NOT_RUNNING:`	cuando `start_timer` determina que no hay temporizadores activos.
`TIMER_START_ONE_RUNNING:`	cuando `start_timer` determina que al menos uno de los temporizadores está activo.
`TIMER_START_THIS_RUNNING:`	cuando `start_timer` determina que el temporizador que debe ponerse en marcha ya se está ejecutando.

La implantación con un solo temporizador sólo requiere los primeros dos. El programa 6.5 presenta el código fuente de `showall` y de otros archivos que son necesarios. La función `showall` hace uso de `printf` aunque esta función no se encuentre en la lista de POSIX de funciones seguras con respecto de señales asíncronas. Sun Solaris tiene una función `printf` segura en lo que corresponde a señales asíncronas, con lo que fue más fácil ceder a la no transportabilidad que hacer uso de `write`.

La función `showall` y sus funciones de soporte deben estar en un archivo aparte. La función `showall` hace la mayor parte de su trabajo con la señal del temporizador bloqueada de modo que la estructura del temporizador no cambiará durante su cálculo. Eso ocasiona que la señal se bloquee, si no es que ya lo estaba. Antes de regresar, `showall` desbloquea la señal

sólo si ésta no se encontraba bloqueada en el momento en que se llamó a `showall`. El último parámetro de `showall` indica si la señal está bloqueada. La función que llama debe tener esta información. Como ayuda de depuración, `showall` verifica si la variable `blocked_flag` tiene esta información de manera correcta.

Programa 6.5: Función `showall`.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>
#include "mytimers.h"
#include "hardware_timer.h"
#include "showall.h"
extern int gettimeofday(struct timeval *tp, void *tzp);

static double initial_tod = 0.0;

/* get_time devuelve como un tipo double el número de segundos transcurridos
desde enero 1, 1970 */
static double get_time(void)
{
   struct timeval tval;
   double thistime = 0.0;

   if (gettimeofday(&tval,NULL) == -1)
      perror("Warning, cannot get time of day");
   else
      thistime = tval.tv_sec + (double)tval.tv_usec/MILLION;
   return thistime;
}

/* get_relative_time devuelve el número de segundos transcurridos desde la
llamada a init_showall */
{
   return get_time() - initial_tod;
}
/* init_showall pone en la variable global initial_tod el número de segundos
transcurridos desde enero 1, 1970 */
void init_showall(void)
{
   initial_tod = get_time();
}

/* time_to_double convierte microsegundos en segundos como un tipo double *
static double time_to_double(long interval)
{
   return (double)interval/MILLION;
}
```

```
/* show_timer_data muestra la estructura de datos timers */
static void show_timer_data(void)
{
   int i;

   printf("(%d,%.2f) A:", get_running_timer(),
      time_to_double(get_timer_value(get_running_timer())));
   for (i = 0; i < MAXTIMERS; i++)
      if (get_timer_value(i) >= 0)
         printf("(%d,%.2f) ", i, time_to_double(get_timer_value(i)));
   printf(" (%dE", get_number_events());
   for (i = 0; i < get_number_events(); i++)
      printf(" %d", get_event(i));
   printf(")\n");
}
/* showall muestra los temporizadores con los mensajes que corresponden a
ev_type */
void showall(int traceflag, event_t ev_type, int timer,
             int blocked_flag)
{
   int actual_blocked_flag;

   if (!traceflag)
      return;
   actual_blocked_flag = is_timer_interrupt_blocked();
   if (!blocked_flag)
      block_timer_interrupt();
   printf("**** %8.4f: ", get_relative_time());
   switch(ev_type) {
      case TIMER_INITIALIZE:
         printf("Timer Initialize ");
         break;
      case TIMER_INTERRUPT_ENTER:
         printf("Timer Interrupt Enter %d ", timer);
         break;
      case TIMER_INTERRUPT_EXIT:
         printf("Timer Interrupt Exit  %d ", timer);
         break;
      case TIMER_START_ENTER:
         printf("Timer Start Enter %d ", timer);
         break;
      case TIMER_START_EXIT:
         printf("Timer Start Exit  %d ", timer);
         break;
      case TIMER_STOP_ENTER:
         printf("Timer Stop Enter %d ", timer);
         break;
      case TIMER_STOP_EXIT:
```

```
            printf("Timer Stop Exit  %d ", timer);
            break;
        case TIMER_REMOVE_EVENT_ENTER:
            printf("Timer Remove Event Enter ");
            break;
        case TIMER_REMOVE_EVENT_EXIT:
            printf("Timer Remove Event Exit  ");
            break;
        case TIMER_REMOVE_EVENT_NONE:
            printf("Timer Remove Event None  ");
            break;
        case TIMER_REMOVE_EVENT_OK:
            printf("Timer Remove Event OK    ");
            break;
        case TIMER_START_NONE_RUNNING:
            printf("Timer Start None Running ");
            break;
        case TIMER_START_THIS_NOT_RUNNING:
            printf("Timer Start This Not Running ");
            break;
        case TIMER_START_THIS_RUNNING:
            printf("Timer Start This Running ");
            break;
        case TIMER_STOP_RUNNING:
            printf("Timer Stop Running ");
            break;
        case TIMER_STOP_EXIT_NOT_ACTIVE:
            printf("Timer Stop Exit Not Active ");
            break;
        case TIMER_STOP_EXIT_NOT_RUNNING:
            printf("Timer Stop Exit Not Running ");
            break;
        case TIMER_WAIT_INPUT:
            printf("Timer Wait Input ");
            break;
        case TIMER_GOT_INPUT:
            printf("Timer Got Input ");
            break;
        default:
            printf("**** Unknown Event Type ");
            break;
        }
    if (blocked_flag)
        printf("B");
    else
        printf("U");
    if (blocked_flag != actual_blocked_flag)
        printf("****");
    show_timer_data();
```

```
    fflush(stdout);
    if (!blocked_flag)
        unblock_timer_interrupt();
}
```

Programa 6.5

Ponga el código del programa 6.5 en un archivo aparte. Instrumente las funciones del temporizador de modo que cada vez que ocurra algo interesante, el programa llame a `showall` con el valor apropiado de `event_t`. Para esta parte, sólo inserte las cuatro líneas siguientes:

Al principio, en `myhandler`:

```
showall(traceflag, TIMER_INTERRUPT_ENTER, this_timer, 1);
```

Antes de regresar de `myhandler`:

```
showall(traceflag, TIMER_INTERRUPT_EXIT, this_timer, 1);
```

Primera línea de `start_timer`:

```
showall(traceflag, START_TIMER_ENTER, timer, 0);
```

Antes de regresar de `start_timer`:

```
showall(traceflag, START_TIMER_EXIT, timer, 0);
```

Pruebe el programa con una amplia variedad de entradas apropiadas y observe la salida de `showall`.

Después de probar el código instrumentado, modifique el programa principal de modo que en lugar de establecer un temporizador a partir de argumentos de la línea comando, éste lea de la entrada estándar el número del temporizador y el intervalo. Añada un ciclo externo de modo que el programa principal se repita hasta que encuentre una marca de fin de archivo en el dispositivo de entrada estándar:

- Lea un número de temporizador y un intervalo en microsegundos del dispositivo de entrada estándar.
- Llame a `start_timer` para establecer el temporizador.
- Espere a que el temporizador expire como lo hizo antes.

6.6 Temporizadores múltiples

Las posibles interacciones entre varios temporizadores hacen que su implantación sea más compleja que la de un solo temporizador. Todos los tiempos que aparecen en el arreglo `active` están especificados en relación con el tiempo de inicio del temporizador de intervalo `ITIMER_REAL`. Suponga que un programa desea configurar el temporizador `4` con una duración de siete segundos, y que han transcurrido dos segundos desde que el programa configuró el temporizador `2` con un lapso de cinco segundos. El procedimiento que debe emplearse es

- Determinar cuánto tiempo queda en el temporizador real mediante una llamada a la función `get_hardware_timer`.

- Encontrar el inicio del temporizador real en relación con el temporizador que en ese momento se encuentra en ejecución, restando el tiempo que queda en el temporizador real del valor de temporización que tiene el temporizador en ejecución. (Utilice la función `get_running_timer`.)
- Calcular el tiempo del temporizador por configurar en relación con el tiempo de inicio sumando el tiempo de inicio relativo del paso 2 al tiempo solicitado.

La figura 6.3 muestra la estructura de datos `timers` después de que el programa configura el temporizador `2` para un lapso de cinco segundos (5,000,000 de microsegundos). Suponga que dos segundos después, el programa configura el temporizador `4` con un intervalo de siete segundos (7,000,000 de microsegundos). La figura 6.5 muestra la estructura de datos `timers` después de la configuración del temporizador `4`. El programa llama a la función `get_hardware_timer` y se ve que todavía quedan tres segundos (3,000,000 de microsegundos) en el temporizador de intervalo, de modo que han transcurrido dos segundos (5,000,000 - 3,000,000 de microsegundos) desde que se puso en marcha el temporizador `2`. A continuación, el programa calcula el tiempo para el temporizador `4` en relación con el tiempo de inicio del temporizador real y el resultado es nueve segundos (2,000,000 + 7,000,000 de microsegundos).

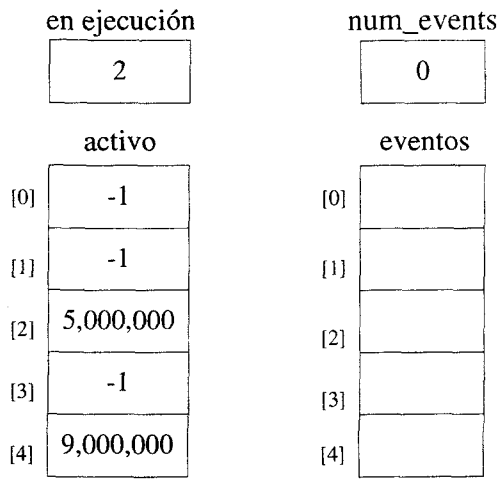


Figura 6.5: Estructura de datos `timers` después de la configuración del temporizador `4`.

El temporizador `running` es el mismo en las figuras 6.3 y 6.5, debido a que el temporizador `4` expira después del temporizador `2`. El programa no cambia la designación del temporizador `timer` ni tampoco lo reinicializa en este caso. Al continuar con la situación de la figura 6.5, suponga que un programa desea configurar el temporizador `3` con un segundo y una llamada a `get_hardware_timer` indica que en el temporizador real quedan dos segundos para que éste expire. La figura 6.6 muestra la situación después de que el programa configura el temporizador `3`. El programa reinicializa el temporizador real para que expire en un segundo y

ajusta todos los demás tiempos que hay en `active`. Los nuevos tiempos se hallan relacionados con el tiempo de inicio del temporizador 3 en lugar de estarlo con respecto del temporizador 2 (tres segundos antes), de modo que el programa resta tres segundos a cada uno de los tiempos activos.

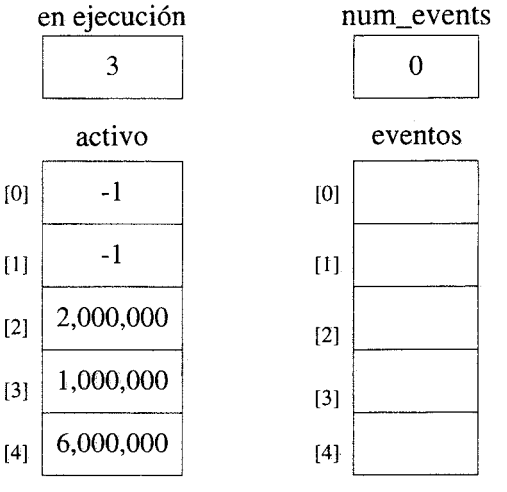


Figura 6.6: Estructura de datos `timers` después de la configuración del temporizador 3.

La figura 6.7 muestra la situación un poco después de haber transcurrido un segundo desde que se configuró el temporizador 3. El temporizador 3 expira y el 2 se convierte en el temporizador en ejecución. Todos los tiempos son reajustados para expirar en relación con el temporizador 2.

La figura 6.8 muestra la situación dos segundos después. El temporizador 2 expira y ahora el 4 es el que se convierte en el temporizador en ejecución.

6.6.1 Configuración de temporizadores múltiples

En esta sección se discute el manejo simultáneo de varios temporizadores. Para ello modifique la función `start_timer` y añada la función `stop_timer` para manejar todos los casos en que se configuren temporizadores mientras otros están en ejecución. En cualquier momento, cada temporizador se encuentra activo o inactivo. Un temporizador activo no puede aparecer en la lista de eventos, y será añadido a ésta en el momento en que expire. Si cualquiera de los temporizadores se encuentra activo, entonces sólo uno de ellos *está en ejecución* (*running*). El temporizador en ejecución es el próximo que expirará. El tiempo de expiración de éste se emplea en `set_hardware_timer` de modo que se genere una señal cuando el tiempo expire.

La forma en que el arranque y el paro de un temporizador afectarán la lista de eventos se elige arbitrariamente. Esta implantación elimina el evento que corresponde al temporizador

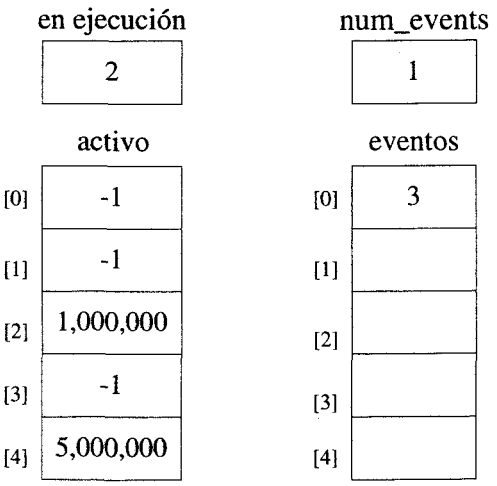


Figura 6.7: Estructura de datos `timers` después de la expiración del temporizador `3`.

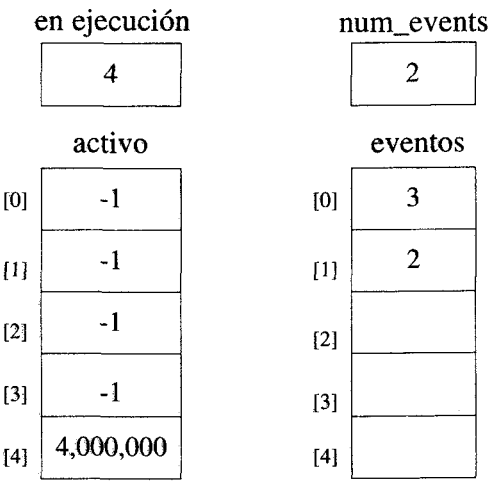


Figura 6.8: Estructura de datos `timers` después de la expiración del temporizador `2`.

que será puesto en marcha o detenido si es que existe uno. Esta opción asegura que ningún temporizador sea representado por más de un evento en la lista de eventos, de modo que el tamaño de la lista nunca pueda ser mayor que el número de temporizadores. La cota en la lista de eventos simplifica la implantación.

Cuando varios temporizadores están activos, el manejador de señal debe actualizar la estructura de datos `timers` restando `active[running]` de todos los tiempos activos. Si el tiempo se vuelve 0, entonces el temporizador correspondiente ha expirado y debe colocarse en la lista de eventos desactivándolo. Este método maneja varios tiempos de expiración al mismo tiempo.

La sección 6.5 estudia el caso más simple, que es el de poner en marcha un temporizador cuando no hay ninguno en ejecución. Algo similar es el caso en que el temporizador tiene que ponerse en marcha cuando ya está en ejecución pero los demás temporizadores están inactivos.

Suponga que algún temporizador está activo cuando otro se pone en marcha. Si este último expira después del temporizador en ejecución, entonces sólo es necesario modificar una entrada de la estructura de datos `timers`. Sin embargo, si el arranque de este temporizador hace que expire antes de que lo haga el que está en ejecución, entonces el temporizador de intervalo debe reiniciarse. También es necesario ajustar las entradas del arreglo `active` de modo que estén coordinadas con el tiempo de arranque del nuevo temporizador en ejecución. Lo anterior puede hacerse disminuyendo los tiempos activos hasta que sean iguales al tiempo que el temporizador en ejecución ha estado activo (`runtime`).

Utilice la función `get_hardware_timer` para saber qué tiempo queda, `remaining`, en el temporizador de intervalo y para calcular `runtime = active[running] - remaining`. Algunas cosas que tienen que hacerse en este caso son

- Eliminar el temporizador de la lista de eventos.
- Ajustar todos los tiempos activos por la cantidad `runtime`.
- Configurar un nuevo temporizador en ejecución, `running`.
- Poner en marcha el temporizador de intervalo con una llamada a `set_hardware_timer`.

Cuando el temporizador que debe ponerse en marcha sea el que está en ejecución, puede considerarse este un caso especial de los anteriormente mencionados o como uno aparte.

Cuando se hace una llamada a `stop_timer` a un temporizador que no se encuentra activo, lo único que ocurre es que se elimina el temporizador de la lista de eventos. Si el temporizador se encontraba activo pero no en ejecución, entonces se desactiva. Un caso interesante sería detener un temporizador en ejecución; esto sería similar a poner en marcha un temporizador y que éste se convirtiera en el temporizador en ejecución debido a que la estructura de datos `timers` necesitaría actualizarse utilizando `runtime`, con lo que sería necesario seleccionar un nuevo temporizador en ejecución.

En esta parte, el programa debe manejar todas las combinaciones de puesta en marcha y paro de los temporizadores, así como eliminar eventos de la lista de eventos. Extienda de manera apropiada las funciones `start_timer` y `myhandler`, y escriba las funciones `remove_top_event` y `stop_timer`, que anteriormente no eran necesarias.

Modifique el programa principal de modo que ahora éste interprete un intervalo negativo como un comando para detener el temporizador. Instrumente las funciones con `showall` para hacer pruebas. A continuación se presenta un resumen de los puntos donde debe colocarse una llamada a `showall`.

En `myhandler` con `timer` igual a `timers.running`:

`TIMER_INTERRUPT_ENTER:`	al inicio de la función
`TIMER_INTERRUPT_EXIT:`	última instrucción antes del regreso

En `remove_top_event` con `timer` igual a 0:

`TIMER_REMOVE_EVENT_ENTER:`	primera proposición
`TIMER_REMOVE_EVENT_EXIT:`	última proposición antes del regreso
`TIMER_REMOVE_EVENT_NONE:`	si no hay eventos que eliminar
`TIMER_REMOVE_EVENT_OK:`	si hay un evento por eliminar

En `start_timer` con `timer` igual al temporizador que debe ponerse en marcha:

`TIMER_START_ENTER:`	primera proposición
`TIMER_START_EXIT:`	última proposición antes del regreso
`TIMER_START_NONE_RUNNING:`	si ningún temporizador está en ejecución
`TIMER_START_THIS_NOT_RUNNING:`	si el temporizador que está en ejecución no es este
`TIMER_START_THIS_RUNNING:`	si este temporizador está en ejecución

En `stop_timer` con `timer` igual al temporizador que debe detenerse:

`TIMER_STOP_ENTER:`	primera proposición
`TIMER_STOP_EXIT:`	última proposición antes del regreso
`TIMER_STOP_EXIT_NOT_ACTIVE:`	última proposición antes del regreso si este temporizador no estaba activo
`TIMER_STOP_EXIT_NOT_RUNNING:`	última proposición si este temporizador estaba activo pero no en ejecución
`TIMER_STOP_RUNNING:`	si este temporizador estaba en ejecución

6.6.2 Pruebas con varios temporizadores

Incluso el código implantado por `showall` es difícil de probar sistemáticamente, puesto que la acción del programa depende de la velocidad con la que se escriba la entrada. Una solución para este problema es utilizar un controlador, `testtime`, para generar la entrada del programa. El programa 6.6 presenta el programa `testtime`. Éste debe vincularse con el objeto `hardware_timer`.

Al igual que con cualquier filtro, `testtime` toma la entrada que proviene del dispositivo de entrada estándar y envía su salida al dispositivo de salida estándar. La entrada consta de líneas que contienen tres enteros, `n` , `m` y `p`. El filtro lee estos enteros, espera `n` microsegundos y luego da salida, en una línea, a `m` y `p`. El programa `testtime` ignora cualquier carácter o grupo de caracteres que se encuentren después de los tres enteros, de modo que el usuario pueda agregar comentarios al final de cada línea de entrada.

Ejemplo 6.5

Cuando `testtime` *recibe la siguiente entrada*

```
1000000  2  5000000
2000000  4  7000000
1000000  3  1000000
```

`testtime` *espera un segundo* (1,000,000 *de microsegundos*) *y luego imprime la línea*

```
2  5000000
```

A continuación el programa espera dos segundos más e imprime la línea

```
4  7000000
```

y luego espera un segundo e imprime la línea

```
3  1000000
```

Suponga que los datos se encuentran en el archivo `timer.input` y que el programa de temporización es `timermain`. Utilice el filtro de la siguiente manera:

```
testtime < timer.input | timermain
```

Este comando hace que el temporizador `2` arranque después de un segundo y expire cinco segundos después (en el tiempo 6). Dos segundos después (en el tiempo 3) se pone en marcha el temporizador `4` y éste expira siete segundos después (en el tiempo 10). Un segundo después (en el tiempo 4) el temporizador `3` se configura para expirar un segundo después (en el tiempo 5). Ésta es exactamente la situación ilustrada en la figura 6.6 de la página 235.

El programa principal lee los valores del temporizador del dispositivo de entrada estándar, igual que antes. Conecte la salida de `testtime` con la entrada estándar del programa principal. Desarrolle un conjunto extenso de datos de entrada para probar el programa.

Programa 6.6: Programa `testtime`.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>
#include "hardware.h"

static int timer_expired = 0;
static void myalarm(int s)
{
   timer_expired = 1;
}

void main(int argc, char *argv[])
{
   long interval;
   int n1;
   int n2;

   if (argc != 1) {
      fprintf(stderr, "Usage: %s\n", argv[0]);
      exit(1);
   }
   catch_timer_interrupt(myalarm);
   for( ; ; ){
```

```
        if (scanf("%ld%d%d%*[^\n]", &interval, &n1, &n2) == EOF)
            break;
        if (interval <= 0)
            break;
        block_timer_interrupt();
        set_hardware_timer(interval);
        while (!timer_expired)
            wait_for_interrupt();
        timer_expired = 0;
        printf("%d %d\n", n1, n2);
        fflush(stdout);
        fprintf(stderr,"%d %d\n", n1, n2);
    }
    exit(0);
}
```

Programa 6.6

6.7 Implantación robusta de varios temporizadores

¿Qué sucede si se deposita una señal `SIGALRM` durante la ejecución de la función `start_timer`? Tanto el manejador de señal como la función `start_timer` modifican la estructura de datos `timers`, la cual es un recurso compartido. Éste es el problema clásico de la sección crítica para variables compartidas, y debe tenerse cuidado para asegurar que la estructura de datos `timers` no sea dañada. Es muy difícil determinar si existe un problema de esta naturaleza en el código con sólo probarlo. Los eventos que pueden dañar la estructura de datos son raros y por lo general no aparecen durante la fase de prueba. Si ocurre uno de estos eventos, no será fácil que esta situación se repita, por lo que habrá poca información sobre su causa.

Por tanto es importante analizar el problema para determinar dónde se encuentran las secciones críticas. En este caso, el análisis es sencillo ya que sólo se tiene una variable global, la estructura de datos `timers`. Cualquier función que modifique esta estructura debe hacerlo en un momento en que no pueda entrarse al manejador de la señal `SIGALRM`. Una manera más sencilla consiste en bloquear la señal `SIGALRM` antes de modificar la estructura de datos `timers`.

Pero el solo bloqueo de `SIGALRM` posiblemente no sea suficiente. ¿Qué sucede si el temporizador de intervalo expira durante la ejecución de la función `start_timer` y la señal `SIGALRM` está bloqueada? La función `start_timer` puede hacer que el nuevo temporizador sea el que está en ejecución, además de reinicializar el temporizador de intervalo. Antes de que la función `start_timer` termine, debe desbloquear la señal `SIGALRM`. En este momento se deposita la señal y el manejador supone que el nuevo temporizador ya ha expirado. Si bien esta secuencia es muy poco probable, el ejercicio 6.7 muestra otro problema.

Ejercicio 6.7

Describa una secuencia de eventos en los que la función `stop_timer` podría fallar si ésta bloqueara la señal a la entrada y la desbloqueara a la salida.

Respuesta:

La función `stop_timer` bloquea la señal `SIGALRM`. El temporizador será detenido cuando expire (esto es, el temporizador de intervalo genera una señal). Esta señal no se deposita de inmediato en el proceso puesto que se encuentra bloqueada. A continuación la función `stop_timer` pone en marcha el temporizador de intervalo que corresponde al próximo temporizador que expirará. Antes de que regrese, la función `stop_timer` desbloquea la señal y ésta es depositada. El manejador de señal se comporta como si el temporizador en ejecución hubiese ya expirado, cuando de hecho el que ha expirado es un temporizador diferente.

La manera más sencilla de manejar el problema descrito en el ejercicio 6.7 es ignorar la señal, más que bloquearla. Una señal ignorada no puede quedar pendiente. Después de ignorar la señal y manipular la estructura de datos, es necesario volver a atrapar la señal. Debe tenerse cuidado de no ignorar la señal a menos que ésta corresponda al temporizador que va a detenerse. El programa no funciona si `stop_timer` ignora la señal a la entrada y la vuelve a antes de regresar. Suponga que la ejecución del programa se encuentra ya en la función `stop_timer` y que el temporizador que debe detenerse no es el que está en ejecución. Si la señal es ignorada y el temporizador en ejecución expira antes de que la señal sea atrapada otra vez, entonces el temporizador puede perderse.

Otra posible opción que no implica ignorar la señal, es tener una variable global que indique a `myhandler` que no debe llevarse a cabo ninguna acción sobre la siguiente señal. En cualquier caso, es probable que sea necesario añadir funciones al objeto `hardware_timer` para manejar este problema.

Haga un análisis completo de las funciones `start_timer` y `stop_timer`, y modifique la implantación de la sección 6.6 de modo que los temporizadores sean manejados de manera robusta. Proponga un método de prueba para verificar que el programa trabaja de manera correcta. (La prueba implica simular eventos raros.)

6.8 mycron, pequeño servicio tipo cron pequeño

El servicio cron en UNIX permite que los usuarios ejecuten comandos en fechas y horas predeterminadas. Este servicio es muy flexible y permite la planificación regular de comandos. La implantación del mismo se hace con el demonio `cron`, y analiza un archivo que contiene información de temporización y comandos.

Implante un servicio cron personal simplificado, `mycron`. Escriba un programa que tome un argumento de la línea comando. El argumento representa un archivo de datos que contiene intervalos de tiempo y comandos. Cada línea del archivo de datos especifica un comando y la frecuencia con la que éste tiene que ejecutarse. Las líneas del archivo de datos tienen el siguiente formato:

```
interval command
```

`interval` especifica el número de segundos entre la ejecución de las instancias del comando. `command` es el comando que debe ejecutarse junto con sus argumentos.

- Implante el servicio cron anterior suponiendo que ninguno de los intervalos en el archivo de datos es mayor que el intervalo máximo que los temporizadores pueden manejar (alrededor de 30 minutos). Llame al archivo ejecutable `mycron`.
- Maneje el caso cuyos intervalos pueden ser arbitrariamente grandes. Suponga que el número de segundos en el intervalo puede representarse con un `long`. Intente hacer lo anterior sin modificar las funciones del temporizador.
- Encuentre una manera de ajustar los tiempos de inicio de modo que si dos comandos tienen intervalos similares, entonces éstos no siempre se ejecuten al mismo tiempo.

6.9 Implantación de temporizadores POSIX

Los temporizadores de POSIX.1b tienen varias ventajas sobre los de Spec 1170. Bajo POSIX un programa puede crear varios temporizadores para un reloj dado, como `CLOCK_REALTIME`. Los temporizadores tienen, en principio, una resolución mayor puesto que los valores son proporcionados hasta el nanosegundo más próximo, en lugar de hacerlo hasta el microsegundo más próximo. Por otra parte, el programa puede especificar la señal que será depositada para cada temporizador, y el manejador de la señal puede determinar qué temporizador la generó. Asimismo, las señales generadas por los temporizadores son colocadas en una cola, y el programa puede determinar cuándo se han perdido señales debido al estancamiento.

Existen varias maneras posibles de implantar los temporizadores múltiples de la sección 6.6 con temporizadores POSIX. El método más sencillo es utilizar un temporizador y hacer cambios pequeños en los tipos de datos para incluir una resolución mayor. Como alternativa, puede implantarse un temporizador POSIX para cada temporizador por *software*. La puesta en marcha y detener un temporizador así como el manejo de la señal del temporizador es independiente de los otros temporizadores, de modo que la única estructura compartida es la cola de eventos. Tal vez sea necesario reorganizar `mytimers` y el objeto `hardware_timer`. Es probable que también exista un número limitado de temporizadores que pueden ser soportados por cada proceso, el cual está dado por la constante `TIMER_MAX`. Si el número de temporizadores necesarios es pequeño, entonces este método será el más fácil de implantar. Una tercera opción es usar un solo temporizador POSIX, pero hay que modificar el método de implantación para hacer más exacta la temporización. El resto de la sección explica un programa de prueba que ilustra cómo hacer esto.

Uno de los problemas que se presentan con la implantación original del temporizador de este capítulo es que puede haber una deriva significativa de tiempo si los intervalos son pequeños. La *deriva de un temporizado (timer drift)* se presenta debido a que las señales no se depositan de inmediato una vez que se generan, y el manejo de la señal requiere un tiempo distinto de cero. Además siempre existe un retraso entre el momento en que el temporizador expira y el momento en que vuelve a iniciarse. Si el tiempo de expiración es breve, la deriva puede ser una parte importante del intervalo de tiempo. Para los temporizadores de Spec 1170 no hay manera de tomar en cuenta el tiempo transcurrido entre la expiración de un temporizador y la puesta en marcha del siguiente.

Suponga que el programa de la sección 6.6, el cual implanta varios temporizadores, pone en marcha dos de ellos —un temporizador expira en cinco segundos, y el otro en 5.1 segundos. Cuando el manejador de la señal atrapa la primera señal, también pone en marcha un temporizador adicional por un lapso de 0.1 segundos. En un sistema con mucha carga de trabajo, es

posible que la señal generada en el tiempo 5 no sea atrapada sino hasta el tiempo 5.05, o incluso más tarde. Si el manejador atrapa la señal en 5.05, entonces configurará el siguiente temporizador para que expire en 0.1 segundos después del tiempo 5.5 en lugar del tiempo 5.1. Este retraso (deriva del temporizador) afecta a todos los temporizadores activos.

La deriva del temporizador aparece debido a que el programa hace uso de tiempos de expiración relativos, con lo que se acumulan los errores pequeños en el tiempo de expiración. Este enfoque está dado por la naturaleza del temporizador de intervalo subyacente. Si este temporizador puede configurarse para que expire en un momento en particular más que en un lapso particular, entonces el programa puede emplear el tiempo real o absoluto para eliminar la deriva debido a que los retrasos en el manejo de la señal no cambiarán, en este caso, el momento de la siguiente expiración.

Los temporizadores de POSIX soportan tanto tiempos relativos, al igual que los temporizadores de Spec 1170, como tiempos absolutos. Para configurar un temporizador de modo que haga uso de tiempos absolutos, el parámetro `flags` de la función `timer_settime` debe hacerse igual con `TIMER_ABSOLUTE`. Con esto, el programa emplea tiempos absolutos para resolver el problema de la deriva del temporizador descrito anteriormente de la siguiente manera.

- Antes de poner en marcha el primer temporizador, se determina la hora actual y se suman cinco segundos para calcular el tiempo absoluto de expiración.
- A continuación se llama a la función `timer_settime` utilizando el tiempo de expiración absoluto para el parámetro `value` y `TIMER_ABSTIME` para el parámetro `flags`.
- Cuando el temporizador expira, el manejador de la interrupción añade 0.1 segundos al tiempo absoluto que está guardado y pone en marcha el temporizador empleando este valor. Siempre y cuando el segundo temporizador se ponga en marcha antes de que expire, no habrá ninguna deriva.

Es posible que el manejador de la señal configure el temporizador para que expire en un tiempo anterior al actual si la latencia en el manejo de la señal es muy grande. Esta expiración tardía no es considerada como un error, y en este caso el temporizador expira de inmediato.

El programa 6.7, `abstime`, permite que el usuario experimente con los temporizadores absolutos y relativos de POSIX.1b, facilitando la configuración de un temporizador para que expire a intervalos regulares. El programa `abstime` también calcula la deriva del temporizador para darle al usuario una idea de la magnitud de ésta cuando los intervalos de tiempo son pequeños. El programa también incluye una cantidad especificable de espera ocupada en el manejador de señal para simular distintas cargas de trabajo en el sistema.

Programa 6.7: Programa `abstime` que ilustra los temporizadores de POSIX.1b con tiempo absoluto.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <time.h>
```

```
#define BILLION  1000000000
#define D_BILLION 1000000000.0
#define INCTIME 0.01
#define NUMTIMES 1
#define SPINTIME 0.0

struct itimerspec interval;
int tflags;
int exitflag = 0;
double inctime = INCTIME;
int numtimes = NUMTIMES;
double spintime = SPINTIME;
int absflag = -1;

double time_to_double(struct timespec t)
{
   return t.tv_sec + t.tv_nsec/D_BILLION;
}

struct timespec double_to_time(double tm)
{
   struct timespec t;

   t.tv_sec = (long)tm;
   t.tv_nsec = (tm - t.tv_sec)*BILLION;
   return t;
}

struct timespec add_to_time(struct timespec t, double tm)
{
   struct timespec t1;

   t1 = double_to_time(tm);
   t1.tv_sec = t1.tv_sec + t.tv_sec;
   t1.tv_nsec = t1.tv_nsec + t.tv_nsec;
   while (t1.tv_nsec > BILLION) {
      t1.tv_nsec = t1.tv_nsec - BILLION;
      t1.tv_sec++;
   }
   return t1;
}

/*
 * spinit ejecuta un ciclo durante stime segundos antes de regresar. Si
 * tiene éxito devuelve 0. Si hay una falla la función devuelve -1 y
 * pone un valor en errno. */
int spinit (double stime)
```

```
{
   struct timespec timecurrent;
   double timenow;
   double timeend;

   if (spintime == 0.0)
      return 0;
   if (clock_gettime(CLOCK_REALTIME, &timecurrent) < 0)
      return -1;
   timenow = time_to_double(timecurrent);
   timeend = timenow + stime;
   while (timenow < timeend) {
      if (clock_gettime(CLOCK_REALTIME, &timecurrent) < 0)
         return -1;
      timenow = time_to_double(timecurrent);
   }
   return 0;
}

void my_handler(int signo, siginfo_t* info, void *context)
{
   static int timesentered = 0;
   int timid;

   timesentered++;
   if (timesentered < numtimes) {
      if (inctime < 0.0)
         return;
      if (spinit(spintime) == -1) {
         perror("Spin failed in my_handler");
         exit(1);
      }
      if (absflag)
         interval.it_value = add_to_time(interval.it_value, inctime);
      timid = *(int *)(info->si_value.sival_ptr);
      if (timer_settime(timid, tflags, &interval, NULL) < 0){
         perror("Could not start timer in handler");
         exit(1);
      }
   }
   else
      exitflag = 1;
}

void main(int argc, char *argv[])
{
   struct sigaction act;
   struct sigevent evp;
   struct timespec currenttime;
```

```
struct timespec res;
sigset_t sigset;
timer_t timid;
double total_time;
double calctime;
double starttime;
double endtime;

if (argc > 1) {
    if (!strcmp(argv[1], "-r"))
       absflag = 0;
    else if (!strcmp(argv[1], "-a"))
       absflag = 1;
}
if ( (argc < 2) || (absflag < 0) ){
   fprintf(stderr,
      "Usage:  %s -r | -a [inctime [numtimes [spintime]]]\n",
      argv[0]);
   exit(1);
}

if (argc > 2)
    inctime = atof(argv[2]);
if (argc > 3)
    numtimes = atoi(argv[3]);
if (argc > 4)
    spintime = atof(argv[4]);
fprintf(stderr, "pid = %ld\n", (long)getpid());

sigemptyset(&act.sa_mask);
act.sa_flags = SA_SIGINFO;
act.sa_sigaction = my_handler;
if (sigaction(SIGALRM, &act, NULL) < 0) {
    perror("sigaction failed");
    exit(1);
}
evp.sigev_notify = SIGEV_SIGNAL;
evp.sigev_signo = SIGALRM;
evp.sigev_value.sival_ptr = &timid;
if (timer_create(CLOCK_REALTIME, &evp, &timid) < 0) {
   perror("Could not create a timer");
   exit(1);
}
if (clock_getres(CLOCK_REALTIME, &res) < 0)
   perror("Can not get clock resolution");
else
   fprintf(stderr, "Clock resolution is %0.3f microseconds\n",
      1000000.0*time_to_double(res));
interval.it_interval.tv_sec = 0;
```

```
    interval.it_interval.tv_nsec = 0;
    if (inctime < 0.0)
       sleep(100);
    if (clock_gettime(CLOCK_REALTIME, &currenttime) < 0) {
       perror("Cannot get current time");
       exit(1);
    }
    starttime = time_to_double(currenttime);
    if (absflag) {
       fprintf(stderr,
          "abs time: interrupts: %d at %.6f seconds, spinning %.6f\n",
          numtimes, inctime, spintime);
       tflags = TIMER_ABSTIME;
       interval.it_value = currenttime;
    }
    else {
       fprintf(stderr,
          "rel time: interrupts: %d at %.6f seconds, spinning %.6f\n",
          numtimes, inctime, spintime);
       tflags = 0;
       interval.it_value.tv_sec = 0;
       interval.it_value.tv_nsec = 0;
    }
    interval.it_value = add_to_time(interval.it_value, inctime);
    if (timer_settime(timid, tflags, &interval, NULL) < 0){
       perror("Could not start timer");
       exit(1);
    }
    sigemptyset(&sigset);
    for( ; ; ){
       sigsuspend(&sigset);
       if (exitflag) {
          if (clock_gettime(CLOCK_REALTIME, &currenttime) < 0) {
             perror("Can not get current time");
             exit(1);
          }
          endtime = time_to_double(currenttime);
          total_time=endtime - starttime;
          calctime = numtimes*inctime;
          fprintf(stderr,
              "Total time: %1.7f, calculated: %1.7f, error = %1.7f\n",
               total_time, calctime, total_time - calctime);
          exit(0);
       }
    }
}
```

Programa 6.7

El programa 6.7 requiere un argumento de línea comando y tiene tres que son opcionales. La sinopsis del programa es

```
abstime -a|-r [inctime [numtimes [spintime]]]
```

El primer argumento de la línea comando debe ser `-a` o `-r`, y sirve para indicar tiempo absoluto o relativo. Los argumentos adicionales son `inctime`, `numtimes` y `spintime`. El programa genera señales `numtimes` señales `SIGALRM` a intervalos de tiempo `inctime`. El manejador de señal desperdicia `spintime` segundos antes de manejar la expiración del temporizador. Para tiempos absolutos, el programa `abstime` inicia el miembro `it_value` con el tiempo de inicio absoluto (tiempo transcurrido desde el 1 de enero de 1970) más el valor `inctime`. Si se elige tiempo relativo, entonces el valor de `it_value` es igual a `inctime`. `inctime` y `spintime` son valores de tipo `double`.

Ejemplo 6.6

El siguiente comando simula un manejador de señal que toma un tiempo apreciable en ejecutarse.

```
abstime -a 0.02 1000 0.01
```

Ejercicio 6.8

El comando del ejemplo 6.6 hace uso de tiempo absoluto. ¿Existen diferencias en la salida cuando se emplea tiempo relativo en lugar de tiempo absoluto?

Respuesta:

Para una ejecución de

```
abstime -a 0.02 1000 0.01
```

la salida puede tener la siguiente apariencia

```
pid = 6766
Clock resolution is 1.000 microseconds
abs time: interrupts:1000 at 0.020000 seconds, spinning 0.010000
Total time: 20.1690290, calculated: 20.0000000, error = 0.1690290
```

mientras que para una ejecución de

```
abstime -r 0.02 1000 0.01
```

la salida puede ser

```
pid = 6767
Clock resolution is 1.000 microseconds
rel time: interrupts:1000 at 0.020000 seconds, spinning 0.010000
Total time: 30.1253541, calculated: 20.0000000, error = 10.1253541
```

Cuando se emplean temporizadores absolutos el error es menor que uno por ciento, mientras que los temporizadores relativos muestran la deriva esperada correspondiente a la cantidad de tiempo de procesamiento.

La resolución del reloj se muestra con una llamada a `clock_getres`. Un valor típico para ésta puede estar entre 1000 nanosegundos y 20 milisegundos. Los 20 milisegundos (20,000,000 de nanosegundos o 50 Hz) es la menor resolución permitida por POSIX.1b. Un microsegundo (1,000 nanosegundos) es el tiempo necesario para ejecutar varios cientos de instrucciones en muchas máquinas rápidas. El hecho de que un sistema tenga una resolución de reloj de 1 microsegundo no implica que el programa pueda hacer uso de temporizadores con un lapso muy cercano a esta resolución. A menudo es necesario hacer una conmutación de contexto antes de que pueda entrarse al manejador de señal y, como lo indica la tabla 1.1, el tiempo necesario para llevar a cabo ésta es considerablemente mayor que la resolución del reloj.

Ejemplo 6.7

El siguiente comando emplea el programa 6.7 para estimar la resolución eficaz del temporizador de hardware *de una máquina.*

```
abstime -a 0
```

Ejemplo 6.8

El siguiente comando emplea el programa 6.7 para determinar el número máximo de señales de temporizador que puede manejar por segundo.

```
abstime -a 0.0 1000 0.0
```

Ejercicio 6.9

Ejecute el programa 6.7 con un valor de `inctime` que sea negativo:

```
abstime -a -1.0 1000 0.0
```

En este caso el programa imprime su número de ID y después suspende su ejecución. El manejador de señal no vuelve a iniciar el temporizador si `inctime` es negativo. El programa 6.8, `multikill`, envía varias señales a un proceso hasta que éste muere. Después de poner en marcha el programa 6.7 con un valor negativo de `inctime`, utilice el programa 6.8 para enviar señales `SIGALRM` a `abstime`. Puesto que `SIGALRM = 14`, utilice

```
multikill pid 14
```

El programa 6.7 imprime su ID de proceso antes de suspender su ejecución. Haga uso de este valor para el `pid` anterior. ¿Cuántas señales por segundo puede recibir el proceso?

Respuesta:

No es sorprendente el hecho de que la resolución del temporizador sea considerablemente peor que la capacidad de manejo de señales de la máquina. Una estación de trabajo Sun SPARCstation, bajo el control de Solaris 2.4, puede manejar sólo 100 señales por segundo generadas por temporizadores, al mismo tiempo que es capaz de manejar más de 4000 señales generadas externamente por el programa 6.8. Una máquina mucho más lenta que se ejecute bajo el control del mismo sistema operativo podría mane-

Programa 6.8: El programa `multikill` envía de manera continua señales a otros procesos hasta que éstos mueren.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void main(int argc, char *argv[])
{
   int pid;
   int sig;
   if (argc != 3) {
       fprintf(stderr, "Usage: %s pid signal\n", argv[0]);
       exit(1);
   }
   pid = atoi(argv[1]);
   sig = atoi(argv[2]);
   while (kill(pid, sig) == 0) ;
}
```

Programa 6.8

jar las mismas 100 señales generadas por temporizadores en un segundo, pero sólo alrededor de 600 señales generadas externamente.

El programa 6.7 ilustra otros puntos que son útiles en el uso de temporizadores POSIX. La información sobre el temporizador que generó la señal está disponible en el manejador de señal. Cuando se crea un temporizador, puede guardarse un entero o un apuntador en el miembro `sigev_value` de la estructura `struct sigevent`. Si el manejador de señal va a reiniciar dicho temporizador o si varios comparten el mismo manejador, éste debe tener acceso al ID del temporizador que generó la señal. Si el manejador fue configurado con el uso de la bandera `SA_SIGINFO`, entonces puede tener acceso al valor que `timer_create` guardó en `sigev_value` a través de su segundo argumento. `timer_create` no puede guardar directamente el ID del temporizador en `sigev_value` debido a que el ID no se conoce sino hasta después de haber creado el temporizador. Por consiguiente, la función guarda un apuntador al ID del temporizador en el miembro `sival_ptr` de `union sigval`.

6.10 Lecturas adicionales

La representación de arreglo para temporizadores trabaja bien cuando el número de tiempos es pequeño. Sin embargo, existen otras implantaciones posibles de las estructuras básicas que es necesario considerar: una cola de prioridad para los temporizadores y una lista ligada para los eventos, por ejemplo. Vale la pena examinar el artículo "Hashed and hierarchical timing wheels: Data structures for effcient implementation of a timer facility" por G. Varghese y T. Lauck [95]. El POSIX.1b Realtime Extension [53] proporciona un estudio excelente de los aspectos que hay detrás de la implantación de temporizadores a nivel de sistema.

En el futuro los aspectos de tiempo real prometen volverse más importantes. El libro *POSIX.4: Programming for the Real World* de Gallmeister [31] proporciona una introducción general a la programación en tiempo real bajo el estándar POSIX.1b. POSIX.4 era el nombre del estándar antes de que fuera aprobado. Ahora éste es una extensión del estándar POSIX.1 conocida como POSIX.1b.

Capítulo 7

Proyecto: *Desarrollo de intérpretes de comando*

En este capítulo se desarrollará un intérprete de comandos desde su inicio y se explorarán las complejidades de la creación de procesos, terminación, identificación y el correcto manejo de las señales. También se analiza el control de trabajos y la E/S (Entrada/Salida) de terminales. El proyecto final integra estos conceptos al incorporar un controlador de trabajos al intérprete de comandos.

Un intérprete de comandos (*shell*) es el proceso que ejecuta la interpretación de la línea de comando. Lee de su entrada estándar y ejecuta el comando correspondiente a la línea alimentada. La ejecución de un comando normalmente significa crear un proceso secundario (hijo) para la ejecución de un comando. En el caso más simple, el intérprete de comandos lee un comando y se bifurca en un hijo (forks) para la ejecución de dicho comando. El proceso primario (padre) espera a que el hijo complete la ejecución antes de leer en otro comando. Un intérprete de comandos real maneja entubamiento de procesos y redireccionamiento, así como procesos en plano (foreground), de fondo (background) y señales.

Este capítulo inicia con los intérpretes de comandos más simples y estudia paso por paso un intérprete de comandos operativo. Todos los ejemplos de intérpretes de comandos incluyen al archivo de encabezado `ush.h` mencionado en el programa 7.1. Los ejemplos también utilizan `makeargv` que se menciona en el programa 1.2 en la página 22 de este libro.

Programa 7.1: El archivo `ush.h` está incluido en todos los programas de este capítulo.

```
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
```

```
#include <stdio.h>
#include <sys/wait.h>
#include <limits.h>

#define STDMODE 0600
#define DELIMITERSET " ><|&"

#ifndef MAX_CANON
#define MAX_CANON 256
#endif
#define TRUE 1
#define FALSE 0
#define BLANK_STRING  " "
#define PROMPT_STRING ">>"
#define QUIT_STRING "q"
#define BACK_STRING "&"
#define PIPE_STRING "|"
#define NEWLINE_STRING "\n"
#define IN_REDIRECT_SYMBOL '<'
#define OUT_REDIRECT_SYMBOL '>'
#define NULL_SYMBOL '\0'
#define PIPE_SYMBOL '|'
#define BACK_SYMBOL '&'
#define NEWLINE_SYMBOL '\n'

int makeargv(char *s, char *delimiters, char ***argvp);
int parsefile(char *inbuf, char delimiter, char **v);
int redirect(char *infilename, char *outfilename);
void executecmdline(char *cmd);
int connectpipeline(char *cmd, int frontfd[], int backfd[]);
```

Programa 7.1

Las partes iniciales del proyecto desarrollan un intérprete de comandos sencillo para que practique el usuario. La sección 7.1 muestra el intérprete de líneas de comando más básico. La sección 7.2 agrega un redireccionamiento y la sección 7.3 agrega entubamientos. La sección 7.4 explica cómo un intérprete de comandos maneja las señales para un proceso de primer plano. Los programas para cada una de estas fases son presentados en conjunto con una serie de ejercicios que destacan las preguntas más importantes. Trabaje con estos ejercicios antes de continuar con la parte principal de este proyecto.

La parte central de este proyecto lo constituyen el manejo de señales y el control de trabajos. La sección 7.5 introduce la maquinaria necesaria para un controlador de trabajos. Escriba el programa `showid` descrito en esa sección y practique con éste diferentes intérpretes de comandos. La sección 7.6 describe cómo manejar procesos de fondo sin utilizar un controlador de trabajo y la sección 7.7 presenta un controlador de trabajo a nivel de usuario. Finalmente, la sección 7.8 explica la implementación de un intérprete de comandos completo con un controlador de trabajo.

7.1 Un intérprete de comandos sencillo

El programa 7.2 muestra la versión 1 de `ush` (intérprete de comandos ultrasencillo). El proceso del intérprete de comandos se bifurca en un hijo que construye un arreglo de tipo `argv` y los comandos `execvp`'s modificados en la entrada normal.

Programa 7.2: Código para la versión 1 del programa `ush`.

```
#include "ush.h"
#define MAX_BUFFER 256

void main (void)
{
    char inbuf[MAX_BUFFER];
    char **chargv;

    for( ; ; ) {
       gets(inbuf);
       if (strcmp(inbuf, QUIT_STRING) == 0)
          break;
       else {
          if (fork() == 0) {
             if (makeargv(inbuf, BLANK_STRING, &chargv) > 0)
                execvp(chargv[0], chargv);
          }
          wait(NULL);
       }
    }
    exit(0);
}
```

Programa 7.2

Ejercicio 7.1

Ejecute el programa 7.2 con una variedad de comandos como `ls`, `grep` y `sort`. ¿Se comporta como era esperado?

Respuesta:

No. El `ush` del programa 7.2 no muestra un indicador (prompt) o expande nombres de archivo que contengan caracteres como `*` y `?`. Este intérprete de comandos tampoco maneja algunos de los más importantes comandos disponibles en todos los intérpretes de comandos (por ejemplo, `cd`).

El `ush` del programa 7.2 tampoco emplea comillas de la misma manera que otros intérpretes de comandos normales lo hacen. Los intérpretes de comandos normales permiten que las comillas garanticen que un argumento específico pase completo hacia el `exec` y no sea interpretado por el intérprete de comandos.

Ejercicio 7.2

¿Qué pasa si el programa 7.2 no llama a `wait`?

Respuesta:

Si un usuario introduce un comando antes de que el anterior se complete, los comandos se ejecutaran simultáneamente. Ambos comandos son afectados por el teclado de entrada y por `ctrl-c`.

Otro problema de la versión 1 de `ush` es que no atrapa los errores en el `execvp`. Esta omisión tiene algunas consecuencias interesantes si el usuario introduce un comando no válido. Cuando el `execvp` ocurre, el control nunca regresa al hijo. Sin embargo, cuando falla, el hijo opera y ¡también trata de obtener una línea comando!

Ejercicio 7.3

Ejecute el programa 7.2 con varios comandos no válidos. Ejecute un `ps` y observe el número de intérpretes de comandos que se están ejecutando. Trate de salir. ¿Qué pasa?

Respuesta:

Cada vez que un comando no válido es introducido, el nuevo proceso actúa como un intérprete de comandos adicional. El usuario debe teclear `q` una vez por cada intérprete de comandos.

Otro pequeño problema con el programa 7.2 es el uso de `MAX_BUFFER`, una constante no portable y definida por el usuario. La versión 1 de `ush` también usa `gets` en lugar de `fgets`, por lo que existe la posibilidad que se desborde el espacio que fue destinado para la entrada.

El programa 7.3 muestra una versión mejorada de `ush` que tiene un aviso y que puede manejar una falla de `execvp`. La constante `MAX_CANON` definida por el sistema reemplaza a la `MAX_BUFFER` fija no portable. El `fgets` reemplaza a `gets`.

Programa 7.3: Versión 2 de `ush`.

```
#include "ush.h"

void main (void)
{
    char inbuf[MAX_CANON+1];
    pid_t child_pid;

    for( ; ; ) {
       fputs(PROMPT_STRING, stdout);
       if (fgets(inbuf, MAX_CANON, stdin) == NULL)
          break;
       if (*(inbuf + strlen(inbuf) - 1) == NEWLINE_SYMBOL)
          *(inbuf + strlen(inbuf) - 1) = 0;
       if (strcmp(inbuf, QUIT_STRING) == 0)
          break;
       else {
          if ((child_pid = fork()) == 0) {
```

```
            executecmdline(inbuf);
            exit(1);
        }
        else if (child_pid > 0)
            wait(NULL);
    }
  }
  exit(0);
}
```

Programa 7.3

El intérprete de comandos en el programa 7.3 no finaliza si existe un error en el `fork`. En general el intérprete de comandos debe ser a prueba de errores y una prueba total que requiere de un gran esfuerzo. La función `executecmdline` reemplaza a las llamadas de `makeargv` y `execvp`. El control nunca debe regresar de esta función.

Ejemplo 7.1

Lo siguiente es una muestra de `executecmdline` *para el programa 7.3.*

```
void executecmdline(char *incmd)
{
    char **chargv;
    if (makeargv(incmd, BLANK_STRING, &chargv) > 0) {
        if(execvp(chargv[0], chargv) == -1) {
          perror("Invalid command");
          exit(1);
        }
    }
    exit(1);
}
```

Ejercicio 7.4

¿Por qué en el ejemplo 7.1 no existe un `perror` antes del último `exit(1)`? Observe el efecto que se obtiene cuando una cadena de caracteres de comandos vacía es introducida.

Respuesta:

En el caso del comando vacío, `makeargv` regresa 0 y `perror` imprime un mensaje correspondiente al valor actual de `errno`. Como no existía un error del sistema, el mensaje mostrado por `perror` no es necesariamente correcto.

Ejercicio 7.5

En el programa 7.2 y programa 7.3, sólo el hijo analiza la línea de comando. ¿Qué pasa si el padre analiza la línea de comando antes de `fork`? ¿Cuáles son los problemas de asignación y de designación de memoria que se presentan en estos programas, al mover el llamado a `makeargv` antes que a `fork`?

Respuesta:

Cuando existe un hijo, toda la memoria asignada por el hijo es liberada. Si el padre llamó a `makeargv` antes que a `fork`, el intérprete de comandos debe preocuparse por liberar la memoria asignada por `makeargv`.

Ejercicio 7.6

Trate de usar el comando `cd` como entrada al programa 7.3. ¿Qué pasa? ¿Por qué? Ayuda: Para obtener una explicación lea la página del manual que habla sobre `cd`.

Respuesta:

Puesto que `cd` debe cambiar el entorno del usuario, entonces no puede ser externa al intérprete de comandos, puesto que los comandos externos son ejecutados por el hijo del intérprete de comandos y un proceso no puede cambiar el entorno del padre. La mayoría de intérpretes de comandos implementan `cd` como un comando interno.

Ejercicio 7.7

Trate de dar a `ush` comandos como `ls-l` y `q` usando extraespacios iniciales, en forma intercalada. ¿Qué pasa?

Respuesta:

El programa 7.3 maneja correctamente los comandos como `ls-l` porque `makeargv` es capaz de manejar espacios extra iniciales y en forma intercalada. El comando `q` no funciona puesto que este comando es manejado directamente por `ush` que no tiene la habilidad de manejar espacios extra intercalados.

Ejercicio 7.8

Ejecute el comando `stty-a` y registre los valores actuales de los caracteres de control de la terminal. Lo siguiente puede ser un ejemplo de lo que podría aparecer:

```
intr = ^c; quit = ^|; erase = ^?; kill = ^u;
eof = ^d; eol = <undef>; eol2 = <undef>; swtch = <undef>;
start = ^q; stop = ^s; susp = ^z; dsusp = ^y;
rprnt = ^r; flush = ^o; werase = ^w; lnext = ^v;
```

Trate cada carácter de control bajo `ush` y bajo un intérprete de comandos normal y compare los resultados.

Ejercicio 7.9

Ejecute el comando `cat` bajo `ush` e introduzca algunas líneas a través de la terminal. Termine con el carácter `eof` (normalmente `ctrl-d`). Continúe con otro comando como `ls-l`. ¿Trabaja? En algunos sistemas, cuando se llega al fin de archivo para un descriptor archivo en particular, debe ser regresado explícitamente con el llamado de `rewind`. ¿Dónde debemos colocar `rewind` dentro del programa 7.3?

Respuesta:

`Rewind` debe de ser ejecutado directamente después de `wait (NULL)`.

En el ejercicio 7.8 el `erase` y `werase` continúan trabajando aun cuando no existe un código explícito para manejarlos en `ush`. La razón es que el programa de intérprete de comandos no recibe los caracteres directamente del teclado. En su lugar, el controlador de la terminal procesa la alimentación del teclado y pasa esta información a través de módulos adicionales al programa. Las terminales pueden operar en modalidad canónica (almacenamiento temporal de líneas:line-buffered) o en modalidad no canónica. El estado por omisión es la modalidad canónica.

En la modalidad canónica, la alimentación es regresada línea por línea. Por lo tanto, un programa no recibe ninguna alimentación hasta que el carácter de la nueva línea es introducido aunque lea un solo carácter a la vez. La alimentación es almacenada temporalmente por un controlador de una terminal hasta que encuentre un símbolo de nueva línea. El controlador de la terminal hace que la línea esté disponible para el programa cuando éste la requiera. Algún procesamiento de la línea ocurre mientras ésta es coleccionada. Si el controlador de la terminal encuentra los caracteres `erase` o `werase`, entonces ajusta el almacenamiento temporal (buffer) según se requiera.

La modalidad no canónica permite una mayor flexibilidad en el manejo de la E/S (Entrada/Salida). Por ejemplo, una aplicación de edición puede mostrar el mensaje `"entering cbreak mode"` para indicar que está entrando en una modalidad no canónica con el eco de carácteres no habilitado y con alimentación de un carácter a la vez. En la modalidad no canónica, la alimentación está disponible para el programa después de que el usuario haya tecleado un número especificado de caracteres o de que pase un tiempo determinado. En esa modalidad, las herramientas de edición de la modalidad canónica no están disponibles. Programas como los editores normalmente operan con la terminal en modalidad no canónica, mientras que los programas de usuario normalmente operan en terminales en modalidad canónica.

Examine y cambie las modalidades de la terminal llamando a `tcgetattr` y a `tcsettattr`, respectivamente. El controlador y sus módulos asociados también manejan caracteres especiales de edición. Llame a `tcsetattr` para cambiar las acciones individuales de edición del controlador y de sus módulos. Cuando se encuentra definida, la constante `MAX_CANON` da la longitud máxima de la línea que se puede procesar en modalidad canónica.

7.2 Redirección

UNIX maneja la E/S en forma independiente del dispositivo a través de descriptores de archivos. Un programa debe abrir un archivo o un dispositivo, por ejemplo una terminal (representada por un archivo especial), antes de ser accesado. El programa entonces accesa el archivo o dispositivo usando una asa (handle), la cual es utilizada como regreso del comando `open`. La redirección permite al programa reasignar una asa (handle) que ha sido abierta para un archivo que designe a otro archivo. (Veáse la sección 3.4 para recordar el concepto de redirección.)

En UNIX, la mayoría de los intérpretes de comandos permiten el redireccionamiento de la entrada estándar, salida estándar y el error estándar de la línea de comando. Los filtros son programas que leen de la entrada estándar y escribe en la salida estándar. El redireccionamiento en la línea de comando permite a los filtros operar en otros archivos sin necesidad de recompilación.

Ejemplo 7.2

El siguiente comando `cat` *redirecciona su entrada normal hacia* `my.input` *y su salida normal hacia* `my.output`.

```
cat < my.input > my.output
```

Recuerde que los descriptores de archivos se enredan en la llamada `exec` (a menos que sea prevenido específicamente de lo contrario). Para intérpretes de comandos esto significa que el hijo debe primero manejar el redireccionamiento antes de llamar a `exec`. (Después del `exec` el proceso no podrá manejar el redireccionamiento.)

El programa 7.4 muestra una versión de `executecmdline` que maneja el redireccionamiento de la entrada estándar y la salida estándar que es designada por una línea de comando de entrada, `incmd`. La función `redirect` mostrada en el programa 7.5 ejecuta la redirección apropiada. `parsefile` en el programa 7.6 parte los nombres de los archivos para que se utilicen en el redireccionamiento. Este remueve la primera aparición del carácter `search_symbol` y la ficha que sigue de `input_buffer`. A su regreso, `*tokenp` apunta hacia la ficha. La función `parsefile` regresa un 0 si la ejecucion fue exitosa o un `-1` en caso contrario.

Programa 7.4: Una versión de `executecmdline` que maneja redireccionamiento.

```
/ *
  * La función executecmdline analiza incmd para una posible redirección.
  * Llama a * redirect para ejecutar una redirección y a makeargv para crear el
    arreglo del comando
  * Luego execvps el comando ejecutable. La función sale
  * en condición de error, por lo que nunca regresa al llamador.
  */
void executecmdline(char *incmd)
{
    char **chargv;
    char *infilename;
    char *outfilename;

    if (parsefile(incmd, IN_REDIRECT_SYMBOL , &infilename) == -1)
       fprintf(stderr, "Incorrect input redirection\n");
    else if (parsefile(incmd, OUT_REDIRECT_SYMBOL, &outfilename) == -1)
       fprintf(stderr, "Incorrect output redirection\n");
    else if (redirect(infilename, outfilename) == -1)
       fprintf(stderr, "Redirection failed\n");
    else if (makeargv(incmd, BLANK_STRING, &chargv) > 0) {
       if (execvp(chargv[0], chargv) == -1)
           perror("Invalid command");
    }
    exit(1);
}
```

Programa 7.4

Programa 7.5: Un programa que redirige entrada y salida normales.

```
/*
 * La función redirect redirige la salida estándar a los archivos
   outfilename y la entrada
 * estándar al archivo infilename. Si cualquiera de infilename u
   outfilename esta en NULL,
 * entonces el correspondiente redireccionamiento no ocurre.
 * Regresa 0 si fue exitoso, o -1 si no lo fue.
 */
int redirect(char *infilename, char *outfilename)
{
   int indes;
   int outdes;

   if (infilename != NULL) {  /* redirige una entrada estándar hacia infilename */
      if ((indes = open(infilename, O_RDONLY, STDMODE)) == -1)
         return -1;
      if (dup2(indes, STDIN_FILENO) == -1) {
         close(indes);
         return -1;
      }
      close(indes);
   }
   if (outfilename != NULL) {/* redirige una salida estándar hacia outfilename */
      if ((outdes =
            open(outfilename, O_WRONLY|O_CREAT, STDMODE)) == -1)
         return -1;
      if (dup2(outdes, STDOUT_FILENO) == -1) {
         close(outdes);
         return -1;
      }
      close(outdes);
   }
   return 0;
}
```

Programa 7.5

Programa 7.6: La función `parsefile` busca fichas.

```
/*
 * Si está actualmente en s,la función parsefile quita la ficha próxima al
 *  delimitador (delimiter).
 * A su regreso, *v apunta hacia la ficha. El delimitador y la ficha han sido
 * quitados de s. Regresa 0 si el proceso fue exitoso, o -1 si existe un error.
 */
int  parsefile(char *s, char delimiter, char **v)
```

```
{
    char *p;
    char *q;
    int offset;
    int error = 0;

                    /* encuentra la posición del carácter delimitador */
    *v = NULL;
    if ((p = strchr(s, delimiter)) != NULL)  {
                      /* extrae la ficha que se halla después del delimitador */
        if ((q = (char *)malloc(strlen(p + 1) + 1)) == NULL)
            error = -1;
        else {
            strcpy(q, p + 1);
            if ((*v = strtok(q, DELIMITERSET)) == NULL)
               error = -1;
            offset = strlen(q);
            strcpy(p, p + offset + 1);
        }
    }
    return error;
}
```

Programa 7.6

7.3 Entubamientos

Los entubamientos (pipelines) son usados para conectar filtros en una línea de ensamble a fin de poder ejecutar funciones más complicadas.

Ejemplo 7.3

Los siguientes comandos redirigen la salida de `ls -l` *a la entrada estándar del comando* `sort` *y a la salida estándar del* `sort` *al archivo* `temp`.

```
ls -l | sort -n +4 > temp
```

El `ls` y el `sort` son procesos distintos conectados mediante un entubamiento. La conexión no implica que los procesos compartan los de archivos descriptores, sino que el intérprete de comandos crea un entubamiento que actúa como un almacenamiento temporal (buffer) entre los comandos como se muestran en la figura 7.1. El *shell* o intérprete de comandos une el entubamiento antes de bifurcar a los hijos, puesto que ambos procesos los accesan.

El programa 7.7 muestra una función que analiza la línea de comando y conecta un proceso a un entubamiento redireccionando la entrada del proceso al entubamiento `frontfd` y la salida estándar a `backfd`.

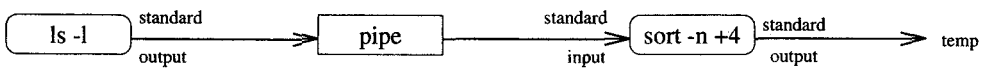


Figura 7.1: Un entubamiento que actúa como almacenamiento temporal (buffer) entre dos procesos en una conexión.

Programa 7.7: La función `connectpipeline` redirige la entrada estándar a *front*[0] y la salida estándar a *back*[1].

```
/* connectpipeline conecta el proceso con un entubamiento redirigiendo la
 * entrada estándar a frontfd[0] y la salida estándar a backfd [1].
 * Si frontfd[0] = -1, el proceso está al principio del entubamiento y la
 * entrada estándar puede ser redirigida hacia un archivo. Si backfd[1] = -1
 * el proceso está al final del entubamiento y una salida estándar puede ser
 * redirigida a un archivo.
 * De otra manera durante la redirección hacia un archivo se incurre en un
 * error. Si una redirección explícita
 * ocurre en cmd, ésta es removida durante el proceso. Se muestra un 0 si el
 * connectpipeline es exitoso y un -1 si no lo es.
 */
int connectpipeline(char *cmd, int frontfd[], int backfd[])
{
   int error = 0;
   char *infilename, *outfilename;

   if (parsefile(cmd, IN_REDIRECT_SYMBOL , &infilename) == -1)
      error = -1;
   else if (infilename != NULL && frontfd[0] != -1)
      error = -1;     /* no se permite redirección al inicio del entubamierto */
   else if (parsefile(cmd, OUT_REDIRECT_SYMBOL, &outfilename) == -1)
      error = -1;
   else if (outfilename != NULL && backfd[1] != -1)
      error = -1;      /* no se permite redirección al final del entubamiento */
   else if (redirect(infilename, outfilename) == -1)
      error = -1;
   else {                          /* ahora conecta los entubamientos apropiados */
      if (frontfd[0] != -1) {
         if (dup2(frontfd[0], STDIN_FILENO) == -1)
            error = -1;
       }
       if (backfd[1] != -1) {
          if (dup2(backfd[1], STDOUT_FILENO) == -1)
             error = -1;
       }
   }
```

```
                                    /* cierra los descriptores de archivos que no
                                       son necesarios */
    close (frontfd[0]);
    close (frontfd[1]);
    close (backfd[0]);
    close (backfd[1]);
    return error;
}
```

Programa 7.7

El último paso en `connectpipeline` del programa 7.7 es cerrar todos los descriptores innecesarios para que así el proceso pueda detectar el fin de archivo para los entubamientos. Puesto que un entubamiento es un almacenamiento temporal (buffer), éste se comporta más como una terminal que como un archivo ordinario, en el sentido de que no toda la entrada se encuentra inmediatamente disponible para el proceso que está leyendo de un entubamiento. La condición de fin de archivo ocurre solamente cuando el entubamiento está vacío y no hay escrituras conectadas al entubamiento. Si en el entubamiento existen descriptores de archivos extrabiertos que no sean usados (particularmente descriptores de escritura), el proceso no detecta el fin de archivo y queda colgado por tiempo indefinido.

La versión de `executecmdline` mostrada en el programa 7.8 maneja un entubamiento de longitud arbitraria. Para cada elemento en el entubamiento, el programa llamador crea un entubamiento de un inicio y otro de un fin del tubo antes de bifurcar el proceso para que ejecute el elemento.

Programa 7.8: Una versión de `executecmdline` que maneja tubos.

```
/*
 * executecmdline analiza una línea de comando y detecta los elementos
 * individuales de
 * un entubamiento. Crea un entubamiento que conecta a cada miembro intermedio del
 * entubamiento con su sucesor y bifurca a los hijos para ejecutar los elementos
 * individuales del entubamiento. La función executecmdline nunca debe
 * regresar. Maneja todos los errores llamando a exit(1).
 */
void executecmdline(char *incmd)
{
    char **chargv;
    pid_t child_pid;
    char *cmd;
    char *nextcmd;
    int frontfd[2];
    int backfd[2];

    frontfd[0] = -1;
    frontfd[1] = -1;
    backfd[0] = -1;
    backfd[1] = -1;

    child_pid = 0;
```

```
    if ((nextcmd = incmd) == NULL)
        exit(1);

    for ( ; ; ) {
        cmd = nextcmd;
        if (cmd == NULL) break;
                /* si es el último en el entubamiento, no bifurque a ningún otro */
        if ((nextcmd = strchr(nextcmd, PIPE_SYMBOL)) == NULL) {
            backfd[1] = -1;
            child_pid = 0;
        }
        else {
                        /* bifurque a un hijo para ejecutar el siguiente comando
                           en el entubamiento */
            *nextcmd = NULL_SYMBOL;
            nextcmd++;
            if (pipe(backfd)== -1) {
                perror("Could not create back pipe");
                exit(1);
            } else if ((child_pid = fork()) == -1) {
                perror("Could not fork next child");
                exit(1);
            }
        }
        if (child_pid == 0) {
                                            /* el hijo ejecuta el comando */
            if (connectpipeline(cmd, frontfd, backfd) == -1) {
                perror("Could not connect to pipeline");
                exit(1);
            } else if (makeargv(cmd, BLANK_STRING, &chargv) > 0) {
                if (execvp(chargv[0], chargv) == -1)
                    perror("Invalid command");
            }
            exit(1);
        }
              /* el padre cierra el inicio del entubamiento y
                 el final del tubo, como el inicio */
        close(frontfd[0]);
        close(frontfd[1]);
        frontfd[0] = backfd[0];
        frontfd[1] = backfd[1];
    }
    close(backfd[0]);
    close(backfd[1]);
    exit(1);
}
```

Programa 7.8

7.4 Señales

El manejo de señales es una parte integral del aprendizaje del usuario para el control de los intérpretes de comandos. Un intérprete de comandos que soporta un controlador de trabajo permite a los usuarios terminar la ejecución de los procesos, sacar líneas de comando (flush) a la mitad de su alimentación y mover procesos de la parte frontal al plano secundario. Es posible que el usuario normal no se dé cuenta de que las señales controlan estas acciones.

Suponga que el usuario teclea un `ctrl-c` para terminar la ejecución de un proceso. El controlador del dispositivo guarda temporalmente la terminal e interpreta los caracteres que han sido tecleados. Si el controlador encuentra el carácter `intr` (normalmente `ctrl-c`), envía una señal `SIGINT` a ese proceso. En una situación normal de operación de un intérprete de comandos, `ctrl-c` provoca que se termine el comando que se esté ejecutando, pero esto no ocasiona que el usuario se salga del intérprete de comandos.

En esta sección discutiremos el manejo de señales en un intérprete de comandos para procesos frontales e introduciremos el concepto de proceso de grupo. En la sección 7.5 se verá proceso de grupos y control de terminales, en la sección 7.6 desarrollaremos una versión de `ush` que maneja correctamente las señales para procesos de fondo, y en la sección 7.7 se estudiará el controlador de trabajo. Estas secciones nos llevarán a la parte principal del proyecto de este capítulo, como se especifica en la sección 7.8.

Si el usuario alimenta un `ctrl-c` con la versión 2 del programa 7.3 de `ush`, el intérprete de comandos ejecutará la acción predeterminada, que es terminar el programa. El programa 7.9 muestra la versión 3 de `ush`. El intérprete de comandos ignora `SIGINT` y `SIGQUIT`.

Programa 7.9: Versión 3 de `ush` que ignora `SIGINT` y `SIGQUIT`.

```
#include "ush.h"
#include <signal.h>
void main (void)
{
   char inbuf[MAX_CANON];
   pid_t child_pid;
   struct sigaction ignorehd;
   struct sigaction defaulthd;
   sigset_t blockmask;

                  /* Establece los manejadores para el indicador y el de omisión */
   ignorehd.sa_handler = SIG_IGN;
   sigemptyset(&ignorehd.sa_mask);
   ignorehd.sa_flags = 0;
   defaulthd.sa_handler = SIG_DFL;
   sigemptyset(&defaulthd.sa_mask);
   defaulthd.sa_flags = 0;
   if ((sigaction(SIGINT, &ignorehd, NULL) < 0) ||
       (sigaction(SIGQUIT, &ignorehd, NULL) < 0))  {
      perror("Shell failed to install signal handlers");
      exit(1);
   }
```

```
                        /* Establece una máscara para bloquear SIGINT y SIGQUIT */
    sigemptyset(&blockmask);
    sigaddset(&blockmask, SIGINT);
    sigaddset(&blockmask, SIGQUIT);
    sigprocmask(SIG_BLOCK, &blockmask, NULL);

    for( ; ; ) {
       fputs(PROMPT_STRING, stdout);
       if (fgets(inbuf, MAX_CANON, stdin) == NULL)
          break;
       if (*(inbuf + strlen(inbuf) - 1) == NEWLINE_SYMBOL)
          *(inbuf + strlen(inbuf) - 1) = 0;
       if (strcmp(inbuf, QUIT_STRING) == 0)
          break;
       else {
          if ((child_pid = fork()) == 0) {
             if ((sigaction(SIGINT, &defaulthd, NULL) < 0) ||
                 (sigaction(SIGQUIT, &defaulthd, NULL) < 0)) {
                perror("Child could not restore default handlers");
                exit(1);
             }
             sigprocmask(SIG_UNBLOCK, &blockmask, NULL);
             executecmdline(inbuf);
             exit(1);
          }
          else if (child_pid > 0)
             wait(NULL);
       }
    }
    exit(0);
}
```

Programa 7.9

Ejercicio 7.10

Si el usuario teclea un `ctrl-c` mientras el programa 7.9 está ejecutando `fgets`, nada aparecerá hasta que la tecla de retorno sea presionada. ¿Qué pasa si el usuario teclea `ctrl-c` a la mitad de una línea de comando?

Respuesta:

Cuando el usuario presiona `ctrl-c` a la mitad de la línea de comando aparecen los símbolos `^C`. Todos los caracteres de la línea anteriores a `ctrl-c` son ignorados debido a que el controlador de la terminal vacía el almacenamiento temporal (buffer) de entrada cuando `ctrl-c` es presionado (modalidad canónica de entrada), pero estos caracteres aparecen incluso en la actual línea de entrada puesto que `ush` no despliega de nuevo el indicador.

Para poder manejar `SIGINT` correctamente, `ush` debe capturar la señal en lugar de ignorarla. El comando `ush` también debe bloquear la señal en momentos decisivos para que no

sean enviadas las señales que no sean requeridas, mientras el intérprete de comandos esté estableciendo manejadores de señales o durante las operaciones de otras señales sensitivas. Recuerde que ignorar es diferente de bloquear. Se ignora una señal por medio del establecimiento de manejador de señal que sea `SIG_IGN` y se bloquea una señal estableciendo una bandera en la máscara de la señal. Las señales bloqueadas no son enviadas al proceso sino detenidas para su envío posterior.

El intérprete de comandos padre y el comando hijo deben manejar `SIGINT` de diferentes maneras. El intérprete de comandos padre borra la línea de entrada y regresa el indicador que el intérprete de comandos lleva a cabo a base de llamadas a `sigsetjmp` y `siglongjmp`.

La estrategia del hijo es diferente. Cuando el hijo se bifurca, hereda la máscara de la señal y tiene el mismo manejador de señal que el padre. El hijo no debe desplegar el indicador si es que la señal ocurre, sino que el hijo debe realizar la acción por omisión: salir. Para lograr esto, el padre debe bloquear la señal antes del `fork`. El hijo entonces instala el manejador por omisión antes de desbloquear la señal. Cuando el hijo ejecuta el `execvp`'s, el manejador por omisión es instalado automáticamente, puesto que el proceso está capturando la señal antes de que ésta sea bloqueada. El programa no puede esperar hasta ese momento para instalar el manejador por omisión, puesto que el hijo necesita desbloquear la señal antes de ejecutar `execvp`'s, y es posible que una señal llegue entre el desbloqueo de la señal y el `execvp`.

El intérprete de comandos en el programa 7.10 usa `sigsetjmp` para regresar el indicador cuando recibe un `ctrl-c`. El hijo instala el manejador predeterminado antes de desbloquear la señal encontrada después del `fork`.

Programa 7.10: Un intérprete de comandos que usa `siglongjmp` para manejar `ctrl-c`.

```
#include "ush.h"
#include <signal.h>
#include <setjmp.h>
static void jumphand(int);
static sigjmp_buf jump_to_prompt;
static volatile sig_atomic_t okaytojump = 0;
void main (void)
{
   char inbuf[MAX_CANON];
   pid_t child_pid;
   struct sigaction jumphd;
   struct sigaction defaulthd;
   sigset_t blockmask;
                        /* Establece una máscara para bloquear SIGINT y SIGQUIT */
   sigemptyset(&blockmask);
   sigaddset(&blockmask, SIGINT);
   sigaddset(&blockmask, SIGQUIT);
                  /* Establece los manejadores para el indicador y el de omisión */
   jumphd.sa_handler = jumphand;
   jumphd.sa_mask=blockmask;
   jumphd.sa_flags = 0;
   defaulthd.sa_handler = SIG_DFL;
```

```
    sigemptyset(&defaulthd.sa_mask);
    defaulthd.sa_flags = 0;
                              /* Prepararse para manejar un salto al indicador */
    if ((sigaction(SIGINT, &jumphd, NULL) < 0) ||
        (sigaction(SIGQUIT, &jumphd, NULL) < 0)) {
       perror("shell failed to install signal handlers");
       exit(1);
    }
    for( ; ; ) {
       if (sigsetjmp(jump_to_prompt, 1))
       /* Despliega de nuevo el indicador en una nueva línea si regresa de la señal */
          fputs(NEWLINE_STRING, stdout);
       okaytojump = 1;
       fputs(PROMPT_STRING, stdout);
       if (fgets(inbuf, MAX_CANON, stdin) == NULL)
          break;
       if (*(inbuf + strlen(inbuf) - 1) == NEWLINE_SYMBOL)
          *(inbuf + strlen(inbuf) - 1) = 0;

       if (!strcmp(inbuf, QUIT_STRING))
          break;
       else {
          sigprocmask(SIG_BLOCK, &blockmask, NULL);
          if ((child_pid = fork()) == 0) {
             if ((sigaction(SIGINT, &defaulthd, NULL) < 0) ||
                 (sigaction(SIGQUIT, &defaulthd, NULL) < 0)) {
                 perror("Child could not restore default handlers");
                 exit(1);
             }
             sigprocmask(SIG_UNBLOCK, &blockmask, NULL);
             executecmdline(inbuf);
             exit(1);
          }
          else if (child_pid > 0)
             wait(NULL);
          sigprocmask(SIG_UNBLOCK, &blockmask, NULL);
       }
    }
    exit(0);
}

static void jumphand(int signalnum)
{
    if (!okaytojump) return;
    okaytojump = 0;
    siglongjmp(jump_to_prompt, 1);
}
```

Programa 7.10

El `sigsetjmp` (sección 5.7) almacena la máscara de la señal y el ambiente actual almacenamiento temporal (buffer) de salto designado. Cuando el manejador de señales llama a `siglongjmp` con ese buffer de salto, los indicadores son restaurados y el control es transferido al punto del llamado de `sigsetjmp`. El programa 7.10 establece el punto de `jump_to_prompt` arriba del aviso del intérprete de comandos. Cuando es llamado en forma directa, `sigsetjmp` regresa un 0, y cuando es llamado a través de un `siglongjmp`, regresa un valor diferente de 0. Esta diferencia le permite al intérprete de comandos emitir una nueva línea cuando una señal ha ocurrido. El llamado de `siglongjmp` saca un elemento de la pila y restituye los valores en el registro a aquellos que tenía al punto donde fue originalmente llamado `sigsetjmp`.

Algunas veces los compiladores designan variables locales en los registros para aumentar la eficiencia. Es importante que estas variables no sean cambiadas cuando un `siglongjmp` es ejecutado y no se va a almacenar en registros. Use el calificador `volatile` desde el ISO C para suprimir este tipo de asignación.

El programa 7.10 usa el mismo manejador de señales de ambos `SIGINT` y `SIGQUIT`. Por ello, establece a `jumphd.sa_mask` para bloquear el envío de ambas señales cuando `jumphd` es llamado.

7.5 Grupos de proceso, sesiones y terminales controladoras

En la sección anterior se estudió el manejo de la señal para `ush` mediante comandos simples. El manejo de señales para entubamientos y procesos de fondo (background) requiere de una maquinaria adicional. Los entubamientos requieren de grupos de proceso; los procesos de fondo requieren de sesiones y de control de terminales.

Un *grupo de proceso* es una colección de procesos establecidos para propósitos de entrega de señales. Cada proceso tiene una *identificación (ID) de grupo de proceso,* que identifica a donde pertenece ese grupo de proceso. Tanto el comando `kill` y la función `kill` tratan a un valor de ID de un proceso negativo como un ID de un grupo de proceso, y mandan una señal a cada miembro del correspondiente grupo de proceso.

Ejemplo 7.4

El siguiente comando manda la señal `SIGINT` *al grupo de proceso* `3245`.

```
kill -INT -3245
```

Por el contrario, el siguiente comando manda a `SIGINT` *solamente al proceso* `3245`.

```
kill -INT 3245
```

Ejemplo 7.5

Si un usuario ejecuta el siguiente comando y después teclea un `ctrl-c`, *los tres procesos (por ejemplo,* `ls`, `sort` *y* `more`*) reaccionan a la señal* `SIGINT`, *pero el intérprete de comandos no reacciona a la señal.*

```
ls -l | sort -n +4 | more
```

Una manera de implementar el comportamiento del manejador de señales del ejemplo 7.5 es haciendo que los miembros de un entubamiento pertenezcan a un grupo de proceso individual que es diferente del intérprete de comandos. (Veáse ejercicio 7.11.)

El *líder del grupo de proceso* es aquel cuyo ID de proceso (identificación) tiene el mismo valor que el ID del grupo de proceso. El grupo de proceso permanece mientras exista cualquier proceso en el grupo. Por lo anterior, es posible que un grupo de proceso no tenga un líder, si éste muere o se asocia con otro grupo. Un proceso puede obtener el ID de su grupo de proceso con el `getpgrp` y puede cambiar su grupo de proceso con el `setpgid`.

SINOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpgrp(void);
int setpgid(pid_t pid, pid_t pgid);
```

POSIX.1, Spec 1170

El sistema de llamado `setpgid` establece el ID del grupo de proceso del proceso `pid`, para que se tenga un ID de grupo de proceso `pgid`. Utiliza el ID del proceso, de proceso llamador, si `pid` o `pgid` son iguales a 0. En el último caso, el proceso inicia un nuevo grupo, convirtiéndose él mismo en su líder.

Cuando un hijo es creado con `fork`, obtiene un nuevo ID de proceso, pero hereda el ID del grupo de proceso del padre. El padre también puede utilizar `setpgid` para cambiar de ID de grupo de un hijo, siempre y cuando el hijo no haya emitido un `exec`. Un proceso hijo puede darse a sí mismo un nuevo ID de grupo de proceso al establecer el ID del grupo de proceso igual al ID del proceso. El nuevo ID del grupo de proceso es exactamente lo que el hijo del intérprete de comandos padre requiere para asegurar que el envío de señales se efectúe correctamente.

Los procesos que están siendo ejecutados en el fondo o que puedan ser almacenados para el futuro en el fondo son más complicados que los procesos frontales. Los procesos de fondo no reciben `ctrl-c`. Si un proceso de fondo es traído hacia el frente, entonces nuevamente recibirá `ctrl-c`.

Ejemplo 7.6

Si un usuario ejecuta el siguiente comando y después teclea `ctrl-c`, *ninguno de los procesos en el entubamiento recibe la señal* `SIGINT`, *puesto que el entubamiento está en la parte del fondo.*

```
ls -l | sort -n +4 | more &
```

El entubamiento del fondo del ejemplo 7.6 no es afectado por `ctrl-c`. Una manera similar de implantar el mismo comportamiento en `ush` es hacer que el proceso ignore la señal. El intérprete de comandos puede hacer esto antes de que los procesos sean `execvp`'d, pero este enfoque no funciona si cualquiera de los procesos que sean `execvp`'d reciben la señal `SIGINT`. También si el usuario mueve el entubamiento de nuevo hacia el frente, entonces el `ctrl-c`

debe ser enviado nuevamente a los procesos en el entubamiento, sin que ninguno de esos procesos realice ninguna acción relacionada con `SIGINT`.

Para poder enviar una señal transparente, POSIX usa las sesiones y el control de terminales. Una *sesión* es una colección de grupos de proceso establecida para propósitos del control de trabajo. Una sesión es identificada por un ID de sesión. Un proceso puede determinar su ID de sesión llamando a `getsid`. Cada proceso pertenece a una sesión, pero un proceso puede cambiar de sesión llamando a `setsid`.

SINOPSIS

```
#include <unistd.h>
#include <sys/types.h>

pid_t getsid(pid_t pid);
```

Spec 1170

SINOPSIS

```
#include <sys/types.h>
#include <unistd.h>

pid_t setsid(void);
```

POSIX.1, Spec 1170

El `setsid` establece ambos, el ID del grupo de proceso y el ID de la sesión del llamador a su ID de su proceso. El intérprete de comandos se convierte en el líder de la sesión y crea una nueva sesión.

La figura 7.2 muestra un intérprete de comandos con varios grupos de proceso. Cada rectángulo sólido representa un proceso. Los ID(es) del proceso, del grupo de proceso y de la sesión están siendo mostrados para cada proceso. Todos los procesos tienen el ID de sesión 1357, que es el ID de proceso y el ID de la sesión del intérprete de comandos. Cada uno de los cuatro trabajos es un grupo de proceso. El ID del grupo de proceso es el mismo que el ID del proceso de uno de sus miembros, el líder del grupo de proceso. Dependiendo de la implementacion del intérprete de comandos, éste puede ser el primero o el último de los procesos en el entubamiento.

Ejemplo 7.7

La siguiente secuencia de comandos debe dar como resultado la estructura del grupo de proceso de la figura 7.2.

```
ls -l | sort -n +4 | grep testfile > testfile.out&
grep process | sort  > process.out &
du . > du.out &
cat /etc/passwd | grep users | sort | head > users.out &
```

Ejercicio 7.11

Escriba un pequeño programa llamado `showid` que tome un argumento de línea de comando, y que despliegue su argumento junto con los ID del proceso, del proceso

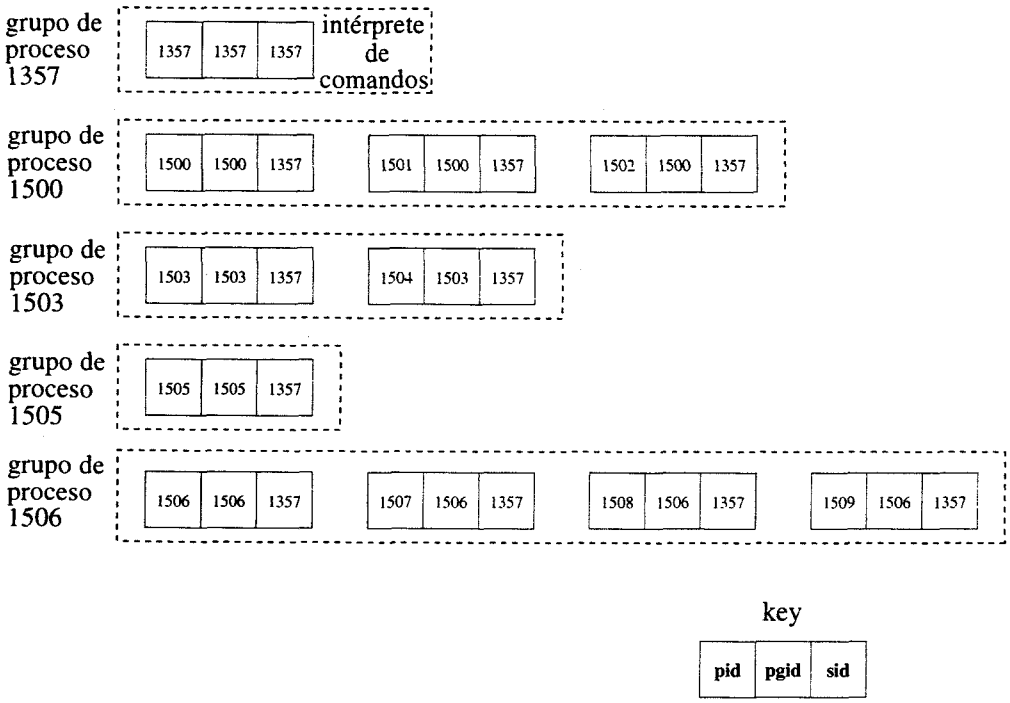


Figura 7.2: Cinco grupos de proceso para la sesión 1357.

padre, del grupo de proceso y de la sesión, todo en una línea, y luego empiece un ciclo infinito. Ejecute los siguientes comandos para verificar cómo su intérprete de comandos de entrada (login) maneja grupos de proceso y sesiones para los entubamientos.

```
showid 1 | showid 2 | showid 3
```

¿Qué proceso en el entubamiento es el líder del grupo de proceso? ¿Está el intérprete de comandos en el mismo grupo de proceso que el entubamiento? ¿Qué procesos en el entubamiento son hijos del intérprete de comandos y cuáles son sus nietos? ¿Qué diferencia habría si el entubamiento es iniciado desde el fondo?

Respuesta:

Los resultados varían dependiendo del tipo de intérprete de comandos utilizados. Algunos intérpretes de comandos generan a todos los procesos hijos del intérprete de comandos. Otros tienen sólo el primero o el último en el entubamiento como un hijo del intérprete de comandos y el resto son nietos. Ya sea el primero o el último puede ser el líder del grupo de proceso. Si un intérprete de comandos no soporta un controlador de trabajo, es posible que el intérprete de comandos sea el líder del grupo de proceso del entubamiento, a menos que el entubamiento sea iniciado en el fondo.

La sesión tiene una terminal controladora, que es la del intérprete de comandos. Cuando mucho un grupo de proceso en la sesión es el grupo de proceso del frente en cualquier momento. Este grupo de proceso recibe las señales generadas por el teclado de alimentación de la terminal controladora. Los otros grupos de proceso son de fondo. Los grupos de proceso de fondo no son afectados por el teclado de alimentación de la terminal controladora de la sesión. Si todos los grupos de proceso están en el fondo, entonces el intérprete de comandos es el proceso frontal del grupo de proceso de la terminal controladora y los otros trabajos son grupos de proceso de fondo. Si alguno de los trabajos es traído hacia el frente, se convierte en el grupo de procesos frontal y el intérprete de comandos se convierte en uno de los grupos de proceso de fondo. Las señales del teclado de alimentación y `SIGINT` generadas cuando se oprime `ctrl-c` en el teclado de la terminal controladora son enviadas sólo a los procesos en los grupos de proceso frontal.

Se dice que un intérprete de comandos tiene control de trabajo cuando permite al usuario mover un grupo de proceso de la parte del fondo hacia la parte frontal. El control de trabajo involucra el cambio de grupo de proceso frontal de una terminal controladora. La función `tcgetpgrp` regresa el ID del proceso de grupo de un proceso de una terminal controladora en particular.

SINOPSIS

```
#include <sys/types.h>
#include <termios.h>

pid_t tcgetpgrp(int fildes);
int tcsetpgrp(int fildes, pid_t pgid);
```

POSIX.1, Spec 1170

SINOPSIS

```
#include <sys/types.h>
#include <termios.h>

pid_t tcgetsid(int fildes);
```

Spec 1170

El `tcsetpgrp` cambia el grupo de proceso asociado con la terminal controladora `fildes`. el `tcgetsid` regresa el ID de la sesión de la terminal asociada con `fildes`.

7.6 Control de procesos de fondo en `ush`

Las principales propiedades de un proceso de fondo son que el intérprete de comandos no espera a que el proceso se complete y que no es terminado por un `SIGINT` enviado desde el teclado. Un proceso de fondo al parecer se ejecuta independiente de la terminal. En esta sección exploraremos cómo es que `ush` debe manejar las señales para procesos de fondo. Un

intérprete de comandos que trabaja correctamente debe prevenir señales generadas por terminal, procura que ninguna alimentación sea enviada a un proceso de fondo y maneja el problema de tener un hijo divorciado de su terminal controladora.

Un signo de *ampersand* (&) al final de un comando le indica a `ush` que debe ejecutar el comando en la parte del fondo. Asumamos que como máximo hay sólo un & en la línea y que, de estar presente, se encuentra al final. El intérprete de comandos debe determinar si este comando se tiene que ejecutar en la parte del fondo antes de bifurcarse hacia el hijo, puesto que ambos deben saber si está en la parte del fondo. Los ejemplos de esta sección enriquecen la versión 2 de `ush` mostrada en el programa 7.3.

El programa 7.11 muestra una modificación de `ush` que permite a un comando ejecutarse en la parte del fondo. Si el comando es ejecutado en la parte del fondo, el hijo llama a `setpgid` para que deje de ser el proceso frontal de su terminal controladora. En este caso, el intérprete de comandos padre no esperará a su hijo.

Programa 7.11: Un intérprete de comandos que trata de manejar procesos de fondo, cambiando a sus grupos de proceso.

```
#include "ush.h"
void main (void)
{
    char inbuf[MAX_CANON];
    pid_t child_pid;
    int inbackground;
    char *backp;

    for( ; ; ) {
       fputs(PROMPT_STRING, stdout);
       if (fgets(inbuf, MAX_CANON, stdin) == NULL)
          break;
       if (*(inbuf + strlen(inbuf) - 1) == NEWLINE_SYMBOL)
          *(inbuf + strlen(inbuf) - 1) = 0;
       if (strcmp(inbuf, QUIT_STRING) == 0)
          break;
       else {
          if ((backp = strchr(inbuf, BACK_SYMBOL)) == NULL)
             inbackground = FALSE;
          else {
             inbackground = TRUE;
             *(backp) = NULL_SYMBOL;
          }
          if ((child_pid = fork()) == 0) {
             if (inbackground)
                if (setpgid(getpid(), getpid())== -1)
```

```
                exit(1);
            executecmdline(inbuf);
            exit(1);
        }
        else if (child_pid > 0 && !inbackground)
            waitpid(child_pid, NULL, 0);
    }
  }
  exit(0);
}
```

Programa 7.11

Ejercicio 7.12

Ejecute el comando `ls &` varias veces bajo el intérprete de comandos en el programa 7.11. Luego ejecute `ps-a` (todavía bajo el mismo intérprete de comandos). Observe que el proceso previo `ls` todavía aparece como `<defunct>`. Salga del intérprete de comandos y ejecute un `ps-a` de nuevo. Explique el estado de estos procesos antes y después de las salidas del intérprete de comandos.

Respuesta:

Puesto que no hay proceso esperando por ellos, los procesos de fondo se convierten en procesos zombies. Se quedarán en esta condición hasta las salidas del intérprete de comandos. A ese tiempo, `init` se convierte en el padre de estos procesos y puesto que `init` periódicamente espera a sus hijos, eventualmente muere.

El intérprete de comandos en el programa 7.12 arregla el problema de los zombies o de los procesos muertos. Cuando un comando se efectúa en el fondo, el intérprete de comandos efectúa un `fork` extra. El primer hijo sale inmediatamente, dejando huérfano al proceso de fondo, que entonces puede ser adoptado por `init`. El intérprete de comandos espera a todos los hijos aun cuando sean de procesos de fondo, puesto que los hijos de fondo salen inmediatamente y los nietos son adoptados por `init`.

Programa 7.12: Un intérprete de comandos que maneja procesos zombies de fondo.

```
#include "ush.h"
void main (void)
{
    char inbuf[MAX_CANON];
    pid_t child_pid;
    char *backp;

    for( ; ; ) {
        fputs(PROMPT_STRING, stdout);
        if (fgets(inbuf, MAX_CANON, stdin) == NULL)
            break;
        if (*(inbuf + strlen(inbuf) - 1) == NEWLINE_SYMBOL)
```

```
            *(inbuf + strlen(inbuf) - 1) = 0;
        if (strcmp(inbuf, QUIT_STRING) == 0)
            break;
        else if ((child_pid = fork()) == 0) {
            if ((backp = strchr(inbuf, BACK_SYMBOL)) != NULL) {
                *(backp) = NULL_SYMBOL;    /* Línea de comando end antes que & /*
                if (fork() != 0) exit(0);
                if (setpgid(getpid(), getpid())== -1)
                    exit(1);
            }
            executecmdline(inbuf);
            exit(1);
        } else if (child_pid > 0)
            waitpid(child_pid, NULL, 0);
    }
    exit(0);
}
```

Programa 7.12

Ejercicio 7.13

Ejecute un largo proceso de fondo tal como lo es `rusers &`, bajo un intérprete de comandos como el dado en el programa 7.12. Haga un `ctrl-c`. Observe cómo el proceso de fondo no es interrumpido aun cuando el intérprete de comandos sale. (Recuerde que la versión del programa en el programa 7.12 no atrapa a `SIGINT`.)

Ejercicio 7.14

Use la función `showid` del ejercicio 7.11 para determinar cuál de los tres procesos en el entubamiento se convierte en el líder del grupo de proceso y cuáles son los hijos del intérprete de comandos para la versión `ush` del programa 7.12. Haga lo anterior para entubamientos que se inician ya sea en el frente o en el fondo.

Respuesta:

Si el entubamiento se inicia en el frente, todos los procesos tienen el mismo grupo de proceso que el intérprete de comandos y éste es el líder del grupo de proceso. El último proceso en el entubamiento es un hijo del intérprete de comandos y los otros son nietos de éste. Si el entubamiento se inicia en el fondo, el último proceso en el tubo es el líder del grupo de proceso y su padre es `init`. Los otros procesos son hijos del último proceso en el entubamiento.

El problema de hijo zombi es más complicado si el intérprete de comandos efectúa control de trabajo. En este caso, el intérprete de comandos debe poder detectar si el proceso de fondo se detiene debido a una señal (por ejemplo, `SIGSTOP`). El `waitpid` cuenta con una opción para detectar qué hijos son detenidos por señales, pero no para detectarlo con nietos. El proceso de fondo del programa 7.12 es un nieto, debido al `fork` extra, por lo que `ush` no lo puede detectar.

Programa 7.13: Un intérprete de comandos que maneja procesos zombies de fondo, usando el `waitpid`.

```
#include "ush.h"
void main (void)
{
    char inbuf[MAX_CANON];
    pid_t child_pid;
    pid_t wait_pid;
    char *backp;
    int inbackground;

    for( ; ; ) {
       fputs(PROMPT_STRING, stdout);
       if (fgets(inbuf, MAX_CANON, stdin) == NULL)
          break;
       if (*(inbuf + strlen(inbuf) - 1) == NEWLINE_SYMBOL)
          *(inbuf + strlen(inbuf) - 1) = 0;
       if (strcmp(inbuf, QUIT_STRING) == 0)
          break;
       else {
          if ((backp = strchr(inbuf, BACK_SYMBOL)) == NULL)
             inbackground = FALSE;
          else {
             inbackground = TRUE;
             *(backp) = NULL_SYMBOL;
          }
          if ((child_pid = fork()) == 0) {
             if (inbackground)
                if (setpgid(getpid(), getpid()) == -1) exit(1);
             executecmdline(inbuf);
             exit(1);
          }
          else if (child_pid > 0) {
             if (!inbackground)
                while((wait_pid = waitpid(-1, NULL, 0)) > 0)
                   if (wait_pid == child_pid) break;
             while (waitpid(-1, NULL, WNOHANG) > 0)
                ;
          }
       }
    }
    exit(0);
}
```

Programa 7.13

El programa 7.13 muestra un enfoque directo para manejar zombies usando `waitpid`. Para poder detectar si hay procesos de fondo que hayan sido detenidos por una señal, esta versión

de ush usa waitpid con el WNOHANG para procesos de fondo, en lugar de bifurcar a un hijo extra. El –1 para el primer argumento de waitpid significa esperar cualquier proceso. Si el comando no es de fondo, ush explícitamente espera que el hijo correspondiente termine.

Ejercicio 7.15

Repita el ejercicio 7.14 del programa 7.13.

Respuesta:

Los resultados son los mismos que en el ejercicio 7.14, excepto que el último proceso en el entubamiento es un hijo del intérprete de comandos.

7.7 Control de trabajo

El control de trabajo le permite a los usuarios que en forma selectiva paren procesos y que reinicie su ejecución más tarde, cuando se desee. Uno de los propósitos del control de trabajo es el de permitir a los usuarios que ejecuten programas largos en el fondo y periódicamente los paren para verificar el estado o proveerlos de algún tipo de alimentación. El intérprete de comandos C (C-shell) y el Korn permiten tener control de trabajo, pero el intérprete de comandos Bourne no lo permite. Esta sección describe control de trabajo para un intérprete de comandos C (C-shell). El intérprete de comandos Korn es casi idéntico en lo que respecta a control de trabajo.

Un trabajo consiste en los procesos necesarios para ejecutar una sola línea de comando. Cuando el intérprete de comandos inicia un trabajo en el fondo, asigna un número de trabajo y despliega los ID de proceso de todos los procesos en el trabajo. Si un entubamiento es iniciado en el fondo, todos los procesos en el entubamiento tienen el mismo número de trabajo. El número de trabajo generalmente es un número entero pequeño. Si no existen trabajos ejecutándose en el fondo, el intérprete de comandos asigna al comando el número de trabajo de uno. En general, el intérprete de comandos asigna a un trabajo en el fondo el número de trabajo igual al máximo de los trabajos, dejando actuales e incrementado a 1.

El comando jobs despliega los trabajos que se están ejecutando bajo el intérprete de comandos.

Ejemplo 7.8

Los siguientes comandos ilustran el control de trabajo para el intérprete de comandos C (shell). El intérprete de comandos despliega el indicador ospmt%. *Los comandos aparecen después de este indicador. El intérprete de comandos produce los mensajes que se muestra a continuación.*

```
ospmt% du . | sort -n > duout &
[1] 23145 23146
ospmt% grep mybook *.tex > mybook.out &
[2] 23147
ospmt% rusers | grep myboss > myboss.out &
[3] 23148 23149
ospmt% jobs
[1]   + Running          du . | sort -n > duout
[2]   - Running          grep mybook *.tex > mybook.out
[3]     Running          rusers | grep myboss > myboss.out
```

El comando `jobs` *muestra tres trabajos de fondo que están siendo ejecutados. El número de trabajo está al principio de la línea entre paréntesis cuadrados. Si el segundo trabajo termina primero, el intérprete de comandos desplegará la siguiente línea después de que el usuario haya presionado la tecla de retorno.*

```
[2]    Done                grep mybook *.tex > mybook.out
```

Si en el momento el usuario ejecuta otro comando `jobs`*, se desplegarán las siguientes líneas.*

```
[1]  + Running             du . | sort -n > duout
[3]  - Running             rusers | grep myboss > myboss.out
```

El ejemplo 7.8 muestra que cuando un intérprete de comandos empieza un trabajo en el fondo, despliega el número de trabajo seguido de los ID de proceso, de los procesos correspondientes a ese trabajo. Refiérase al trabajo `n` como `%n`.

Ejemplo 7.9

El siguiente comando elimina el trabajo dos sin referirse a los ID del proceso.

```
kill -KILL %2
```

El ejemplo 7.8 muestra un trabajo precedido por a +, lo cual significa que éste es el *current job* y que por omisión no se usa ningún número después de `%`. El - representa al trabajo anterior. Nótese que el trabajo actual es el primero en empezarse y no el último.

Un trabajo de fondo puede ser ejecutado o detenido. Para detener un trabajo que está siendo ejecutado, use el comando `stop`. El trabajo detenido se convierte en el trabajo actual y éste es suspendido.

Ejemplo 7.10

El siguiente comando detiene el trabajo dos.

```
stop %2
```

Para reiniciar un trabajo que se ejecutaba en el fondo y ha sido detenido, use el comando `bg`. En este caso, `bg` o `bg %` o `bg %2`, todos trabajan, puesto que el trabajo dos es el trabajo actual.

Use el comando `fg` para mover un trabajo de fondo (ya sea que esté ejecutándose o detenido) hacia la parte frontal, y el carácter `SIGSTOP` (normalmente `ctrl-z`) para desplazar un trabajo frontal hacia la parte del fondo, cuando éste se encuentre parado. La combinación `ctrl-z` y `bg` hace que un trabajo frontal se convierta en un trabajo de fondo ejecutable.

Los comandos `fg`, `bg` y `jobs` normalmente no tienen sus propias páginas en el manual, puesto que estos comandos ya están integrados dentro del intérprete de comandos. Para obtener información sobre estos comandos en el intérprete de comandos C (C-shell), ejecute `man csh`.

Ejercicio 7.16

Experimente con el control de trabajo (asumiendo que está disponible). Mueva proceso de fondo a plano primario (frontal) y viceversa.

Un intérprete de comandos que soporta un control de trabajo debe seguir la pista a todos los grupos de proceso frontales o de fondo que pertenezcan a esa sesión. Cuando la terminal genera una interrupción SIGSTOP (normalmente como respuesta a un ctrl-z), el grupo de proceso frontal es colocado en estado de paro (stopped state). ¿Cómo debe el intérprete de comandos volver a estar bajo control? Afortunadamente el waitpid bloquea al intérprete de comandos padre, hasta que el estado de uno de sus hijos cambie. Así, una apropiada llamada a waitpid hecha por el intérprete de comandos padre permite a este último regresar a estar bajo control, después de que un grupo de proceso frontal es suspendido. El intérprete de comandos puede iniciar un grupo de proceso mandándole la señal de SIGCONT. Si el intérprete de comandos quiere reiniciar ese grupo frontal, tiene que usar tcsetpgrp para avisarle a la terminal controladora qué significa el grupo de proceso frontal. Si un proceso o grupo de proceso puede ejecutarse en diferentes tiempos durante su ejecución en la parte frontal o en la parte del fondo, cada comando hijo debe iniciar un nuevo grupo de proceso sin importar si éste es iniciado como un proceso frontal o un proceso de fondo.

Un problema del control de trabajo que todavía no hemos mencionado es el concerniente a cómo un proceso obtiene su alimentación desde la entrada estándar. Si este proceso se ejecuta en plano primario, no existe ningún problema. Si no existe control de trabajo y el proceso es iniciado en el fondo, su entrada estándar es redirigida a /dev/null, para prevenir que sean tomados caracteres de la entrada estándar del proceso frontal. Esta simple redirección no trabaja con control de trabajo. Una vez que la entrada estándar es redirigida, resulta difícil obtener alimentación de la terminal controladora original cuando el proceso es traído como proceso en el plano primario. La solución especificada por POSIX es que el núcleo genere una señal SIGTTIN cuando un proceso de fondo trate de leer de una terminal controladora. El manejador por omisión para SIGTTIN detiene el trabajo. El intérprete de comandos detecta un cambio en el estado del hijo cuando éste da un waitpid y despliega un mensaje. El usuario entonces tiene la opción de mover el proceso hacia el frente para que se pueda recibir la alimentación.

En POSIX.1, los trabajos de fondo pueden escribir al error estándar. Si tratan de escribir hacia la salida estándar (y la salida estándar es todavía la terminal controladora), el controlador de la terminal genera un SIGTTOU para el proceso, si el miembro c_lflag del struct termios de la terminal tiene activa la bandera TOSTOP. Si es así, el usuario tiene la opción de mover el trabajo hacia el plano primario para que pueda mandar una salida a la terminal controladora. Si el proceso ha redirigido la entrada y la salida estándares, hará la E/S desde las fuentes redirigidas.

Ejercicio 7.17

Escriba un programa simple que a su vez escriba a la salida estándar. Empiece en la parte del fondo y vea si puede escribir a la salida estándar sin que se genere una señal SIGTTOU.

7.8 Control de trabajo para ush

Esta sección describe la implementación de un control de trabajo para ush. Empiece combinando el manejo de señales del programa 7.10 con el manejo de procesamiento de fondo del

programa 7.13, para producir un intérprete de comandos que maneje correctamente `SIGINT` y `SIGQUIT`. Pruebe completamente el programa para los siguientes casos.

- Comandos simples.
- Comandos incorrectos.
- Comandos con las entradas y salidas estándares redirigidas.
- Entubamientos.
- Procesos de fondo.
- Todos los anteriormente mencionados interrumpidos por un `ctrl-c`.

7.8.1 Objeto lista de trabajo

Para poder efectuar control de trabajos, `ush` debe seguir la pista de sus hijos. Use un objeto lista similar al usado en el programa 2.4 para mantener una historia del programa. Los nódulos en la lista deben tener la siguiente estructura:

```
typedef enum jstatus
        {FOREGROUND, BACKGROUND, STOPPED, DONE, TERMINATED}
    job_status_t;

typedef struct job_struct {
    char *cmdstring;
    pid_t pgid;
    int job;
    job_status_t jobstat;
    struct job_struct *next_job;
} joblist_t;

static joblist_t *job_head = NULL;
static joblist_t *job_tail = NULL;
```

Coloque la estructura de la lista en un archivo aparte junto con las siguientes funciones para manipular la lista de trabajos:

- La función `add_list` adiciona el trabajo especificado a la lista. El prototipo de `add_list` es:

  ```
  intt add_list (pid_pgid, char *cmd, job_status_t status) ;
  ```

  El `pgid` es el ID del grupo de proceso y el `cmd` es la hilera de comandos para el trabajo. El valor de `status` puede ser `FOREGROUND` o `BACKGROUND`. La función `add_list` regresa un número de trabajo si se ejecuta correctamente, o un `-1` si hay alguna falla.
- La función `delete_list` elimina el nodo correspondiente al trabajo especificado de la lista. El prototipo para `delete_list` es:

  ```
  int delete_list(int job);
  ```

El delete_list regresa un número de trabajo si el nodo es correctamente borrado o un -1 si existe alguna falla. Asegúrese de que libere todo el espacio asociado con el nodo borrado.

- La función show_jobs muestra una lista de trabajos y el estado en que se encuentra cada uno de ellos. Use un formato similar a:

```
[job]   status      pgid   cmd
```

- La función set_status establece el valor del nodo para cada trabajo correspondiente, ya sea a FOREGROUND o a BACKGROUND. Su prototipo es:

```
int set_status(int job, job_status_t status);
```

La función set_status regresa un 0 si se ejecuta correctamente o un -1 si hay alguna falla.

- La función get_status regresa el valor asociado con el trabajo especificado. Su prototipo es:

```
int get_status(int job, job_status_t *pstatus);
```

La función get_status regresa un 0 si se ejecuta correctamente o un -1 si hay alguna falla.

- La función get_process regresa el ID del grupo de proceso del trabajo especificado. El prototipo para get_process es:

```
pid_t get_process(int job) ;
```

Si job no existe, get_process regresa un 0.

- El get_largest_job_number recorre la lista de trabajos para encontrar el mayor número que existe en esa lista. El prototipo para get_largest_job_number es:

```
int get_largest_job_number(void) ;
```

El get_largest_job_number regresa el mayor número de trabajo si existe cualquier nodo en la lista, o un 0 si la lista está vacía.

Escriba un programa controlador para probar exhaustivamente la lista de funciones, independientemente de ush.

7.8.2 La lista de trabajo en ush

Después de probar el funcionamiento correcto de las funciones, añada el objeto lista de trabajos como sigue:

- Cada vez que ush bifurca a un hijo para ejecutar un proceso de fondo, adiciona un nodo a la lista de trabajo. Establece al miembro pgid del nodo joblist_t al valor que regresa fork. El estado del proceso es BACKGROUND. Llama a get_largest_job_number para determinar el mayor número de trabajo que actualmente está en los procesos de fondo actuales y asigna al nuevo proceso de fondo un número de trabajo mayor que 1.

- Si el comando es ejecutado en el fondo, `ush` muestra un mensaje de la siguiente forma:

```
[job]      pid1  pid2  ...
```

donde `job` es el número de trabajo y `pid1`, `pid2`, y así sucesivamente, son los ID de los procesos hijos en el grupo de proceso del comando. El padre de `ush` solamente conoce el ID del hijo inicial, por lo que el hijo que ejecuta el `executecmdline` debe producir el mensaje.

- El `ush` llama a la función `show_jobs`, cuando un usuario alimenta el comando `jobs`.

- Reemplace la llamada `waitpid` en `ush`, con un enfoque más elaborado, usando `waitpid` en un ciclo con la opción `WUNTRACED`. La opción `WUNTRACED` especifica que `waitpid` debe reportar el estado de cada hijo detenido cuyo estado no haya sido reportado. Este reporte es necesario para la implementacion del control de trabajo en la siguiente etapa.

SINOPSIS

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

POSIX.1, Spec 1170

Use `-1` para el parámetro `pid`. Si el comando está siendo ejecutado en el plano primario, realice ciclos hasta que el hijo frontal complete su ejecución. Si el comando está siendo ejecutado en el fondo, use la opción `WNOHANG` además de `WUNTRACED`. Cuando el intérprete de comandos correctamente espere a algún hijo, se debe utilizar las siguientes macros para determinar el estado del hijo:

```
WIFSTOPPED (status)   /* diferente de cero si el hijo está
                         detenido */

WIFSIGNALED (status)  /* número de señal si el hijo ha sido
                         terminado por una señal que no fue
                         atrapada */
WIFEXITED(status)     /* diferente de cero si el hijo ha
                         terminado normalmente */
```

El `status` es un valor de un número entero al que está apuntando `stat_loc` en el `waitpid`.

- Cuando un hijo termina normalmente (`WIFEXITED` regresa un valor diferente de cero), o termina debido a una señal (`WIFSIGNALED` regresa un valor diferente de cero), el `ush` elimina el nodo correspondiente de la lista de trabajo.

Ponga a prueba `ush` con la lista de trabajo. No aplique control de trabajo en este paso. Ejecute frecuentemente el comando `jobs` para ver el estado de los procesos de fondo. Cuidadosamente experimente con un intérprete de comandos existente que tenga control de trabajo (por ejemplo, intérprete de comandos C o KornShell). Asegúrese de que `ush` maneja procesos en plano primario y de fondo de manera similar.

7.8.3 Control de trabajo en `ush`

Incorpore control de trabajo en `ush` aplicando los siguientes comandos a `ush` además del comando de `jobs` visto en la sección anterior:

`stop`	Detiene el trabajo actual
`bg`	Inicia la ejecución del trabajo actual en el fondo
`bg %n`	Inicia la ejecución del trabajo `n` en el fondo
`fg %n`	Inicia la ejecución del trabajo `n` en el plano primario
`mykill -SIGNUM %n`	Manda la señal `SIGNUM` al trabajo `n`

Algunos de estos comandos se refieren al trabajo actual. Cuando existen muchos trabajos, uno de ellos es el *trabajo actual*; éste empieza como el primer trabajo que es iniciado *en la parte del fondo*. Un usuario puede hacer que otro trabajo sea el trabajo actual trayéndolo al plano primario con un `fg`.

El `ush` debe manejar `SIGCONT`, `SIGTSTP`, `SIGTTIN` y `SIGTTOU`, además de `SIGINT` y `SIGQUIT`. Cuando `ush` detecta que un hijo ha sido detenido debido a `SIGTTIN` o a `SIGTTOU`, escribe un mensaje informativo al error estándar, indicando que el hijo está esperando por una entrada o salida, respectivamente. El usuario puede mover el trabajo hacia la parte del fondo para que así pueda leer o escribir desde la terminal controladora.

Pruebe el programa exhaustivamente. Preste particular atención a cómo efectúa el control de trabajo y ajusta `ush` el intérprete de comandos C para que se vea lo más parecido posible.

7.9 Lecturas adicionales

Un libro elemental en programación de intérpretes de comandos en C es *Unix Shell Programming,* escrito por Arthur [4]. El libro *Learning the Kern Shell* escrito por Rosenblatt [74] es una clara referencia al intérprete de comandos KornShell. Otro libro acerca de KornShell es *The New KornShell Command and Programming Language,* segunda edición y escrito por Bolsky y Korn [12]. El libro reciente, *Using* `csh` y `tsch` escrito por DuBois [28] parece ser un buen libro técnico de referencia.

Parte III

Concurrencia

Capítulo 8

Secciones críticas y semáforos

Los programas que manejan recursos compartidos deben ejecutar de manera totalmente independiente porciones de código fuente llamadas *secciones críticas*. En este capítulo se discutirá la protección de secciones críticas mediante el uso de semáforos. Además de presentar una noción de semáforo, en este capítulo se verán los semáforos POSIX.1b y System V.

Imaginemos un sistema de computación cuyos usuarios comparten una sola impresora en la cual imprimen simultáneamente. ¿Cómo sería la impresión? Es lógico que si las líneas impresas por el trabajo de cada usuario fueran intercaladas, el sistema no serviría para nada. Los equipos periféricos, como las impresoras, son llamados *recursos de uso exclusivo*, pues deben ser accesados por un proceso a la vez. Los procesos que comparten accesos a diversos recursos deben ser ejecutados de *manera completamente independiente.*

Una *sección crítica* es un segmento de código que debe ejecutarse en forma por completo independiente. El código que modifica una variable compartida, normalmente consta de las siguientes partes:

Sección de entrada:	Es el código que solicita el permiso para modificar la variable compartida.
Sección crítica:	Es el código que modifica la variable compartida.
Sección de salida:	Es el código que libera el acceso.
Sección restante:	Es el código restante.

Cuando se habla del problema de la sección crítica, se hace referencia a aquel que presenta el hecho de ejecutar secciones críticas correcta y simétricamente. Las soluciones al problema de sección crítica deben satisfacer cada uno de los siguientes elementos:

Completa independencia:	Cuando mucho sólo puede estar en su sección crítica un proceso a la vez.
Progreso:	Si no se está ejecutando ningún proceso en su sección crítica, si alguno desea entrar, puede hacerlo. Sólo aquellos procesos

que no están en su sección restante pueden participar en la decisión y determinación de cuál es el próximo proceso que entrará a su sección crítica.

Límites de espera: Ningún proceso puede ser pospuesto indefinidamente. Por otro lado, después de que un proceso haya requerido entrar a su sección crítica, debe haber un número máximo de veces en que a aquéllos les sea permitido entrar a sus sesiones críticas.

El programa 8.1 contiene una modificación del programa 2.12. El programa genera una cadena de procesos. Después de salir de un ciclo de bifurcación, cada proceso manda un mensaje informativo, un carácter a la vez, hacia la salida de error estándar. Puesto que la salida de error estándar es compartida por todos los procesos en la cadena, esa parte de código es una sección crítica y debe ser ejecutada en forma completamente independiente. La sección crítica del programa 8.1 no está protegida, por lo que se intercalan al azar y en forma distinta salidas de diferentes procesos en cada ejecución.

Programa 8.1: Un programa que genera una cadena de procesos.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <unistd.h>

void main  (int argc, char *argv[])
{
    char buffer[MAX_CANON];
    char *c;
    int i;
    int n;
    pid_t childpid;

    if ( (argc != 2) || ((n = atoi(argv[1])) <= 0) ) {
       fprintf (stderr, "Usage: %s number_of_processes\n", argv[0]);
       exit(1);
    }

    for (i = 1; i < n;  ++i)
       if (childpid = fork())
          break;

    sprintf(buffer,
       "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
       i, (long)getpid(), (long)getppid(), (long)childpid);
```

```
    c = buffer;
       /*************principio de la sección crítica*****************/
    while (*c != '\0') {
       fputc(*c, stderr);
       c++;
    }
    fputc('\n', stderr);
       /*****************fin de la sección crítica*******************/

    exit(0);
}
```

Programa 8.1

Cada proceso del programa 8.1 ejecuta órdenes en forma secuencial, pero las órdenes (y por tanto la salida) de los diferentes procesos pueden ser intercaladas de forma arbitraria. Una analogía de este intercalado arbitrario puede ser un paquete de barajas. Parta el paquete de barajas una vez. Piense que cada sección representa un proceso. Cada baraja dentro de una sección representa una orden que está ejecutando el proceso. Ahora baraje ambas secciones. Es lógico que exista una infinidad de posibilidades en que puedan quedar ordenadas las barajas, dependiendo de los mecanismos utilizados para barajar. De la misma manera, existen muchas posibilidades de intercalamiento de las órdenes de los dos procesos debido a que el tiempo exacto de procesos relativo a otros depende de factores externos (por ejemplo, cuántos otros procesos están requiriendo del uso de la CPU o cuánto tiempo permanece un proceso en estado de bloqueo esperando una entrada/salida [E/S]. El reto para los programadores es desarrollar programas que funcionen para todas las posibles maneras en que puedan sucederse las órdenes del programa.

8.1 Operaciones atómicas

Una vez iniciada una *operación atómica,* finaliza en forma lógicamente indivisible (por ejemplo, sin tener ningún otro tipo de instrucciones intercaladas). La mayoría de las soluciones al problema de sección crítica se basan en la existencia de ciertas operaciones atómicas.

Las instrucciones individuales no pueden ser interrumpidas en muchas de las máquinas que tienen sólo un procesador, y por lo tanto son atómicas. Sin embargo, es muy importante que lea el manual de referencia (Hardware Manual) de cada máquina antes de concluir que sus instrucciones son de operación atómica. Puesto que las máquinas RISC proveen una cantidad muy limitada de tipos de operación en memoria, es posible que una simple instrucción en C no sea compilada como una instrucción de máquina de tipo atómica. Por ejemplo, las transferencias de memoria a memoria y aquellas operaciones en donde se efectúa operaciones aritméticas en una dirección de memoria, normalmente no son atómicas. En las máquinas donde diversas CPU comparten memoria, una implementación efectiva de operaciones atómicas es mucho más complicada.

Ejemplo 8.1

El siguiente segmento de código representa una implementacion típica de una orden C `counter++`.

```
a:|   R1 <-- counter
b:|   R1 <-- R1 + 1
c:|   counter <-- R1
```

En el ejemplo 8.1, `R1` representa un registro de máquina. Mientras cada una de las órdenes *a, b* y *c* es indivisible, otras instrucciones pueden ser ejecutadas entre *a* y *b*, si una señal ocurre o si el proceso pierde el uso de la CPU en un instante inadecuado. Si el resto de las instrucciones modifican a `counter`, el programa quizá arroje resultados inconsistentes.

Dejemos que a_1 sea el proceso uno que ejecuta la orden *a*. Un intercalado para procesos puede ser representado por una secuencia de tales valores. Entonces a_1, b_1, a_2 significa que el proceso uno ejecutó las órdenes *a* y *b*, y que el proceso dos ejecutó la orden *a*. El código no sería inválido si fallara cualquier tipo de intercalado.

Ejemplo 8.2

Los siguientes son dos tipos de intercalado del pseudocódigo del ejemplo 8.1.

$$a_1, b_1, c_1, a_2, b_2, c_2$$
$$a_1, b_1, a_2, b_2, c_2, c_1$$

Supongamos que `counter` en el ejemplo 8.1 tiene un valor inicial de 1. Después de la primera secuencia del ejemplo 8.2, `counter` tiene un valor final de 3, mientras que después de la segunda secuencia tiene un valor final de 2, por lo que la ejecución de las tres órdenes en el ejemplo 8.1 no produce resultados consistentes. La falla en el ejemplo 8.1 no tiene nada que ver con el hecho de que los dos procesos se ejecutaron utilizando los mismos registros. Cuando el intercambio de contexto ocurre, se guardan los registros. El mismo problema ocurre si una variable local reemplaza a `R1`.

Un método para implementar una independencia completa es tener una simple variable compartida, e indicar si es seguro entrar en una sesión crítica. La variable representa un *lock* y una variable con un valor de 1 indicaría que un proceso está ejecutando una porción de código en la sección crítica. Un valor cero indicaría que es seguro entrar a la sección crítica.

Ejemplo 8.3

El siguiente pseudocódigo ilustra una manera incorrecta de implementar independencia completa. La estrategia utiliza la variable compartida `lock` *que en un principio se asigna un valor de cero.*

```
a:|   while (lock)
b:|        ;
c:|   lock = 1;
d:|      <critical section>
e:|   lock = 0;
```

Un proceso ejecuta el ciclo `while` del ejemplo 8.3 siempre y cuando el valor de `lock` sea 1. Cuando `lock` se convierta en 0, el proceso colocará a `lock` en 1 para evitar que otros procesos entren en la sección crítica y vaciará a `lock` cuando se complete la sección crítica. El método de cerradura (locking) del ejemplo 8.3 falla si dos procesos están en su ciclo de `while`, cuando `lock` se convierte en 0 y un proceso pierde la CPU en el lapso que transcurre entre el momento de salir del ciclo de `while` y cuando se está en proceso de asignar un valor a `lock`.

Ejemplo 8.4

El siguiente intercalamiento de órdenes del ejemplo 8.3 permite a los procesos uno y dos ejecutar secciones críticas al mismo tiempo.

$$a_1, b_1, a_2, b_2, c_2, d_2, c_1, d_1, e_1, e_2$$

La implementación y depuración (debugging) de secciones críticas puede ser difícil, pues es posible que los intercalados que propicien fallas no ocurran con frecuencia ni se repitan continuamente. Por ello es necesario ser muy cuidadoso cuando se utilicen variables compartidas. Aun sin variables compartidas explícitas, un programa puede tener secciones críticas problemáticas. Un problema de intercalado que por desgracia se presencia con frecuencia es el uso de variables externas en los manejadores de señales. También los sistemas no entrantes (nonreentrant) y las funciones de biblioteca realizan una interacción inadecuada con los manejadores de señales.

Ejercicio 8.1

El ejemplo 8.3 no satisface el requerimiento de independencia absoluta. ¿Qué otros requerimientos del problema de sección crítica no son satisfechos por este ejemplo?

Respuesta:

Este ejemplo no cumple con los límites de espera.

El ejemplo 8.3 carece de una ejecución consistente debido a que la prueba de `lock` y la asignación de valores a `lock` no son atómicas. En el caso de un sistema de una sola CPU, el problema que representa no cumplir con los límites de espera del ejemplo 8.3 puede ser corregido al prevenir la pérdida de la CPU durante los intervalos de prueba y de asignación de un valor de la variable *lock*. Para casos de CPU múltiples, el inhabilitar todos los interruptores no es suficiente, puesto que un proceso que está siendo ejecutado en otra CPU puede modificar una variable compartida en cualquier momento.

Muchos sistemas proveen una instrucción `TestAndSet` atómica, o una instrucción `Swap` atómica para manejar este problema. La instrucción `TestAndSet` inicia una variable a 1 y regresa el anterior valor de la variable. `Swap` tiene un efecto más general que `TestAndSet`. Posee dos argumentos e intercambia sus valores automáticamente.

Ejemplo 8.5

La siguiente función C implementa una operación similar a `TestAndSet`.

```
int TestAndSet(int *target)
{
    int returnval;
```

```
    returnval = *target;
    *target = 1;
    return returnvalue;
}
```

El punto importante es que la función `TestAndSet` debe ser ejecutada de manera atómica, aun cuando se esté trabajando en un sistema de multiproceso. Si las instrucciones `TestAndSet` son ejecutadas simultáneamente (por diferentes CPU en la misma máquina), esto sucede en forma secuencial. Si son ejecutadas concurrentemente (dos procesos en un ambiente de multitareas), ninguna es interrumpida por la otra.

Ejercicio 8.2

¿Qué efecto tiene la siguiente orden si la variable `lock` tiene un valor 0 antes de que la orden sea ejecutada?

```
mytemp = TestAndSet(&lock);
```

Respuesta:

Después de la ejecución, `mytemp` es 0 y `lock` es 1.

Ejercicio 8.3

¿Qué efecto tiene la siguiente orden si la variable `lock` tiene un valor 1 antes de que la orden se ejecute?

```
while (TestAndSet(&lock))
   ;
```

Respuesta:

Si el valor de `lock` no es modificado en ninguna otra parte, el proceso entraría en un ciclo interminable.

Ejemplo 8.6

El siguiente pseudocódigo utiliza la función `TestAndSet` *para proteger una sección crítica. La variable* `lock` *es iniciada a 0 antes de que ningún proceso ejecute el segmento.*

```
a:|  while(TestAndSet(&lock))
b:|     ;
c:|  <critical section>
d:|  lock = 0;
e:|  <remainder section>
```

El primer proceso que ejecuta el pseudocódigo `TestAndSet` en el ejemplo 8.6 provoca que `lock` sea iniciado a 1. La función `TestAndSet` regresa el valor antiguo de `lock` (que es 0), para así terminar el ciclo de `while`, y el primer proceso ejecuta su sección crítica. Si algún otro proceso ejecuta el ciclo de `while` mientras `lock` tiene un valor de 1, el ciclo continuará puesto que `TestAndSet` es regresado al valor de 1. Eventualmente el primer proceso termina y asigna otra vez a `lock` el valor a 0. El siguiente proceso que ejecuta `TestAndSet` recibe en-

tonces un valor de regreso cero y cesa la ejecución del ciclo de `while`. En el ejemplo 8.6 se asume que la orden *d* es atómica.

Ejercicio 8.4

Suponga que los procesos uno y dos están ejecutando sus secciones críticas como se especifica en el ejemplo 8.6. ¿Es la siguiente expresión una posible intercalación de las instrucciones para estos procesos?

$$a_1, a_2, c_2, c_1, d_1, e_1, d_2, e_2$$

Respuesta:

El intercalamiento no es posible. Después de a_1, el valor de `lock` es 1. Por lo tanto, la función `TestAndSet` en a_2 debe regresar a 1, por lo que el proceso dos no puede salir del ciclo `while`.

Ejercicio 8.5

Observe la solución de `TestAndSet` presentada en el ejemplo 8.6. ¿Resuelve el problema de sección crítica?

Respuesta:

La respuesta es no. La solución provee independencia exclusiva pero no satisface la necesidad de límite de espera.

Ejercicio 8.6

Supongamos que tanto `key` como `lock` tengan un valor de 1. ¿Qué efecto tendría el siguiente código? ¿Qué pasaría si `lock` fuera inicializada con un valor de 0 en lugar de 1?

```
while(key == 1)
   Swap(&lock, &key);
```

Respuesta:

Si `lock` fuera inicialmente 1, el proceso se estancaría en el ciclo de `while`. Si `lock` fuera inicialmente 0, el proceso cesaría el ciclo después de haber intentado la primera iteración.

Ejemplo 8.7

El siguiente pseudocódigo utiliza a `Swap` *para proteger una sección crítica. La variable compartida* `lock` *tiene inicialmente el valor 0.*

```
key = 1;
while(key == 1)
   Swap(&lock, &key);
<critical section>
lock = 0;
<remainder section>
```

El primer proceso que ejecuta el ciclo `while` en el ejemplo 8.7 se detiene después de la primera iteración, puesto que `lock` tiene un valor de 0. Los procesos subsecuentes se estancarán en el ciclo `while` mientras el primer proceso no termine de ejecutar su sección crítica y mientras no restablezca a `lock` un valor de 0. Cuando el valor de `lock` sea 0 y uno de los

procesos ejecuta el Swap, entraría dentro del ciclo de while. Recuerde que Swap es atómica, por lo que sólo un proceso a la vez ve el valor de lock en 0 y sigue adelante.

La implementación de TestAndSet en el ejemplo 8.6 y la implementación de Swap en el ejemplo 8.7 utiliza la señal de *ocupado en espera (busy waiting)*, con lo que se entiende que un proceso continuamente ejecuta una operación para determinar si éste puede ser procesado. Estas pruebas continuas utilizan ciclos de máquina que no realizan ningún trabajo productivo. Algo más conveniente consiste en bloquear el proceso hasta que éste pueda proceder.

La función TestAndSet no es una solución al problema de sección crítica que esté al alcance del usuario, debido a que utiliza el método de ocupado en espera, y éste depende de la máquina. El sistema operativo usa instrucciones como TestAndSet para proveer una atomicidad para sincronizaciones primitivas de alto nivel, tales como *semáforos, contadores de eventos o variables condicionales*. El resto de este capítulo estará dedicado a los semáforos. La sección 8.2 nos presenta el concepto de abstracción de un semáforo. La sección 8.3 analiza el nuevo POSIX.1b para semáforos y la sección 8.4 estudia la versión antigua del Sistema V para semáforos. El capítulo 10 explora mecanismos adicionales de sincronización, como las variables condicionales que son parte de la especificación de hilos del POSIX.1c.

8.2 Semáforos

En 1965, E.W. Dijkstra [25] propuso el concepto de abstracción de un semáforo para el manejo de independencia exclusiva y sincronización a alto nivel. Un semáforo es un número entero variable con dos operaciones atómicas: wait y signal. Otros nombres para wait son down, P y lock. Otros nombres para signal son up, V, unlock y post.

Un proceso que ejecuta un wait en un semáforo variable s no puede proceder hasta que el valor de s sea positivo. Es entonces cuando decrementa el valor de s. La operación de signal incrementa el valor del semáforo variable. En la terminología de POSIX.1b [53], estas operaciones son llamadas *cerrojo de semáforo* y *apertura de semáforo*.

En esta sección se asume que los semáforos variables son del tipo semaphore_t, éste siempre contiene un número entero variable que un proceso siempre está en posibilidad de probar, incrementar o reducir, a través de las operaciones asociadas de semáforo. En algunas ocasiones el semáforo es solamente un número entero variable, y en otras existe una estructura más compleja dependiendo del tipo de implementación de que se trate. Por ahora, pensemos que el semaphore_t es como un int.

Ejemplo 8.8

El siguiente pseudocódigo implementa una serie de operaciones de semáforo.

```
void wait(semaphore_t *sp)
{
   while(*sp <= 0)
      ;
   (*sp)--;
}
```

```
void signal(semaphore_t *sp)
{
   (*sp)++;
}
```

Las operaciones de `wait` y `signal` mostradas en el ejemplo 8.8 deben ser atómicas. En este contexto el termino atómico significa que si el llamador llega al estado de reducir el `wait`, ningún otro proceso tratará de cambiar al semáforo variable en el intervalo de la prueba final en el `while` y la terminación del proceso de reducción. Naturalmente que el código C especificado arriba para `wait` no funcionará correctamente, puesto que no está presentado en forma atómica.

Ejemplo 8.9

El siguiente pseudocódigo protege una sección crítica si el semáforo variable `S` *es iniciado a 1.*

```
wait(&S);
<critical section>
signal(&S);
<remainder section>
```

Los procesos que utilizan semáforos deben cooperar para asegurar que se obtenga una independencia exclusiva. El código en el ejemplo 8.9 protege a una sección crítica siempre y cuando todos los procesos ejecuten el `wait(&S)` antes de entrar a sus respectivas secciones críticas, y siempre y cuando ejecuten la `signal(&S)` al salir. Si cualquier proceso falla en ejecutar el `wait(&S)`, ya sea por un error o por un descuido, los procesos pueden no ser ejecutados cumpliendo con el requisito de independencia exclusiva. Si algún proceso no ejecuta la `signal(&S)` cuando termina su sección crítica, otros procesos cooperativos serán bloqueados cuando traten de entrar a sus secciones críticas.

Ejercicio 8.7

¿Qué sucedería si en el ejemplo anterior S inicialmente tuviera un valor de 0? ¿Qué pasaría si `S` tuviera un valor inicial de 8?

Respuesta:

Si `S` inicialmente tuviera un valor de 0, entonces cada `wait(&S)` sería bloqueado y se establecería una espera indefinida (dead lock), a menos que algún otro proceso inicializara `S` a 1. Si `S` inicialmente tuviera un valor de 8, entonces sólo un máximo de ocho procesos se ejecutarían concurrentemente en sus secciones críticas.

Ejemplo 8.10

Supongamos que el proceso uno debe ejecutar la orden a_1 *antes de que el proceso dos ejecute la orden* b_2. *El semáforo* `sync` *dirigirá el ordenamiento del siguiente pseudocódigo, asumiendo que* `sync` *está inicialmente en 0.*

```
Process 1 executes:          Process 2 executes:
   a;                           wait(&sync);
   signal(&sync);               b;
```

Puesto que el sync utilizado en el ejemplo 8.10 tiene un valor inicial de 0, el proceso dos está bloqueado en su wait hasta que el proceso uno ejecuta su signal.

Ejercicio 8.8

¿Qué sucedería si en el siguiente pseudocódigo los semáforos S y Q tuvieran inicialmente el valor 1?

```
Process 1 executes:
   for( ; ; ) {
      wait(&S);
      a;
      signal(&Q);
   }
```

```
Process 2 executes:
   for( ; ; ) {
      wait(&Q);
      b;
      signal(&S);
   }
```

Respuesta:

Cualquiera de ambos procesos puede ejecutar su orden de wait primero. Los semáforos se encargan de que cualquier proceso específico no esté más adelante de una iteración que el otro proceso. Si un semáforo está inicialmente en 1 y el otro está en 0, los procesos procederán a ejecutarse en forma estrictamente alterna. Si los dos semáforos están inicialmente en 0, entonces ocurre una espera infinita (deadlock).

Ejercicio 8.9

¿Qué sucedería en el ejercicio 8.8 si S estuviera iniciado a 8 y Q a 0? Ayuda: Piense que S está representado como ranuras en el almacenamiento temporal y Q como unidades de un almacenamiento temporal.

Respuesta:

El proceso uno siempre estará entre cero y ocho iteraciones adelante del proceso dos. Si el valor de S representa ranuras vacías y el valor de Q representa elementos en esas ranuras, el proceso uno adquiere ranuras y produce elementos, mientras que el proceso dos adquiere elementos y produce ranuras vacías. Esta generalización sincroniza el acceso a un almacenamiento temporal que puede acomodar no más de ocho elementos.

Ejercicio 8.10

¿Qué pasaría con el siguiente pseudocódigo si los semáforos S y Q tuvieran un valor inicial de 1?

```
Process 1 executes:
   for( ; ; ) {
      wait(&Q);
      wait(&S);
      a;
      signal(&S);
      signal(&Q);
   }
```

```
Process 2 executes:
   for( ; ; ) {
      wait(&S);
      wait(&Q);
      b;
      signal(&Q);
      signal(&S);
   }
```

Respuesta:

El resultado dependería del orden en que los procesos llegaran a la CPU. Este código debiera trabajar la mayor parte del tiempo, pero si el proceso uno pierde a la CPU después de la ejecución de wait(&Q) y el proceso dos logra entrar, ambos procesos se bloquean en su segundo wait y ocurre una espera infinita (deadlock).

8.2.1 Implementación de semáforos con `TestAndSet`

Los sistemas operativos utilizan `TestAndSet` en *hardware,* o su equivalente, para asegurar la atomicidad de estructuras de alto nivel sincronizadas (por lo menos en sistemas de una sola CPU). Sin embargo, los semáforos y otro tipo de mecanismos de sincronización de alto nivel son más convenientes. Los semáforos son independientes de la arquitectura del sistema, por lo que el programador no necesita saber detalles arquitectónicos de la máquina de destino. Adicionalmente, un proceso del usuario no puede bloquear directamente un proceso pues requiere de una llamada del sistema para obtener el permiso del núcleo (kernel), por lo que no es posible que un programa llame directamente a `TestAndSet` y después se bloquee a sí mismo (por lo menos no poniéndose directamente en la lista del nivel apropiado del núcleo). Puesto que el programa requiere una llamada de sistema para bloquear, entonces es preferible que el `TestAndSet` sea también parte de las llamadas de sistema. En otras palabras, el uso de `TestAndSet` para asegurar una independencia exclusiva sin la intervención del núcleo implica la implementación de un tipo de ocupado en espera.

El resto de esta sección se dedicará a la implementación de un tipo de ocupado en espera para semáforos mediante el uso de `TestAndSet`. El principal punto de discusión es el hecho de que `TestAndSet` debe utilizarse como un cerrojo sólo para periodos cortos de tiempo.

Ejercicio 8.11

¿Cuál es el error en la siguiente implementación de `wait` y `signal`? Asuma que `lock` tiene un valor inicial de 0 y `*sp` un valor inicial de 1.

```
      void wait(semaphore_t *sp)
      {
a:|      while(TestAndSet(&lock))
b:|         ;
c:|      while((*sp) <= 0)
d:|         ;
e:|      (*sp)--;
f:|      lock = 0;
      }

     void signal(semaphore_t *sp)
     {
g:|      while(TestAndSet(&lock))
h:|         ;
i:|      (*sp)++;
k:|      lock = 0;
     }
```

Respuesta:

Supongamos que el proceso dos llama a `wait`, encuentra que `*sp` es igual a 1 y lo reduce a 0, para luego regresar del `wait` para ejecutar su sección crítica. El valor de `lock` fue por algún tiempo 1, pero ahora es 0 nuevamente. Ahora supongamos que el proceso uno ejecuta el `wait`. El proceso uno está en un ciclo indefinido en las órdenes *c* y *d* con `lock` teniendo un valor de 1. Cuando el proceso dos llama a `signal`, se quedará suspendido en las órdenes *g* y *h*, puesto que `lock` esta siendo detenido por el

proceso uno. El resultado es una espera indefinida (deadlock). Un ejemplo de una secuencia de intercalamiento en donde el proceso uno ejecuta un `wait` y el dos bloquea con un `signal` es:

$$a_1, b_1, c_1, d_1, c_1, d_1, g_2, h_2, g_2, h_2$$

La implementación en el ejercicio 8.11 falla puesto que `wait` detiene a `lock` por mucho tiempo (no sólo el tiempo suficiente para realizar una tarea sino todo el tiempo que está esperando la variable del semáforo). La implementación de `signal` es correcta puesto que detiene a `lock` lo suficiente para incrementar la variable del semáforo. Una implementación correcta detiene a `lock` el tiempo suficiente para asegurar su atomicidad.

Ejemplo 8.11

La siguiente implementación de un semáforo en términos de `TestAndSet` *no causa una espera indefinida.*

```
      void wait(semaphore_t *sp)
      {
a:|      for( ; ; ) {
b:|         while(TestAndSet(&lock))
c:|            ;
c:|         if (*sp > 0) {
d:|            (*sp)--;
e:|            break;   /* sale del ciclo */
f:|         }
g:|         lock = 0;
h:|      }
i:|      lock = 0;
      }

      void signal(semaphore_t *sp)
      {
j:|      while(TestAndSet(&lock))
k:|         ;
l:|      (*sp)++;
m:|      lock = 0;
      }
```

El `wait` del ejemplo 8.11 inicia a `lock` con valor 1, hace una sola prueba para ver si el semáforo es positivo y reinicia `lock` a 0. Para esta implementacion, la variable del semáforo puede ser un simple `int`. Esta implementación usa el ocupado esperando.

8.2.2 Semáforos sin ocupado esperando

Una implementación que utiliza y confía en el uso de un ocupado esperando es ineficiente y no garantiza el límite de espera. Una mejor implementación bloquea el proceso cuando debe

esperar. Cuando un proceso inicia su espera, el sistema operativo lo coloca en una línea de espera especial de procesos en espera de un semáforo y, de igual manera, en una línea de procesos en espera de E/S (entrada/salida). El sistema operativo maneja esta línea de espera (digamos, como una línea de espera FIFO [primeras entradas/primeras salidas]) y elimina un proceso cuando ocurre una `signal` del semáforo. Un posible tipo de implementación lo representa la estructura `semaphore_t`, `S`, que consta de un número entero miembro (designado por `S.value`) y ligado con una lista (designada `S.list`) de procesos en espera. El número entero miembro representa el número de recursos disponibles.

Ejemplo 8.12

El siguiente pseudocódigo muestra una implementación de bloqueo de semáforos.

```
void wait(semaphore_t *sp)
{
   if (sp->value > 0)
      sp->value--;
   else {
      <Agregue este proceso a sp->list>
      <bloquea>
   }
}
void signal(semaphore_t *sp)
{
   if (sp->list != NULL)
   <elimine un proceso de sp->list y póngalo en estado listo para ejecución>
 else
      sp->value++;
}
```

Recuerde que tanto `wait` y `signal` deben ser atómicas, y en el ejemplo 8.12 no se muestra el código extra necesario para asegurar su atomicidad. El tipo de ordenamiento usado para la lista del semáforo determina cuál es el siguiente proceso que debe ser atendido. Una lista de espera tipo FIFO garantiza límites de espera. Otras técnicas para el manejo de líneas de espera pueden dar como resultado que un proceso sea completamente ignorado. A este problema se le conoce como *inanición.*

8.2.3 Sincronización *AND*

El semáforo sincroniza los procesos al requerir que el valor de la variable del semáforo sea siempre mayor que 0. En la práctica existen más formas generales de sincronización que las que hemos analizado hasta ahora (por ejemplo, las variables condicionales permiten la sincronización en cualquier tipo de condición), y existen también mecanismos para combinar las condiciones de sincronización. La *sincronización OR* se refiere al esperar hasta que cualquier condición en un grupo específico sea satisfecha. El uso de `select` o `poll` para observar múltiples descriptores de archivos de entrada es una forma de sincronización OR. La sincronización NOT se refiere a esperar hasta que una condición en el grupo no sea verdadera, y puede usarse para asegurar que se cumpla una orden según una prioridad preestablecida [64].

La *sincronización AND* se refiere al esperar hasta que todas las condiciones en un grupo sean satisfechas. La sincronización AND puede ser utilizada para el control simultáneo de múltiples recursos. Esta subsección explora el uso y la implementación de semáforos simultáneos que son una forma de sincronización AND. En el Sistema V UNIX de grupos de semáforos que discutimos en la sección 8.4, usamos una forma de sincronización AND.

Ejemplo 8.13

Considere un sistema que tiene dos unidades de cinta marcadas A y B. El proceso uno periódicamente vacía hacia la cinta utilizando la unidad A. El proceso dos copia de la unidad A en la unidad B. El proceso tres periódicamente vacía hacia la cinta usando la unidad B. El siguiente pseudocódigo correctamente sincroniza las interacciones de los tres procesos usando simples semáforos, `A` y `B`, los cuales están iniciados a 1.

```
Process 1:              Process 2:                Process 3:
   wait(&A);               wait(&A);                 wait(&B);
   <use tape A>            wait(&B);                 <use tape B>
   signal(&A);             <use tapes A and B>       signal(&B);
                           signal(&B);
                           signal(&A);
```

Desafortunadamente, el proceso dos del ejemplo 8.13 puede detener la cinta A mientras espera que la cinta B esté disponible. Por lo tanto, la cinta A está desocupada si el proceso tres se encuentra activo y el proceso dos está bloqueado. Una mejor solución al ejemplo 8.13 es hacer que el proceso espere a ambas unidades de cinta antes de atender cualquiera de ellas. La notación `wait(&A, &B)` denota una *espera simultánea* en dos semáforos `A` y `B`. El proceso bloquea si así fuera necesario en cualquier semáforo individualmente, o lo que es lo mismo, no reduce la variable del semáforo a menos que reduzca ambas variables de los semáforos sin bloquearlos.

Ejemplo 8.14

El siguiente pseudocódigo muestra el uso de espera simultánea para el problema de la unidad de cinta del ejemplo 8.13.

```
Process 1:              Process 2:                Process 3:
   wait(&A);               wait(&A, &B);             wait(&B);
   <use tape A>            <use tapes A and B>       <use tape B>
   signal(&A);             signal(&A, &B);           signal(&B);
```

Cuando un proceso ejecuta una espera simultánea, prueba en forma atómica todos los semáforos en el grupo para ver si alguno pudiera causar un bloqueo al proceso. Si no se ocasionara bloqueo, el proceso en forma atómica reduciría las variables de los semáforos y continuaría. Si cualquiera de las operaciones de decremento causara algún bloqueo, el proceso se bloquearía sin modificar los valores de los semáforos.

Cuando el proceso ejecuta una señal de semáforo simultánea, se activan todos los procesos bloqueados en los semáforos. Los nuevos procesos desbloqueados tratan otra vez de efec-

tuar sus reducciones. De esta manera los procesos desbloqueados compiten para pasar a través de la espera simultánea. El ejemplo 8.15 muestra un pseudocódigo para manejar un `wait` con dos variables simultáneas. Compare la implementación de dos variables con la de una variable implementada en la sección 8.2.2.

Ejemplo 8.15

En el siguiente pseudocódigo de la implementación de un `wait` *con dos variables simultáneas, el proceso aparece cuando mucho en una lista. El* `wait` *debe ser ejecutado en forma atómica.*

```
void wait(semaphore_t *ap, semaphore_t *bp)
{
   if ((ap->value > 0) && (bp->value > 0)){
       (ap->value)--;
       (bp->value)--;
   }
   else {
      if (ap->value <= 0)
         <Agregue este proceso a ap->list>
      else
         <Agregue este proceso a ap->list>
      <restablezca el contador del programa a la condición
       del comienzo de espera>
      <bloqueo>
   }
}
```

La función `signal` con dos variables de semáforo simultáneas activa todos los procesos que están esperando en cualquier semáforo. Puesto que cada proceso trata de readquirir sus semáforos requeridos, sólo el proceso cuyos semáforos están disponibles continúa mientras que el resto de los procesos son bloqueados nuevamente.

Ejemplo 8.16

El siguiente pseudocódigo implementa una señal simultánea de semáforo.

```
void signal(semaphore_t *ap, semaphore_t *bp)
{
    ap->value++;
    <mover todos los procesos en ap->list a la cola de procesos listos para
     ejecución>
    bp->value++;
    <mover todos los procesos en bp->list a la cola de procesos listos para
     ejecución>
}
```

La señal simultánea de semáforo del ejemplo 8.16 despierta a todos los procesos cuando los semáforos son incrementados. Este enfoque parece ser ineficiente, pero los métodos alternativos de implementación son muy complicados.

Supongamos en vez de esto que las esperas simultáneas ponen al proceso en todas las colas de espera y que la señal simultánea sólo despierta el primer proceso en espera que está en cada cola. Estos procesos podrían estar esperando a otros semáforos y no hay la garantía de que sea seleccionado el proceso correcto. El ejemplo 8.12 muestra estos problemas usando una implementación alternativa.

Ejercicio 8.12

Los semáforos `A`, `B` y `C` son inicializados a 0. ¿Cómo podría fallar el siguiente pseudocódigo, si la implementación de la señal simultánea sólo despierta el primer proceso en cada cola de espera del semáforo?

```
Process 1                          Process 2
a:| wait(&A, &B);                  a:|  wait(&B, &C);
b:| <critical section>             b:|  <critical section>
c:| signal(&A, &B);                c:|  signal(&B, &C);

Process 3
a:| signal(&B, &C);
```

Respuesta:

Si el orden de ejecución es a_1 seguido de a_2, las colas de espera del semáforo son:

```
A:  1
B:  1, 2
C:  2
```

Si la señal simultánea del semáforo despierta únicamente el primer proceso en espera en cada cola, la orden a_3 elimina al proceso uno de la cola de espera `B` y al proceso dos de la cola de espera `C`.

```
A:  1
B:  2
C:
```

El proceso uno detiene al semáforo `B` mientras bloquea al semáforo `A`. El proceso dos permanece bloqueado hasta que el proceso uno obtiene el semáforo `A`, violando así el propósito de obtención simultánea.

Ambos, `wait` y `signal` simultáneos, se pueden generalizar para cualquier número de semáforos. Un programa no debe llamar un `signal` con el mismo número de semáforos que un `wait`. Un `signal` despierta todos los procesos que están esperando en sus correspondientes semáforos. Puesto que los procesos que han sido despertados de nuevo compiten por los recursos del semáforo en forma descontrolada, es posible que un proceso sin suerte nunca adquiera sus recursos del semáforo.

8.3 Semáforos en POSIX

Los semáforos son parte del estándar POSIX.1b adoptado en 1993 [53]. Debido a que esta es una novedad, es posible que los semáforos POSIX no estén disponibles en todos los sistemas

operativos que dicen ser compatibles con POSIX.1. Una implementación de semáforos es soportada por POSIX si ésta define a `_POSIX_SEMAPHORES` en `unistd.h`.

El semáforo POSIX.1b es un tipo variable de `sem_t` con operaciones atómicas tales como iniciación, incrementación, reducción de su valor. El estándar POSIX.1b define dos tipos de semáforos: nombrado y no nombrado. Un *semáforo* POSIX.1b *no nombrado* puede ser usado por un solo proceso o por los hijos que el proceso haya creado. Un *semáforo* POSIX.1b *nombrado* puede ser usado por cualquier proceso. La diferencia entre los semáforos no nombrados y los nombrados es análoga a la que existe entre entubamientos ordinarios y entubamientos con nombre (FIFO).

Ejemplo 8.17

El siguiente segmento de código declara una variable de semáforo llamada `sem`.

```
#include <semaphore.h>
sem_t sem;
```

El estándar POSIX.1b no especifica la naturaleza del tipo de `sem_t`. Una posibilidad es que éste actúa como un descriptor de archivo y es un desplazamiento en una tabla local. Los puntos de entrada de la tabla son entradas a una tabla de archivos del sistema. Una implementación en particular puede no usar el modelo de una tabla de descriptor de archivo o una tabla de archivos del sistema, por lo que la información acerca del semáforo puede ser almacenada mediante la variable `sem` y todas las funciones del semáforo toman como parámetro un apuntador a la variable del semáforo.

Todas las funciones del semáforo POSIX.1b regresan un valor de -1 e inician `errno` para indicar error. Uno de los posibles valores de `errno` para `sem_init` es `ENOSYS`, indicando que `sem_init` no está siendo soportada por una implementación. (`_POSIX_SEMAPHORES` puede ser definido, pero el sistema de momento no es capaz de soportarlo usando semáforos POSIX.1b.) El número de semáforos que el sistema puede soportar es limitado, y si ese límite es sobrepasado, el valor de `errno` es `ENOSPC`. POSIX no indica cuáles operaciones del semáforo deben retornar en caso de éxito, pero el estándar establece que en una versión futura estos valores deberán regresar a 0.

8.3.1 Iniciación de semáforos no nombrados

Los semáforos POSIX.1b son semáforos contadores que tienen valores no negativos y que deben iniciarse antes de ser usados. En la sección 8.3.3 se discuten técnicas de iniciación para semáforos nombrados. La función `sem_init` inicia un semáforo no nombrado.

SINOPSIS

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
```

POSIX.1b

La función `sem_init` inicia al semáforo para que contenga un valor `value`. El parámetro `value` no puede ser negativo. Si el valor de `pshared` no es 0, el semáforo puede ser usado entre procesos (por ejemplo, por el proceso que lo inicia y por los hijos de ese proceso). De otra manera sólo puede ser usado por los hilos dentro del proceso que lo inicia.

Piense que `sem` se refiere al semáforo, en lugar de pensar que éste es el semáforo en sí. De hecho, la función `sem_init` crea el semáforo e inicia a `sem` para referirse a éste. Como resultado, si `pshared` no es 0, los hijos heredan semáforos de la misma manera que heredan descriptores de archivos abiertos.

8.3.2 Operaciones de semáforos POSIX

Las siguientes funciones manipulan semáforos no nombrados y nombrados después de su iniciación.

SINOPSIS

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
```

POSIX.1b

La función `sem_destroy` destruye un semáforo previamente iniciado. Si un proceso intenta destruir un semáforo que tiene otro proceso esperándolo, `sem_destroy` puede regresar un mensaje de error iniciando `errno` a `EBUSY`. Desafortunadamente la especificación de POSIX1.b no requiere que el sistema detecte esto.

El `sem_wait` es una operación estándar de espera de semáforo. Si el valor del semáforo es 0, entonces `sem_wait` efectúa un bloqueo hasta el punto en que se pueda reducir el valor del semáforo o hasta que éste sea interrumpido por una señal tal como `SIGINT`. En la sección 8.5 se discute este tema con mayor amplitud. La función `sem_trywait` es similar a la función `sem_wait`, excepto que en lugar de bloquear cuando se intente reducir el valor de un semáforo a cero, regresa un valor de –1 e inicia `errno` a `EAGAIN`.

La función `sem_post` incrementa el valor del semáforo y es la clásica señal de operación de un semáforo. El estándar POSIX.1b requiere que `sem_post` sea reentrante con respecto a las señales, o lo que es lo mismo, que sea una señal asíncrona segura y que pueda ser llamada desde un manejador de señales.

La función `sem_getvalue` le permite al usuario que examine el valor de un semáforo no nombrado o nombrado. La función es iniciada por `sval` al referenciar el número entero al valor del semáforo. Si existen procesos esperando el semáforo, el estándar de POSIX.1b permite que esta función inicialice a `sval`, ya sea con un 0 o con un número negativo cuyo valor ab-

soluto representa el número de procesos que están esperando el semáforo, en cualquier momento durante el llamado de `sem_getvalue`. Esta ambigüedad hace que no pueda usarse este recurso. Un valor positivo de `sval` representa el valor del semáforo en cualquier momento de la ejecución de `sem_getvalue`, pero no necesariamente en el momento en que `sem_getvalue` regresa. La función regresa a 0 en caso de éxito. En caso de error, `sem_getvalue` regresa un valor de –1 e inicia `errno`.

El programa 8.2 muestra una modificación del programa 8.1, que protege la sección crítica por medio de un semáforo POSIX.1b, llamado `my_lock`. El código inicia a `my_lock` a 1 para proporcionar independencia exclusiva.

Programa 8.2: Un programa que protege a una sección crítica mediante el uso de un semáforo.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <unistd.h>
#include <semaphore.h>

void main  (int argc, char *argv[])
{
    char buffer[MAX_CANON];
    char *c;
    int i;
    int n;
    pid_t childpid;
    sem_t my_lock;

    if ( (argc != 2) || ((n = atoi(argv[1])) <= 0) ) {
       fprintf (stderr, "Usage: %s number_of_processes\n", argv[0]);
       exit(1);
    }
    if (sem_init(&my_lock, 1, 1) == -1) {
       perror("No puede iniciar el semáforo mylock");
       exit(1);
    }
    for (i = 1; i < n;  ++i)
       if (childpid = fork())
          break;

    sprintf(buffer,
       "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
       i, (long)getpid(), (long)getppid(), (long)childpid);

    c = buffer;
       /**********************sección de entrada**********************/
```

```
    if (sem_wait(&my_lock) == -1)  {
       perror("Semáforo no válido")
       exit(1);
    }
       /***************principio de sección crítica*******************/
    while (*c != '\0') {
       fputc(*c, stderr);
       c++;
    }
    fputc('\n', stderr);
       /******************fin de sección crítica *********************/

       /*********************sección de salida************************/
    if (sem_post(&my_lock) == -1) {
       perror("Semáforo ejecutado")
       exit(1);
    }
       /******************* sección remanente ***********************/
    exit(0);
}
```

Programa 8.2

8.3.3 Semáforos nombrados

Los semáforos nombrados POSIX.1b pueden sincronizar procesos que no tienen ningún ancestro en común. Los semáforos nombrados poseen un nombre, una identificación de usuario, una identificación de grupo y accesos como los tienen los archivos. El nombre de un semáforo es una cadena de caracteres que cumple con las condiciones de la construcción de un nombre de ruta. POSIX.1b no requiere que su nombre aparezca en el manejador de archivos, ni tampoco especifica las consecuencias de tener a dos procesos referiéndose al mismo nombre, a menos que el nombre empiece con un cáracter diagonal. Si el nombre empieza con una diagonal (/), entonces dos procesos (o hilos) que abren al semáforo con ese nombre se estarán refiriendo al mismo semáforo. Por lo tanto, siempre use nombres que empiecen con una diagonal (/) para nombrar semáforos POSIX.

La función `sem_open` establece la conexión entre el semáforo nombrado y el valor de `sem_t`. Existen dos formas de `sem_open`.

SINOPSIS

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode,
                unsigned int value);
```

POSIX.1b

Si la llamada es exitosa, sem_open regresa un indicador que identifica a las llamadas futuras del semáforo en sem_wait, sem_trywait, sem_post, sem_destroy y sem_getvalue. Nótese la analogía entre los semáforos nombrados y los valores de sem_t con nombres de archivo y descriptores de archivos. Recuérdese que las funciones del semáforo utiliza indicadores para valores de sem_t.

El valor de oflag determina si la función sem_open accesa un semáforo previamente creado o crea uno nuevo. Una oflag de valor 0 especifica la primera forma y sem_open regresa una asa a un semáforo previamente abierto y que tiene el mismo nombre. Si tal semáforo no ha sido abierto anteriormente, sem_open regresa un -1 e inicia errno a ENOENT.

Si oflag tiene un valor de O_CREAT o de O_CREAT | O_EXCL requiere de la segunda forma de sem_open, que contiene dos parámetros adicionales. El tercer parámetro especifica los permisos del semáforo si es que se está creando, en la misma forma en que ocurre el llamado de sistema open. El cuarto parámetro especifica el valor inicial del semáforo si éste está siendo creado.

Si oflag es O_CREAT, sem_open crea un semáforo si es que no existe alguno. Si ya existe un semáforo del mismo nombre, la función sem_open accesa al semáforo previamente creado. En el último caso, sem_open ignora el tercero y cuarto argumentos.

Si oflag es O_CREAT | O_EXCL, sem_open crea un semáforo si es que no existe todavía. Si ya existe un semáforo con el mismo nombre, sem_open regresa un -1 e inicia errno a EEXIST. En POSIX.1b sólo se definen los siguientes valores de 0 para oflag O_CREAT y O_CREAT | O_EXCL. Otros valores son dependientes del sistema.

Existen dos funciones adicionales que pueden usarse con semáforos nombrados, el sem_close y el sem_unlink.

SINOPSIS

```
#include <semaphore.h>

int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

POSIX.1b

Ambos, sem_close y sem_unlink, son regresados a 0 en caso de éxito. En caso de error, estas funciones regresan -1 e inicializan errno. Aun cuando POSIX.1b no requiere que los semáforos nombrados correspondan a las entradas del sistema de archivos, es conveniente pensar en esta representación y comparar sem_close y sem_unlink con el close y el unlink de las llamadas del sistema.

Cuando se termina un proceso en un semáforo nombrado, éste llama a sem_close para desasignar todos los recursos del sistema que fueron asignados por el llamador para este semáforo. La función sem_close no necesariamente elimina el semáforo nombrado, pero lo vuelve inaccesible para el proceso. Las llamadas de sistema _exit y exec también desasignan recursos para procesos de semáforo.

La función sem_unlink elimina un semáforo nombrado del sistema. Si otros procesos todavía hacen referencia al semáforo, sem_unlink pospone la destrucción hasta que todas las

referencias sean cerradas por `sem_close`, `_exit` o `exec`. Las llamadas con el mismo nombre a `sem_open` se refieren a un nuevo semáforo después de haber ejecutado un `sem_unlink`, aun cuando algunos otros procesos todavía tengan un semáforo antiguo abierto. El `sem_unlink` siempre regresa inmediatamente, aun cuando otros procesos tengan el semáforo abierto.

POSIX.1b no especifica en dónde se asignan los recursos para semáforos POSIX. Los semáforos de System V, analizados en la siguiente sección, son un ejemplo de semáforos a nivel núcleo (kernel), puesto que sus recursos residen en el núcleo.

8.4 Semáforos en System V (Spec 1170)

Los semáforos System V son parte de las características de intercomunicación de procesos (IPC) de System V, que también incluye la facilidad de memoria compartida (sección 8.9) y colas de espera de mensajes (sección 8.10). Un proceso crea un semáforo System V al ejecutar una llamada de sistema `semget`. La llamada crea una estructura de datos del semáforo en el núcleo y regresa un número entero para manejar al semáforo. Los procesos no pueden accesar la estructura de datos del semáforo en forma directa, sólo a través de llamadas del sistema. Los identificadores del semáforo o manijas son similares a los descriptores de archivos.

Los semáforos, la compartición de memoria y las colas de mensajes del System V, no son parte de POSIX.1, pero están incluidos en la especificación Spec 1170. Sus estructuras de datos son creadas y guardadas en el núcleo y son referenciadas a través del manejo de asas números enteros. En contraste, un programa declara una variable de tipo `sem_t` y pasa un indicador a esa variable cuando ésta es requerida por funciones del semáforo POSIX.

8.4.1 Grupos de semáforos

Un semáforo en UNIX System V es realmente un *grupo de semáforos* consistente en un anexo de *elementos de semáforo*. Los elementos del semáforo corresponden a los clásicos semáforos de número entero propuesto por Dijsktra. Un proceso puede ejecutar operaciones en todo el grupo con una sola llamada del sistema.

La representación interna de los grupos de semáforos y los elementos individuales de los semáforos no son directamente accesibles, pero cada elemento de un semáforo incluye por lo menos lo siguiente:

- Un número entero no negativo que representa el valor del elemento del semáforo.
- La identificación ID del último proceso que manipuló el elemento del semáforo.
- El número de procesos que está esperando el incremento del valor del elemento del semáforo.
- El número de procesos que está esperando que el valor del elemento del semáforo sea igual a 0.

Las operaciones del semáforo permiten a un proceso bloquear hasta que el valor del elemento del semáforo sea 0, o hasta que éste se convierta en positivo. Cada elemento tiene dos colas de espera asociadas con éstos: una cola de procesos esperando el incremento del valor del elemento del semáforo y la otra cola de procesos esperando que el valor del elemento sea igual a 0.

8.4.2 Creación de semáforos

El `semget` crea un grupo de semáforos e inicia cada elemento a 0.

SINOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg);
```

Spec 1170

El `semget` toma tres argumentos: una llave que identifica al grupo de semáforos, el tamaño del grupo de semáforos y las banderas. El `semget` regresa una asa número entero para llamadas subsecuentes de `semop` y `semctl`. El `semget` regresa un `-1` e inicia `errno` en caso de que la llamada falle. Puede ocurrir una falla si los argumentos no son válidos o si existen suficientes recursos del sistema asignables al semáforo.

El parámetro `key` indica cuál es el grupo particular de semáforos que va a ser creado o accesado. Un programa especifica una llave de una de las siguientes tres maneras: usando `IPC_PRIVATE` y haciendo que el sistema desarrolle una llave; escogiendo al azar una llave número o usando `ftok` para que genere una llave de un nombre de ruta.

El parámetro `nsems` especifica el número de elementos en el grupo de semáforos. Los elementos individuales dentro de un grupo de semáforos son referenciados por los números enteros del 0 hasta `nsems -1`. Los semáforos tienen permisos especificados por el argumento `semflg` de `semget`. Los valores de los permisos se especifican de la manera descrita en la sección 3.3.1 para archivos, y el cambio de permisos se hace llamando a `semctl`. Si un proceso intenta crear un semáforo que ya existe, recibe una manija para el semáforo existente a menos que se especifique un valor de `semflg` que incluya a ambos, `IPC_CREAT` e `IPC_EXCL`. En el último caso, `semget` falla e inicia a `errno` con un valor igual a `EEXIST`.

Ejemplo 8.18

El siguiente segmento de código crea un grupo privado de semáforos que contiene tres elementos de semáforo.

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
#define PERMS S_IRUSR|S_IWUSR
#define SET_SIZE 3

int semid;

if ((semid = semget(IPC_PRIVATE, SET_SIZE, PERMS)) < 0)
    perror("Could not create new private semaphore")
```

El valor de PERMS del ejemplo 8.18 especifica que el semáforo sólo puede ser leído o escrito por el dueño. La llave IPC_PRIVATE garantiza que semget creará un nuevo semáforo. Para obtener un nuevo grupo de semáforos a partir de una llave hecha o de una llave derivada de un nombre de ruta, el proceso debe especificar que está creando un nuevo semáforo mediante el uso de la bandera IPC_CREAT.

Ejemplo 8.19

El siguiente segmento de código crea un grupo de semáforos con un solo elemento identificado por el valor de la llave 99887.

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#include <errno.h>

#define PERMS S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
#define SET_SIZE 1
#define KEY ((key_t)99887)

int semid;
if ((semid = semget(KEY, SET_SIZE, PERMS | IPC_CREAT)) < 0)
   fprintf(stderr, "Error creating semaphore with key %d: %s\n",
      (int)KEY, strerror(errno));
```

Dar un valor específico a una llave permite que procesos cooperativos estén de acuerdo en un grupo de semáforos común. Los permisos en el ejemplo 8.19 permiten que todos los procesos de usuario accesen el semáforo. Si el semáforo ya existe, el semget regresa una manija al semáforo existente. Reemplaza el argumento semflg de semget con PERMS | ICP_CREAT | IPC_EXCL, y semget regresa un mensaje de error si ya existe el semáforo.

La tercera manera de identificar al grupo de semáforos es derivando una llave a partir de un nombre de ruta mediante el llamado de ftok. El archivo debe existir y ser accesible a los procesos que deseen accesar el semáforo. La combinación del nombre de ruta path y del número entero id identifican de una manera única al semáforo. El parámetro id permite que varios grupos de semáforos sean identificados de un único nombre de ruta.

SINOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t ftok(const char *path, int id);
```

Spec 1170

El programa 8.3 crea un grupo de semáforos con dos elementos a partir de una llave derivada. Los parámetros `path` e `id` son argumentos de línea de comando.

Programa 8.3: Un programa que crea un semáforo a partir de un nombre de ruta.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#include <errno.h>

#define PERMS S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH
#define SET_SIZE 2

void main(int argc, char *argv[])
{
   int semid;
   key_t mykey;

   if (argc != 3) {
      fprintf(stderr, "Usage: %s filename id\n", argv[0]);
      exit(1);
   }
   if ((mykey = ftok(argv[1], atoi(argv[2]))) == (key_t) -1) {
      fprintf(stderr, "Could not derive key from filename %s: %s\n",
            argv[1], strerror(errno));
      exit(1);
   }
   else if ((semid = semget(mykey, SET_SIZE, PERMS | IPC_CREAT)) < 0) {
      fprintf(stderr, "Error creating semaphore with key %d: %s\n",
            (int)mykey, strerror(errno));
      exit(1);
   }
   printf("semid = %d\n", semid);
   exit(0);
}
```

Programa 8.3

8.4.3 Operaciones de semáforo System V

Un proceso puede incrementar, reducir o probar en forma individual los elementos de un semáforo para determinar un valor cero, mediante el uso de un llamado de sistema `semop`.

SINOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop(int semid, struct sembuf *sops, int nsops);
```

Spec 1170

El parámetro `semid` es la manija regresada por `semget` y el parámetro `sops` apunta a un conjunto de elementos de operación. El parámetro `nsops` especifica el número de elementos de operación en el arreglo `sops`. Si `semop` es interrumpido por una señal, regresa un `-1` con `errno` iniciado a `EINTR`. Una implementación de independencia exclusiva realizada con semáforos System V realizada con cuidado reinicia la función `semop` si es que ha sido interrumpida por una señal.

Todas las operaciones especificadas en `struct sembuf` son ejecutadas en forma atómica, en un grupo único de semáforos. Si cualquiera de los elementos individuales de operación provocara que el proceso se bloqueara, entonces ninguna de las operaciones sería ejecutada.

La estructura `struct sembuf` especifica un elemento de operación de un semáforo e incluye los siguientes miembros:

`short sem_num:` El número del elemento del semáforo.
`short sem_op:` La operación en particular que va a ser ejecutada en el elemento del semáforo.
`short sem_flg:` Las banderas para especificar las opciones de la operación.

El elemento de operaciones `sem_op` son los valores que indican la cantidad por la cual se puede cambiar el valor del semáforo:

- Si `sem_op` es un número positivo, entonces `semop` suma el valor al valor correspondiente al elemento del semáforo y despierta todos los procesos que están esperando que el elemento sea incrementado.
- Si `sem_op` es igual a 0 y el elemento del semáforo no tiene un valor de 0, entonces `semop` bloquea todos los procesos llamadores (esperando un 0) e incrementa el contador de procesos que está esperando que el elemento tenga un valor de cero.
- Si `sem_op` es un número negativo, entonces `semop` suma al valor del elemento correspondiente del semáforo, un valor cuyo resultado no sea negativo. Sin embargo, si la operación da como resultado un valor del elemento negativo, entonces `semop` bloquea al proceso en caso de que el valor del elemento del semáforo sea incrementado. Si el valor del resultado es 0, entonces `semop` despierta los procesos que están esperando un 0.

En la descripción de semop se asume que el valor de sem_flg es 0. Si sem_flg & IPC_NOWAIT es verdadero, el llamado nunca bloquea pero en su lugar regresa un -1 con errno inicializado a EAGAIN. Si sem_flg & SEM_UNDO es verdadero, la función también modifica el valor de ajuste del semáforo para el proceso. Este valor de ajuste permite al proceso *deshacer* su efecto sobre el semáforo cuando éste sale.

Ejemplo 8.20

Una práctica poco conveniente consiste en iniciar struct sembuf *en una declaración. El siguiente código C declara la estructura* struct sembuf *como* myopbuf *y la inicia de tal manera que* sem_num *es 1,* sem_op *es* 1 *y* sem_flg *es* 0.

```
struct sembuf myopbuf = {1, -1, 0};
```

No use el método de iniciación del ejemplo 8.20 porque no es del tipo de implementación independiente. La estructura struct sembuf, que especifica las operaciones para la estructura semop, tiene como garantía el tener como miembros a sem_num, sem_op y a sem_flg. El Spec 1170 no especifica en qué orden aparecen estos miembros en la definición, ni el estándar dice que struct sembuf sólo debe tener esos miembros.

Ejemplo 8.21

La función set_sembuf_struct *inicia los miembros de la estructura* struct sembuf: sem_num, sem_op y sem_flg *en forma de implementación independiente.*

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

void set_sembuf_struct(struct sembuf *s, int num, int op, int flg)
{
   s->sem_num = (short) num;
   s->sem_op = op;
   s->sem_flg = flg;
   return;
}
```

Ejemplo 8.22

El siguiente segmento de código incrementa en forma atómica los elementos del grupo de semáforos definidos en el programa 8.3.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
```

```
#include <errno.h>
struct sembuf myop[2];

set_sembuf_struct(&(myop[0]), 0, 1, 0);
set_sembuf_struct(&(myop[1]), 1, 2, 0);
if (semop(semid, myop, 2) == -1)
   perror("Semaphore operation failed");
```

El elemento cero del ejemplo 8.22 es incrementado en 1 y el elemento uno es incrementado en dos. El código usa el `set_sembuf_struct` del ejemplo 8.21 para iniciar las operaciones.

Ejemplo 8.23

Supongamos que un UNIX System V de dos elementos en el grupo de semáforos, `S`*, representa el sistema de unidad de cinta del ejemplo 8.13.* `S[0]` *representa a la cinta A y* `S[1]` *representa a la cinta B. Ambos elementos de* `S` *son inicializados a* 1*. El siguiente segmento de pseudocódigo define las operaciones de semáforo requeridas para accesar una o ambas unidades de cinta.*

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
struct sembuf GET_TAPES[2];
struct sembuf RELEASE_TAPES[2];

set_sembuf_struct(&(GET_TAPES[0]), 0, -1, 0);
set_sembuf_struct(&(GET_TAPES[1]), 1, -1, 0);
set_sembuf_struct(&(RELEASE_TAPES[0]), 0, 1, 0);
set_sembuf_struct(&(RELEASE_TAPES[1]), 1, 1, 0);

Process 1:      semop(S, GET_TAPES, 1);
                <use tape A>
                semop(S, RELEASE_TAPES, 1);

Process 2:      semop(S, GET_TAPES, 2);
                <use tapes A and B>
                semop(S, RELEASE_TAPES, 2);

Process 3:      semop(S, GET_TAPES + 1, 1);
                <use tape A>
                semop(S, RELEASE_TAPES + 1, 1);
```

El programa 8.4 es una variante del programa 8.1, que usa los semáforos del System V para proteger una sección crítica. El programa llama a `set_sembuf_struct`, definido en el ejemplo 8.21, y `remove_semaphore`, definido en el ejemplo 8.26. Recomenzaría las operaciones `semop` si fueran interrumpidas por una señal.

Programa 8.4: Una modificación del programa 8.1 que usa el System V de semáforos para proteger a una sección crítica.

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <limits.h>
#include <errno.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define PERMS S_IRUSR | S_IWUSR
#define SET_SIZE 2

void set_sembuf_struct(struct sembuf *s, int semnum,
                                         int semop, int semflg);
int remove_semaphore(int semid);

void main (int argc, char *argv[])
{
   char buffer[MAX_CANON];
   char *c;
   int i;
   int n;
   pid_t childpid;
   int semid;
   int semop_ret;
   struct sembuf semwait[1];
   struct sembuf semsignal[1];
   int status;

   if ( (argc != 2) || ((n = atoi (argv[1])) <= 0) ) {
      fprintf (stderr, "Usage: %s number_of_processes\n", argv[0]);
      exit(1);
   }
            /* Crea un semáforo que contiene un solo elemento */
   if ((semid = semget(IPC_PRIVATE, SET_SIZE, PERMS)) == -1) {
      fprintf(stderr, "[%ld]: Semáforo no válido el acceso: %s\n"
         (long)getpid(), strerror(errno));
      exit(1);
   }
                     /* Inicia el elemento del semáforo a 1 */
   set_sembuf_struct(semwait, 0, -1, 0);
   set_sembuf_struct(semsignal, 0, 1, 0);

   if (semop(semid, semsignal, 1) == -1) {
```

```
        fprintf(stderr, "[%ld]: Semáforo no válido el acceso - %s\n"
            (long)getpid(), strerror(errno));
        if (remove_semaphore(semid) == -1)
            fprintf(stderr, "[%ld], Semáforo no válido el acceso - %s\n"
                (long)getpid(), strerror(errno));
        exit(1);
    }

    for (i = 1; i < n;  ++i)
        if (childpid = fork())
            break;
    sprintf(buffer,
        "i:%d  process ID:%ld  parent ID:%ld  child ID:%ld\n",
        i, (long)getpid(), (long)getppid(), (long)childpid);

    c = buffer;
        /********************** sección de entrada **********************/
    while(( (semop_ret = semop(semid, semwait, 1)) == -1) &&
            (errno == EINTR))
        ;
    if (semop_ret == -1)
        fprintf(stderr, "[%ld]: Semáforo no válido el acceso - %s\n"
            (long)getpid(), strerror(errno));
    else {
        /************** principio de sección crítica ******************/
        while (*c != '\0') {
            fputc(*c, stderr);
            c++;
        }
        fputc('\n', stderr);
        /***************** final de sección crítica *******************/
        /********************* sección de salida ***********************/
        while(((semop_ret = semop(semid, semsignal, 1)) == -1) &&
                (errno == EINTR))
            ;
        if (semop_ret == -1)
            fprintf(stderr, "[%ld]: Semáforo no válido el acceso - %s\n"
                (long)getpid(), strerror(errno));
    }
        /******************* sección remanente ***********************/
    while((wait(&status) == -1) && (errno == EINTR))
        ;
    if (i == 1)  /* el proceso original elimina al semáforo *//
        if (remove_semaphore(semid) == -1)
            fprintf(stderr, "[%ld], Semáforo no válido el acceso - %s\n"
                (long)getpid(), strerror(errno));
    exit(0);
}
```

Programa 8.4

Los semáforos de System V crean e inician llamadas de sistema por separado. Si un proceso creara un semáforo y otro proceso intentara un `semop` antes de que el proceso original tuviera la oportunidad de iniciar el semáforo, los resultados de esta ejecución serían impredecibles. Esta impredicibilidad es un ejemplo de la *condición de competencia*, puesto que la condición de error depende del tiempo preciso entre las instrucciones de varios procesos. El programa 8.4 no tiene este problema, pues el padre original crea e inicia el semáforo antes de efectuar un `fork`. Puesto que el semáforo es privado, sólo el proceso original lo puede accesar al tiempo de creación.

Un enfoque que podría solucionar el problema de iniciación cuando los semáforos no son privados sería esperar un valor 0, en lugar de esperar un valor positivo [85]. La señal tradicional de operación se convertiría en una reducción a 0 y la espera tradicional de operación se incrementaría a 1.

Ejemplo 8.24

El siguiente segmento de código implementa una espera de semáforo como un incremento en lugar de una reducción.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#include <errno.h>
#define SET_SIZE 2
struct sembuf myop[SET_SIZE];

set_sembuf_struct(&(myop[0]), 0, 0, 0);
set_sembuf_struct(&(myop[1]), 0, 1, 0);
if (semop(semid, myop, 2) == -1)
   perror("La operación del semáforo falló");
```

Obsérvese que ambas operaciones en `myop` del ejemplo 8.24 se refieren al elemento cero. Si el elemento cero no tiene actualmente el valor 0, el proceso lo bloquea hasta que éste se convierta en 0. Cuando el elemento es 0, éste es incrementado a 1 por medio de una operación atómica. Puesto que `semget` inicia los elementos del semáforo a 0, el semáforo está listo para ser usado inmediatamente después de `semget`, sin necesidad de ninguna otra iniciación.

El enfoque utilizado en el ejemplo 8.24 asume que las operaciones del semáforo son intentadas en orden [85]. El Spec 1170 no ordena de manera explícita esta implementacion, por lo que no se debe depender de esto. La función `create_and_initialize` del programa 8.5 muestra un enfoque alternativo para iniciar un elemento del semáforo con un valor en particular.

El elemento del semáforo que va a ser iniciado en el programa 8.5 tiene asociado consigo un elemento de candado. El semáforo es el elemento uno y el candado el elemento cero. La función regresa una manija al semáforo en caso de éxito, o un `-1` en caso de falla. Si el semá-

foro ya existe, el llamador espera a que el elemento de candado se inicialice a 1 antes de proceder. El sistema de verificación de error para `semop` no se muestra en este análisis, pero la llamada a `semop` debe ser reemplazada por una llamada a `semop_restart`, como se muestra en el ejemplo 8.29, para así poder reiniciar la llamada después de una interrupción por señal. El programa 8.5 debe desasignar al semáforo si es que ocurre un error real en `semop`.

Programa 8.5: Una función que en forma atómica crea e inicia un semáforo de System V.

```
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
#include <errno.h>
#define PERMS S_IRUSR|S_IWUSR
#define SET_SIZE 2

int create_and_initialize(key_t mykey, int value)
{
   int semid;
   struct sembuf getlock, setlock, setinit;

   semid = semget(mykey, SET_SIZE, PERMS|IPC_CREAT|IPC_EXCL);
   if ((semid < 0) && (errno != EEXIST)) {
       perror("Semaphore create failed");
       return -1;
   } else if (semid > 0) {
        /* Inicia el semáforo y abre el candado */
       set_sembuf_struct(&setinit, 1, value, 0);
       semop(semid, &setinit, 1);
       set_sembuf_struct(&setlock, 0, 1, 0);
       semop(semid, &setlock, 1);
   } else {
       /* el semáforo ya existe - espere la inicialización */
       if ((semid = semget(mykey, SET_SIZE, PERMS)) < 0) {
          perror("Could not access existing semaphore");
          return -1;
       }
       set_sembuf_struct(&getlock, 0, -1, 0);
       semop(semid, &getlock, 1);
       set_sembuf_struct(&setlock, 0, 1, 0);
       semop(semid, &setlock, 1);
   }
   return 0;
}
```

Programa 8.5

8.4.4 Control de semáforo

La función `semctl` conjunta o agrupa los valores de elementos individuales de semáforo. También ejecuta otras funciones de control, incluyendo la destrucción del semáforo.

SINOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl(int semid, int semnum, int cmd,
                      /* union semun arg */ ...);
```

Spec 1170

En este caso, `semid` identifica al grupo de semáforos, y `semnum` señala al elemento del semáforo dentro del grupo, si es que `cmd` se refiere a elementos individuales. El argumento `cmd` especifica cuáles son los comandos por ejecutar y `arg` es usado de diferentes formas para los distintos valores de `cmd`. Los siguientes son los comandos más importantes que han de usarse junto con `semctl`:

`GETVAL`:	Regresa el valor a un elemento específico del semáforo.
`GETPID`:	Regresa la identificación (ID) al último proceso que manipuló al elemento.
`GETNCNT`:	Regresa el número de procesos que están esperando que un elemento se incremente.
`GETZCNT`:	Regresa el número de procesos que están esperando que un elemento se convierta en 0.
`SETVAL`:	Inicia el valor específico de un elemento del semáforo al valor de `arg.val`.
`IPC_RMID`:	Elimina el semáforo identificado por `semid`.
`IPC_SET`:	Inicia los permisos del semáforo.

Quizá sea necesario que se incluya la definición `union semun` directamente en los programas, puesto que algunos sistemas no la definen en sus archivos de encabezado para semáforos. Su definición es:

```
union semun {
   int val;
   struct semid_ds *buf;
   ushort *array;
};
```

Si ocurre un error, el `semctl` regresa un `-1` y se inicia a `errno`. El valor de regreso cuando no existe ningún error depende de `cmd`. Los valores `GETVAL`, `GETPID`, `GETNCNT` y `GETZCNT` de `cmd` originan que `semctl` regrese un valor asociado con `cmd`. El resto de los valores de `cmd` hacen que `semctl` regrese 0.

Ejemplo 8.25

La función `initialize_sem_element` *inicia el valor de un elemento específico del semáforo a* `semvalue`.

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/sem.h>

int initialize_sem_element(int semid, int semnum, int semvalue)
{
   union semun arg;
   arg.val = semvalue;
   return semctl(semid, semnum, SETVAL, arg);
}
```

Los parámetros `semid` y `semnum` de `initialize_sem_element` en el ejemplo 8.25 identifican al grupo de semáforos y al elemento dentro de ese grupo cuyo valor vaya a ser iniciado a `semvalue`. La función `initialize_sem_element` regresa un 0 en caso de éxito. En caso de falla regresa un `-1` con `errno`, iniciado de acuerdo con `semctl`.

Ejemplo 8.26

La función `remove_semaphore` *borra el semáforo especificado por* `semid`.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int remove_semaphore(int semid)
{
   return semctl(semid, 0, IPC_RMID);
}
```

La función `remove_semaphore` del ejemplo 8.26 regresa un 0 en caso de éxito. En caso de falla regresa un `-1` con `errno` iniciado de acuerdo con `semctl`.

8.4.5 Estado de semáforo

Las implementaciones de semáforo de System V típicas mantienen una tabla única de los grupos de semáforos en el núcleo. Investigue el estado de estos grupos de semáforos desde el intérprete de comandos utilizando el comando `ipcs`. Elimine grupos de semáforos utilizando el comando `ipcrm`.

Ejemplo 8.27

El siguiente comando enlista todos los semáforos.

```
ipcs -s
```

El comando `ipcs` del ejemplo 8.27 puede producir un mensaje indicando que la facilidad semáforo no está en el sistema. Bajo el paquete Sun Solaris 2, la facilidad del semáforo es cargada hasta después de que el programa ejecuta una operación de semáforo, normalmente la creación de un semáforo. La facilidad es cargada cuando el sistema es apagado o reiniciado, o cuando específicamente se descarga.

Ejemplo 8.28

El siguiente comando borra el semáforo con la identificación ID 12345.

```
ipcrm -s 12345
```

8.5 Semáforos y señales

En el contexto de semáforos, la palabra *señal* tiene dos significados: incrementar a un semáforo y generar una notificación de un evento UNIX. En esta sección se usará el significado de señal como una señal UNIX y no como una operación de señal de semáforo.

Durante la ejecución de llamadas de sistema lentas (aquellas que pueden bloquear por tiempo indefinido), la acción de la señal depende de cómo haya sido capturada ésta. Bajo el System V y el Spec 1170, `sigaction` provee la opción de reiniciar la llamada del sistema después de que el manejador de señal sale (`SA_RESTART`). Esa opción no es parte de POSIX, y cuando se esté escribiendo funciones de biblioteca usadas por otros, puede ser que el desarrollador no tenga control de la manera como fueron establecidas las señales. Si las llamadas de sistema lentas no son automáticamente reiniciadas por los manejadores de señal, una operación de semáforo tal como una espera (reducción de un semáforo) puede regresar sin que el decremento del semáforo haya sido ejecutado. Si `semop` es interrumpido por una señal, regresa un `-1` e inicia `errno` a `EINTR`.

Ejemplo 8.29

Insertar un código de reinicio para cada `semop` *es una labor tediosa. El* `semop_restart` *es como un* `semop`, *pero reinicia la llamada si ésta es interrumpida por una señal.*

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <errno.h>

int semop_restart(int semid, struct sembuf *sops, int nsops)
{
   int retval;
   while ( ((retval = semop(semid, sops, nsops)) == -1) &&
           (errno == EINTR) )
      ;
   return retval;
}
```

Ejemplo 8.30

El `sem_wait` *de POSIX.1b también regresa si éste es interrumpido por una señal. La función* `sem_wait_restart` *reinicia la llamada si ésta fue interrumpida por una señal.*

```
#include <semaphore.h>
#include <errno.h>
```

```
int sem_wait_restart(sem_t *sem)
{
   int retval;
   while ( ((retval = sem_wait(sem)) == -1) &&
           (errno == EINTR) )
      ;
   return retval;
}
```

8.6 Ejercicio: Semáforos no nombrados POSIX

Este ejercicio describe una implementación de semáforos no nombrados tipo POSIX, en términos de semáforos de System V. Los prototipos para las funciones de semáforo son:

```
#include "mysem.h"
int mysem_init(mysem_t *sem, int pshared, unsigned int value);
int mysem_destroy(mysem_t *sem);
int mysem_wait(mysem_t *sem);
int mysem_post(mysem_t *sem);
```

Todas estas funciones regresan un 0 en caso de éxito. En caso de error regresan un `-1` e inician a `errno` de forma apropiada. En realidad, el último punto es un poco sutil. Es probable que las únicas órdenes que pudieran crear un error en el System V serían las llamadas de semáforo, y sería raro, pues éstas son las que inician a `errno`. Si este fuera el caso, la función regresaría el valor correcto de `errno`, siempre y cuando no hubiera intervenciones del sistema o llamadas a bibliotecas. Recuerde que en POSIX.1c se permite a `errno` que sea una macroinstrucción, para así evitar iniciar explícitamente esta variable.

Defina un tipo de variable llamada `mysem_t` para los semáforos. Para este ejercicio, `mysem_t` es simplemente un `int`. Ponga las declarativas y cualquier otro tipo de información en el archivo de encabezado llamado `mysem.h` para que así los usuarios puedan incluir este archivo en programas que llamen a estas funciones. Puesto que un proceso que va a compartir con sus hijos un semáforo no nombrado, se debe llamar a `mysem_init` antes de la bifurcación, por lo que no hay que preocuparse acerca de condiciones de competencia cuando se está implementando la iniciación de semáforos en términos de un System V. El valor de `mysem_t` es la identificación (ID) del semáforo de un semáforo de System V. Ignore el valor de `pshared`, puesto que los semáforos de System V son compartibles entre procesos. Utilice una llave de `IPC_PRIVATE`.

Implemente los `mysem_wait` y `mysem_post` directamente con llamadas a `semop`. Los detalles dependerán de cómo inicie `sem_init` el semáforo. Implemente `mysem_destroy` con una llamada a `semctl`.

Escriba un controlador de prueba basado en el programa 8.2 y verifique que la biblioteca obtenga la cualidad de independencia exclusiva. El programa de prueba debe crear y borrar todos los semáforos que use.

Antes de salir del sistema, utilice `ipcs -s` de la línea de comando. Si todavía existen semáforos (debido a un problema con el programa), borre cada uno de ellos usando:

```
ipcrm -s n
```

Este comando borra el semáforo con una identificación (ID) n. El semáforo debe ser creado sólo una vez por el programa de prueba. Asimismo, debe ser borrado sólo una vez, pero no por todos los hijos en la cadena del proceso.

8.7 Ejercicio: Semáforos nombrados POSIX

Este ejercicio describe una implementación de semáforos nombrados tipo POSIX, en términos de semáforos de System V. Los prototipos para las funciones de semáforo son:

```
#include "mysem.h"
mysem_t *mysem_open(const char *name, int oflag, mode_t mode,
                    unsigned int value);
int mysem_close(mysem_t *sem);
int mysem_unlink(const char *name);
int mysem_wait(mysem_t *sem);
int mysem_post(mysem_t *sem);
```

Todas estas funciones regresan un -1 e inician a errno cuando existe un error. Para simplificar el proceso de interfase, siempre llame a mysem_open con cuatro parámetros.

Use un archivo ordinario para representar al semáforo. Este archivo contiene el ID del semáforo utilizado por el System V que está siendo usado para implementar el semáforo POSIX. La función mysem_open crea el archivo, asigna al semáforo de System V y almacena su ID en el archivo. La función mysem_open regresa el indicador a un descriptor de archivo que representa al archivo. Debe asignar espacio para este descriptor de archivo. Las funciones mysem_unlink, mysem_wait y mysem_post utilizan un indicador dirigido a este descriptor como su parámetro. La función mysem_close hace que el semáforo sea inaccesible a cualquier llamada. La función mysem_unlink borra el semáforo, su archivo correspondiente y el número entero que contiene el descriptor de archivo. La función mysem_wait reduce al semáforo y la función mysem_post incrementa al semáforo.

Ponga todas estas funciones del semáforo en una biblioteca, por separado, y considere a ésta un objeto cuyas únicas relaciones con el exterior sean las cinco funciones arriba enlistadas. No se preocupe por condiciones de competencia con el uso de mysem_open para la creación del archivo hasta que no tenga una versión rudimentaria del programa de prueba que está trabajando. Desarrolle un mecanismo para liberar el semáforo de System V después de ejecutarse la última sem_unlink, y sólo después de que el último proceso haya cerrado este semáforo. El mysem_unlink no puede hacer esta liberación directamente, porque otros procesos pueden tener todavía abierto el semáforo POSIX. Una posibilidad es hacer que mysem_close verifique el contador de enlaces en el nodo y que libere al semáforo de System V, si el contador de enlaces tiene el valor de 0.

Cuando todo lo demás esté trabajando, trate de manejar las diversas condiciones de competencia usando un semáforo adicional de System V, para proteger las secciones críticas de la

iniciación del semáforo. Utilice el mismo semáforo para todas las copias de bibliotecas de semáforos POSIX para protegerse de la interacción entre procesos no relacionados. Refiérase a este semáforo con un nombre de archivo que se pueda convertir en una llave usando `ftok`.

8.8 Ejercicio: Manejador de licencias

Los ejercicios en esta sección se refieren principalmente a los programas de `runsim` desarrollados en los ejercicios de la sección 2.13. En esos ejercicios, `runsim` lee comandos de entrada normal y bifurca al hijo para que `execvp` ejecute cada comando. El programa `runsim` de un argumento toma líneas de comando individuales, especificando el número de procesos hijos que le es permitido ejecutar simultáneamente. También lleva una cuenta de los hijos y usa a `wait` para bloquear cuando se ha llegado al límite.

En estos ejercicios, `runsim` nuevamente lee comandos de una entrada normal y bifurca hacia un hijo. El hijo en turno bifurca a un nieto para hacer el `execvp`. El primer hijo espera a que el nieto termine y luego sale. La figura 8.1 muestra la estructura de `runsim` mientras tres pares como el descrito se están ejecutando. Este programa utiliza semáforos para controlar el número de ejecuciones simultáneas.

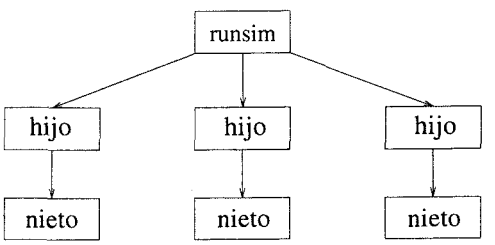


Figura 8.1: Esta es la estructura de `runsim` cuando los `execvp` son ejecutados por los nietos en lugar de los hijos.

Escriba un programa `runsim` que ejecute hasta `n` procesos a la vez. Empiece el programa `runsim` tecleando el siguiente comando:

```
runsim n
```

Antes de empezar el programa `runsim`, cree el siguiente archivo si es que no existe aún:

```
/tmp/license.uid
```

El `uid` debería ser el valor numérico del ID del usuario del proceso. Asegúrese de eliminar este archivo antes de salir del sistema.

Implemente `runsim` de la siguiente manera:

- Verifique que se tenga número correcto de argumentos de línea de comando y genere un mensaje en caso de que sea incorrecto.
- Genere una llave de semáforo llamando a la función `ftok` con el nombre de ruta `/tmp/license.uid`, donde `uid` es el ID del usuario del proceso. También utilice esta ID para el parámetro `id` de `ftok`.
- Cree un grupo de semáforos usando la llave generada en el paso anterior. El grupo de semáforos tiene un elemento que es iniciado a `n`. Use la variable `license` para guardar el ID del semáforo que fue regresado por `semget`.
- Ejecute el siguiente ciclo hasta que se encuentre el fin de archivo de una entrada estándar:
  * Lea un comando de la entrada estándar hasta que se llegue al número de caracteres en `MAX_CANON`.
  * Ejecute una operación de espera de semáforo en el semáforo `license`.
  * Bifurque a un hijo que llama a `perform_command` y luego sale. Pase la cadena de caracteres de entrada a `perform_command`.
  * Verifique si alguno de los hijos ha terminado (`waitpid`) con la opción de (`WNOHANG`)
- Espere a que todos los hijos terminen y después elimine el semáforo `license`. Si el programa se cae durante la depuración, asegúrese de eliminar el semáforo usando el comando `ipcrm` antes de empezar a correr `runsim` nuevamente.

La función `perform_command` tiene el siguiente prototipo:

```
void perform_command(char *cline);
```

Implemente `perform_command` de la siguiente manera:

- Bifurque a un hijo (un nieto del original). Este nieto llama a `makeargv` sobre `cline` y `execvp` el comando.
- Espere a este hijo y después haga una señal de semáforo sobre el semáforo `license`.
- Salga.

Verifique el programa como en la sección 2.13. Corrija los mensajes de error para hacerlos más legibles. Escriba un programa de prueba que toma dos argumentos en la línea de comando: el tiempo dormido y el factor de repetición. El programa de prueba simplemente repetirá un mismo ciclo el número de veces especificado. Duerme en el ciclo y luego despliega un mensaje con el ID de su proceso. Luego sale. Utilice `runsim` para ejecutar múltiples copias del programa de prueba.

Ahora modifique a `runsim` para que, en caso de que el semáforo `license` ya exista, el programa accese el semáforo sin cambiar su valor. El programa `runsim` sigue esperando a sus hijos pero no elimina los semáforos. Trate de ejecutar varias copias de `runsim` en forma concurrente. Puesto que todos usan el mismo semáforo, el número de procesos nietos seguirá estando limitado por `n`. Cuando termine, asegúrese de usar `ipcrm` para eliminar al semáforo.

8.9 Ejercicio: Memoria compartida en System V

Las características de comunicación entre procesos del System V provee tres tipos de objetos: semáforos, memoria compartida y colas de mensajes. Esta sección describe la implementación de un *software* sencillo de entubamiento, usando semáforos y memoria compartida. En la sección 8.10 se muestra una implementación similar para colas de mensajes.

8.9.1 Un panorama de memoria compartida del System V

Las reglas para crear un acceso a la memoria compartida del System V son muy similares a las de los semáforos.

SINOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
void *shmat(int shmid, void *shmaddr, int shmflg);
int shmdt(void *shmaddr);
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Spec 1170

El `shmget` crea un segmento de memoria compartida usando una llave especificada. El `size` es el número de bytes en el segmento de memoria y `shmflg` indica los permisos de acceso para el segmento. Hay programas que usan la característica de memoria compartida `shmat` o `shmdt` para agregar o quitar segmentos de memoria compartida del espacio del usuario. La función `shmat` regresa un indicador a `void *`, para que el programa use el valor de regreso como un indicador ordinario de memoria que obtiene de `malloc`. Utilice un `shmaddr` de `NULL`. En algunos sistemas puede ser necesario iniciar a `shmflg` para que el segmento de memoria esté correctamente alineado. Cuando se termina con un segmento de memoria compartida, el programa lo debe quitar con una llamada a `shmdt`. El último proceso que se quita del segmento debe desasignar el segmento de memoria compartida llamando a `shmctl`.

8.9.2 Especificación para implementación de *software* de entubamiento

En esta sección se desarrolla una especificación para un *software* de entubamiento que consta de un grupo de semáforos para proteger el acceso al entubamiento y un segmento de memoria compartida para guardar los datos en el entubamiento y para que proporcione información. La información del estado de entubamiento incluye el número de bytes de datos que hay en el entubamiento, la posición que guarda el siguiente byte que será leído y la información de su estado. El entubamiento puede guardar, cuando mucho, un mensaje de un máximo tamaño de `_POSIX_PIPE_BUF`. Represente el entubamiento con la siguiente estructura `pipe_t`, asignada en memoria compartida.

```
typedef struct pipe {
   int semid;                    /* ID (identificación) de protección del
                                    grupo de semáforo */
   int shmid;                    /* ID (identificación) del segmento de
                                    memoria compartida */
   char data[_POSIX_PIPE_BUF];   /* buffer para datos en el entubamiento */
   int data_size;                /* bytes ocupados actualmente en el
                                    entubamiento */
   void *scurrent_start;         /* apuntador de inicio de datos actuales */
   int end_of_file;              /* verdadero (true) después de que el
                                    entubamiento ha sido cerrado para
                                    escritura */
} pipe_t;
```

Un programa crea y referencia entubamientos usando como manija un apuntador a `pipe_t`. Para que sea más sencillo, asuma que sólo un proceso puede leer del entubamiento y que sólo un proceso puede escribir al entubamiento. Es responsabilidad del lector limpiar el entubamiento cuando sea cerrado. Cuando el escritor cierra el entubamiento, inicia el miembro `end_of_file` de `pipe_t`, para que el lector pueda detectar el fin de archivo.

El grupo de semáforos protege la estructura de datos `pipe_t` durante el acceso compartido por el lector y el escritor. El elemento cero del grupo de semáforos controla el acceso exclusivo al anexo `data`. Su valor inicial es de `1`. Los lectores y escritores pueden acceder al entubamiento reduciendo el elemento del semáforo, y pueden dejar el acceso incrementando este elemento. El elemento uno del grupo de semáforos, controla la sincronización de la escritura para que el buffer `data` contenga sólo un mensaje, esto es, la salida de una sola operación de `write`. Cuando este elemento del semáforo es `1`, el entubamiento esta vacío. Cuando está en 0, el entubamiento contiene datos o un mensaje que indica que se ha llegado a un fin de archivo. Inicialmente el elemento uno se inicia a `1`. El escritor reduce al elemento uno antes de escribir cualquier dato. El lector espera hasta que el elemento uno sea 0 antes de leer. Cuando ha leído todos los datos del entubamiento, el lector incrementa al elemento uno para indicar que ahora el entubamiento está disponible para escribir. Escriba las siguientes funciones:

- La función `pipe_open` crea un *software* de entubamiento y regresa un apuntador tipo `pipe_t *` que será usado como una manija en otras llamadas. El prototipo para `pipe_open` es:

  ```
  pipe_t *pipe_open(void);
  ```

  El algoritmo para `pipe_open` es:
  * Crear un segmento de memoria compartida que guarde la estructura de datos `pipe_t` llamando a `shmget`. Utilice una llave de `IPC_PRIVATE` y permisos de lectura/escritura del propietario.
  * Junte al segmento llamando a `shmat`. Efectúe una conversión del tipo de salida del valor de regreso de `shmat` a un `pipe_t *` y asígneselo a una variable local `p`.
  * *Inicie* `p->shmid` *a la ID del segmento de memoria compartida que fue regresado por* `shmget`.
  * Inicie `p->data_size` y `p->end_of_file` a 0.

* Cree un grupo de semáforos que contenga dos elementos, llamando a `semget` con una llave `IPC_PRIVATE` y con permisos de lectura, escritura y ejecución para el propietario.
* Inicie ambos elementos del semáforo a `1` y ponga el valor del ID resultante del semáforo en `p->semid`.
* Si todas las llamadas fueron exitosas, regrese una `p`.
* Si ocurre un error, desasigne todos los recursos y regrese un apuntador `NULL`.

- El `pipe_read` tiene el prototipo:

```
int pipe_read(pipe_t *p, char *buf, int bytes);
```

La función `pipe_read` se comporta como una llamada ordinaria del sistema `read` con bloqueo. El algoritmo para `pipe_read` es:

* Ejecute un `semop` en `p->semid` para decrementar en forma atómica el elemento cero del semáforo y verificar si el elemento uno es 0. El elemento cero provee la independencia exclusiva. El elemento uno es solamente 0 si existe algo dentro del buffer.
* Si `p->data_size` es mayor que 0, copie como máximo el valor de `bytes` en bytes de información, empezando en la posición `p->current_start` del *software* de entubamiento hacia `buf`. Tome en consideración que el buffer de datos del entubamiento tiene un tamaño total de `_POSIX_PIPE_BUF`.
* Actualice a los miembros de la estructura de datos del entubamiento `p->current_start` y `p->data_size`.
* Asigne el valor de regreso al número de bytes que realmente hayan sido leídos o regrese el valor `-1` si es que hubo algún error. Si `p->data_size` es 0 y `p->end_of_file` tiene una condición válida, entonces asegure el valor de regreso a 0 para indicar un fin de archivo.
* Ejecute otra operación `semop` para liberar el acceso al entubamiento. Incremente el elemento cero. Si no existen más datos en el entubamiento, también incremente el elemento uno a menos que `p->end_of_file` esté en condición válida. Ejecute estas operaciones en forma atómica mediante un solo llamado a `semop`.

- El `pipe_write` tiene el prototipo:

```
int pipe_write(pipe_t *p, char *buf, int bytes);
```

La función `pipe_write` se comporta como una llamada ordinaria del sistema de `write` con bloqueo. El algoritmo de `pipe_write` es:

* Ejecute un `semop` en `p->semid` para decrementar en forma atómica los elementos cero y uno del semáforo.
* Copie en el buffer del entubamiento un máximo de `_POSIX_BUF_MAX` bytes.
* Asigne `p->data_size` al número de bytes que fueron realmente copiados y asigne `p->current_start` el valor 0.

* Ejecute otra llamada `semop` para incrementar en forma atómica el elemento cero del grupo de semáforo.
* Regrese el número de bytes copiado o el valor `-1` si algún llamado de `semop` falló.

- El `pipe_close` tiene el prototipo:

```
int pipe_close(pipe_t *p, int how);
```

El parámetro `how` determina si el entubamiento está cerrado ya sea para lectura o escritura. Sus valores posibles son `O_RDONLY` y `O_WRONLY`. El algoritmo para `pipe_close` es:

* Use la función `semop` para reducir en forma atómica el elemento cero de `p->semid`. Si `semop` falla, regrese el valor `-1`.
* Si `how & O_WRONLY` es verdadero,
  - Asigne a `p->end_of_file` el valor true (verdadero).
  - Ejecute un `semctl` para asignar el elemento de `p->semid` a 0.
  - Copie a `p->semid` dentro de una variable local `semid_temp`.
  - Ejecute un `shmdt` para soltar a `p`.
  - Ejecute un `semop` para incrementar en forma atómica el elemento cero de `semid_temp`.

  Si cualquiera, `semop`, `semctl` o `shmdt`, llama una falla, inmediatamente regrese un `-1`.
* Si `how & O_RDONLY` es una en condición verdadera,
  - Ejecute un `semctl` para eliminar al semáforo `p->semid`. (Si el escritor está esperando en el grupo de semáforos, su `semop` regresará un error cuando esto suceda.)
  - Copie `p->shmid` dentro de una variable local, `shmid_temp`.
  - Llame a `shmdt` soltar a `p`.
  - Llame a `shmctl` para desasignar el segmento de memoria compartida identificado como `shmid_temp`.

  Si cualquiera, `semctl`, `shmdt` o `shmctl`, llama una falla, regrese inmediatamente un valor `-1`.

Verifique el *software* de entubamiento escribiendo un programa principal similar al programa 3.3. El programa crea un *software* de entubamiento y después bifurca hacia un hijo. El hijo lee desde la entrada estándar y escribe al entubamiento. El padre lee lo que el hijo ha escrito en el entubamiento y lo saca a la salida estándar. Cuando el hijo detecta el fin de archivo en la entrada estándar, cierra el entubamiento para propósitos de escritura. El padre entonces detecta el fin de archivo en el entubamiento, cierra éste para propósitos de lectura (lo cual destruye el entubamiento) y sale. Asegúrese de que todo fue correctamente destruido ejecutando el comando `ipcs`.

La especificación arriba mencionada describe versiones con bloqueo de las funciones `pipe_read` y `pipe_write`. Modifique y pruebe también una versión sin bloqueo.

8.10 Ejercicio: Colas de mensajes del System V

Las características de colas de mensajes del System V IPC permite a un programa añadir o eliminar diferentes tipos de mensajes en la cola. Después de crear una cola con `msgget`, un programa inserta mensajes en la cola con `msgsnd`, destruye mensajes con `msgrcv` y desasigna una cola de mensajes o permite cambios con `msgctl`.

SINOPSIS

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg);
int msgsnd(int msqid, const void *msgp, size_t msgsz,
           int msgflg);
int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
           int msgflg);
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Spec 1170

Formule una especificación para la implementación de un *software* de entubamiento, en términos de colas de mensajes. Implemente las siguientes funciones:

```
pipe_t *pipe_open(void);
int pipe_read(pipe_t *p, char *buf, int chars);
int pipe_write(pipe_t *p, char *buf, int chars);
int pipe_close(pipe_t *p);
```

Diseñe una estructura `pipe_t` para que concuerde con la implementación. Verifique ésta como se hizo en la sección 8.9.

8.11 Lecturas adicionales

La mayoría de los libros en sistemas operativos [80,92] discuten la clásica abstracción de semáforo. *UNIX Networking Programming,* escrito por Stevens [85], ofrece una amplia discusión en comunicación de interprocesos de System V, incluyendo semáforos, memoria compartida y colas de mensajes. El libro *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmer,* escrito por Schimmel [77], presenta un enfoque avanzado de cómo estos temas se aplican al diseño de núcleos con multiprocesadores.

Capítulo 9

Hilos POSIX

Un método para lograr el paralelismo consiste en hacer que varios procesos cooperen y se sincronicen mediante memoria compartida. Una alternativa es el empleo de múltiples hilos de ejecución en un solo espacio de direcciones. En este capítulo ilustramos la utilidad de los hilos comparando formas de enfocar el problema de vigilar múltiples descriptores de archivo para la entrada. A continuación, el capítulo presenta un panorama general de la gestión básica de hilos bajo el estándar POSIX.1c. En este capítulo veremos diversos modelos de hilos y cómo se ajustan estos modelos al estándar.

Cuando se ejecuta un programa, la CPU utiliza el valor del contador de programa del proceso para determinar cuál instrucción debe ejecutar a continuación. El flujo de instrucciones resultante se denomina *hilo de ejecución* del programa, y es el flujo de control para el proceso representado por la secuencia de direcciones de instrucciones indicadas por el contador de programa durante la ejecución del código de éste.

Ejemplo 9.1

El proceso 1 ejecuta los enunciados 245, 246 *y* 247 *en un ciclo. Su hilo de ejecución puede representarse así:* 245_1, 246_1, 247_1, 245_1, 246_1, 247_1, 245_1, 246_1, 247_1.... *Los subíndices identifican el hilo de ejecución. En este caso sólo hay uno.*

Desde el punto de vista del programa, la secuencia de instrucciones de un hilo de ejecución es un flujo ininterrumpido de direcciones, como se aprecia en el ejemplo 9.1. En cambio, desde

el punto de vista del procesador, los hilos de ejecución de diferentes procesos están entremezclados, y el punto en que la ejecución cambia de un proceso a otro se denomina *conmutación de contexto.*

Ejemplo 9.2

El proceso 1 ejecuta sus enunciados 245, 246 *y* 247 *en un ciclo igual que en el ejemplo 9.1, y el proceso 2 ejecuta sus enunciados* 10, 11, 12.... *La CPU ejecuta instrucciones en el orden* 245_1, 246_1, 247_1, 245_1, 246_1, 10_2, 11_2, 12_2, 13_2, 247_1, 245_1, 246_1, 247_1... *Hay conmutación de contexto entre* 246_1 *y* 10_2 *y entre* 13_2 *y* 247_1. *Para el procesador los hilos de ejecución están intercalados, pero para los procesos individuales son secuencias continuas.*

Una extensión natural del modelo de proceso es permitir la ejecución de varios hilos dentro del mismo proceso. Esta extensión provee un mecanismo eficiente para controlar hilos de ejecución que comparten tanto código como datos, con lo que se evita conmutaciones de contexto. La estrategia también tiene un potencial importante para mejorar el rendimiento, porque las máquinas con varios procesadores pueden ejecutar múltiples hilos simultáneamente. Los programas con paralelismo natural en forma de tareas independientes que operan sobre datos compartidos pueden aprovechar la capacidad de ejecución adicional de estas máquinas multiprocesador. Los sistemas operativos, en particular, pertenecen a la clase de programas que presentan considerable paralelismo natural y que pueden experimentar mejoras sustanciales en su rendimiento al tener múltiples hilos de ejecución simultáneos. Cuando los proveedores anuncian apoyo de *multiprocesamiento simétrico* quieren decir que el sistema operativo y las aplicaciones pueden tener múltiples hilos de ejecución indistintos y pueden aprovechar el *hardware* paralelo.

Cada hilo de ejecución se asocia con un *hilo*, un tipo de datos abstracto que representa el flujo de control dentro de un proceso. Cada hilo tiene su propia pila de ejecución, valor de contador de programa, conjunto de registros y estado. Al declarar muchos hilos dentro de los confines de un solo proceso, el programador puede lograr paralelismo con un gasto extra bajo. Si bien los hilos permiten tener paralelismo a bajo costo, agregan ciertas complicaciones en cuanto a la necesidad de sincronización.

Este capítulo aborda aspectos de implementación básicos— cómo se crean y controlan los hilos. En la sección 9.1 se ilustra la utilidad de los hilos con un problema sencillo de paralelismo natural —la vigilancia de múltiples descriptores de entrada. Se analizan cinco estrategias para resolver este problema. La complejidad de una implementación eficiente sin hilos deberá ser una prueba convincente de la utilidad de los hilos en este tipo de problemas. En el resto del capítulo estudiaremos la estructura básica y gestión de los hilos, según se especifica en el estándar POSIX.1c. En el capítulo 10 se tratará la sincronización de hilos, el manejo de señales y la cancelación de acuerdo con POSIX.1c.

Uno de los problemas de utilizar múltiples hilos de ejecución es que hasta hace poco no existía un estándar. La extensión POSIX.1c se aprobó en junio de 1995. Con la adopción de un estándar POSIX para los hilos, seguramente se harán más comunes las aplicaciones comerciales con hilos.

9.1 Un problema motivador: Vigilancia de descriptores de archivo

Como ejemplo motivador, en esta sección examinaremos las dificultades que se presentan cuando un programa sin hilos debe vigilar múltiples descriptores de archivo. (Si el lector ya está convencido de la utilidad de los hilos, puede pasar directamente a la sección 9.1.5.)

Cuando un proceso realiza una operación de E/S bloqueadora, como una lectura, se bloquea hasta que está disponible la entrada. El bloqueo puede crear problemas cuando un proceso espera entradas de más de una fuente, pues el proceso no tiene forma de saber cuál descriptor de archivo va a producir la siguiente entrada. Las seis estrategias generales de vigilancia de múltiples descriptores de archivo para entrada en UNIX son:

- Utilizar E/S no bloqueadora con escrutinio (polling).
- Emplear E/S asíncrona con la señal `SIGPOLL`.
- Usar E/S asíncrona POSIX.1b (descrita en el ejemplo 5.5).
- Utilizar `select` para bloquear hasta que esté disponible la entrada.
- Emplear `poll` para bloquear hasta que esté disponible la entrada.
- Crear un hilo independiente para vigilar cada descriptor de archivo.

En esta sección implementaremos cinco de las estrategias para el problema de leer y procesar entradas de dos descriptores de archivo. (En el ejemplo 5.5 se utiliza E/S asíncrona POSIX.1b para un problema similar, por lo que aquí no implementaremos un programa que utilice E/S asíncrona POSIX.) Compare los diferentes enfoques para entender las ventajas de utilizar múltiples hilos.

9.1.1 Escrutinio simple

Una operación de E/S no bloqueadora devuelve un `-1` de inmediato y asigna `EAGAIN` a `errno` si pudiera sufrir un retardo. Entonces, el proceso invocador puede intentar la operación más tarde. (En la sección 3.7 presentamos la sintaxis de la E/S no bloqueadora.) La consulta repetida de descriptores para determinar si hay datos de entrada disponible se llama *escrutinio* (polling). Esta forma de ocupado en espera (busy waiting) desperdicia ciclos de CPU y no debe utilizarse excepto cuando sólo es necesario consultar ocasionalmente los descriptores de archivo.

La función `poll_and_process` (escrutar y procesar) del programa 9.1 lee del descriptor de archivo abierto `fd`. El descriptor de archivo `fd` se abrió para lectura no bloqueadora. La función lee hasta `BLKSIZE` bytes y devuelve `-1` si hay un error o un fin de archivo; si no hay error, devuelve 0.

Programa 9.1: La función `poll_and_process` lee de un descriptor de archivo abierto.

```
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#define BLKSIZE 1024
```

```
void process_command(char *, int);

int poll_and_process(int fd)
{
   int nbytes;
   char buf[BLKSIZE];

   nbytes = read(fd, buf, BLKSIZE);
   if ( ((nbytes < 0) && (errno != EAGAIN) && (errno != EINTR))
        || !nbytes)
      return -1;
   else if (nbytes > 0)
      process_command(buf, nbytes);
   return 0;
}
```

Programa 9.1

La E/S no bloqueadora complica el manejo de códigos de error. Si `read` devuelve un valor de `-1`, puede ser que haya ocurrido un error real; sin embargo, un `read` no bloqueador también devuelve un `-1` si no había entrada disponible o si fue interrumpido por una señal. Ninguna de estas condiciones debe hacer que se suspenda el escrutinio, aunque no haya nada disponible para procesarse. La función `poll_and_process` del programa 9.1 determina si ha ocurrido cualquiera de estas condiciones probando si `errno` tiene los valores `EAGAIN` y `EINTR`, respectivamente. La función llama a `process_command` (procesar comando) sólo si en realidad hubo una entrada que procesar.

El programa 9.2 abre dos archivos para lectura no bloqueadora. Los nombres de archivo se pasan mediante la línea de comandos. A continuación, el programa vigila los descriptores de archivo escrutándolos de forma alternada hasta agotar todas las entradas. El programa llama a la función `poll_and_process` del programa 9.1 para realizar el escrutinio y procesar la información.

Programa 9.2: Un programa que escruta descriptores de archivo no bloqueadores.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stropts.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>

#define FALSE 0

int poll_and_process(int fd);
void do_some_other_stuff(void);
```

```
void main (int argc, char *argv[])
{
   int fd_1;
   int fd_2;
   int fd_1_done = FALSE;
   int fd_2_done = FALSE;

   if (argc != 3) {
      fprintf(stderr, "Usando: %s nombre de archivo1 nombre de archivo2\n", argv [0]);
      exit(1);
   }

   if ((fd_1 = open(argv[1], O_RDONLY | O_NONBLOCK)) == -1) {
      fprintf(sterr, "No puede abrir %s: %s\n",
         argv[1], strerror(errno));
      exit(1);
   }
   if ((fd_2 = open(argv[2], O_RDONLY | O_NONBLOCK)) == -1) {
      fprintf(sterr, "No puede abrir %s: %s\n",
         argv[2], strerror(errno));
      exit(1);
   }

   while(!fd_1_done || !fd_2_done) {
      if (!fd_1_done)
         fd_1_done = poll_and_process(fd_1);
      if (!fd_2_done)
         fd_2_done = poll_and_process(fd_2);
      do_some_other_stuff();
   }
   exit(0);
}
```

Programa 9.2

9.1.2 E/S asíncrona para eliminar la ocupación en espera

El programa 9.2 utiliza escrutinio para vigilar dos descriptores de archivo de entrada que se han abierto para E/S no bloqueadora. Un método más eficiente emplea E/S asíncrona. En esta estrategia, el programa no realiza una operación de E/S hasta que el sistema operativo indica que hay algo disponible mediante el envío de la señal `SIGPOLL`. `SIGPOLL` es una señal de System V que sólo se maneja para dispositivos STREAMS, pero casi toda la E/S en System V se implementa en términos de STREAMS. (Véase la sección 12.6.) La E/S asíncrona con `SIGPOLL` forma parte de la especificación 1170 pero no de POSIX. (BSD UNIX emplea la señal `SIGIO` para E/S asíncrona.)

La estrategia general para la E/S asíncrona es

- Abrir los descriptores de archivo para E/S no bloqueadora.
- Bloquear la señal `SIGPOLL`.

- Instalar un controlador de señales para atrapar la señal `SIGPOLL`. El controlador de señales se limita a establecer una variable global para indicar que llegó una señal `SIGPOLL`.
- Habilitar la señal `SIGPOLL` llamando a `ioctl` con la bandera `I_SETSIG` para cada descriptor de archivo por vigilar.
- Realizar el escrutinio y `sigsuspend` en un ciclo.

El programa 9.3 muestra cómo vigilar E/S asíncrona en dos descriptores de archivo. El programa llama a la función `poll_and_process` del programa 9.1 para que realice el escrutinio real del dispositivo. El programa bloquea la señal `SIGPOLL` antes de establecer los descriptores de archivo que generan esta señal. Después de intentar leer los descriptores de archivo, el programa entra en un ciclo y se bloquea hasta que atrapa la señal `SIGPOLL`. Esta señal se bloquea durante toda la ejecución excepto cuando `sigsuspend` está activa. El programa regresa de `sigsuspend` con su máscara de señal automáticamente restablecida al estado previo con `SIGPOLL` bloqueada.

Programa 9.3: Programa para vigilar dos descriptores de archivo empleando E/S asíncrona.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stropts.h>
#include <fcntl.h>
#include <signal.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

int poll_and_process(int fd);

int sigpoll_received = 0;

static void mypoll_handler(int signo)
{
    sigpoll_received = 1;
}

void main(int argc, char *argv[])
{
   int fd_1;
   int fd_2;
   int fd_1_done = 0;
   int fd_2_done = 0;
   sigset_t oldmask;
```

```
sigset_t newmask;
sigset_t zeromask;
struct sigaction newact;

                /* abrir descriptores de archivo para E/S no bloqueadora */
if (argc != 3) {
   fprintf(stderr, "Usage: %s filename1 filename2\n", argv[0]);
   exit(1);
}
if ((fd_1 = open(argv[1], O_RDONLY | O_NONBLOCK)) == -1) {
   fprintf(stderr,"Could not open %s: %s\n",
       argv[1], strerror(errno));
   exit(1);
}
if ((fd_2 = open(argv[2], O_RDONLY | O_NONBLOCK)) == -1) {
   fprintf(stderr,"Could not open %s: %s\n",
      argv[2], strerror(errno));
   exit(1);
}

                  /* bloquear la señal SIGPOLL y establecer controlador */
sigemptyset(&newmask);
sigaddset(&newmask, SIGPOLL);
if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) == -1) {
   perror("Could not block SIGPOLL");
   exit(1);
}
newact.sa_handler = mypoll_handler;
sigemptyset(&newact.sa_mask);
newact.sa_flags = 0;
if (sigaction(SIGPOLL, &newact, NULL) == -1) {
   perror("Could not set SIGPOLL handler");
   exit(1);
}
           /* indicar al controlador que genere SIGPOLL si llega un mensaje */
if ((ioctl(fd_1, I_SETSIG, S_INPUT | S_HANGUP) == -1) ||
    (ioctl(fd_2, I_SETSIG, S_INPUT | S_HANGUP) == -1)) {
   perror("Could not set descriptors for SIGPOLL");
   exit(1);
}
              /* ahora vigilar suspendiendo hasta que llegue SIGPOLL */
sigemptyset(&zeromask);
while(!fd_1_done || !fd_2_done) {
   if (!fd_1_done)
      fd_1_done = poll_and_process(fd_1);
   if (!fd_2_done)
      fd_2_done = poll_and_process(fd_2);
   while (!sigpoll_received && (!fd_1_done || !fd_2_done) )
      sigsuspend(&zeromask);
```

```
        sigpoll_received = 0;
    }
    exit(0);
}
```

Programa 9.3

9.1.3 Empleo de `select` para eliminar la ocupación en espera

Una alternativa a la E/S asíncrona consiste en usar `select`, una construcción de BSD para vigilar descriptores de archivo que presentamos en la sección 3.8. La llamada `select` está disponible en la mayor parte de los sistemas UNIX y forma parte de la especificación 1170 pero no de POSIX. Esta llamada utiliza máscaras distintas para lectura, escritura y condiciones excepcionales. Las máscaras indican cuáles descriptores deben vigilarse. La llamada `select` hace que el hilo de ejecución se bloquee hasta que llegue una entrada.

La función `monitor_fd` del programa 9.4 vigila un arreglo de descriptores de archivo abiertos `fd`. Cuando está disponible una entrada en el descriptor de archivo `fd[i]`, el programa lee un comando y llama a `process_command`. La función `monitor_fd` tiene tres parámetros: un arreglo de descriptores de archivo abiertos, el número de descriptores de archivo contenidos en el arreglo y un entero que es superior por lo menos en uno que el descriptor de archivo más grande de los que se van a vigilar.

Programa 9.4: Función para vigilar descriptores de archivo empleando `select`.

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

void process_command(char *, int);

/* vigilar un arreglo fd[] de descriptores de archivo abiertos empleando select */
void monitor_fd(int fd[], int num_fds, int max_fd)
{
    fd_set readset;
    int i;
    int num_ready;
    int num_now;
    int bytes_read;
    char buf[BUFSIZE];

    num_now = num_fds;
    while (num_now > 0) {
        FD_ZERO(&readset);  /* establecer la máscara de descriptor de archivo */
        for (i = 0; i < num_fds; i++)
            if (fd[i] >= 0)
                FD_SET(fd[i], &readset);
```

```
        num_ready = select(max_fd, &readset, NULL, NULL, NULL);
        for (i = 0; i < num_fds && num_ready > 0; i++)
           if ( (fd[i] >= 0) && FD_ISSET(fd[i], &readset)) {
              bytes_read = read(fd[i], buf, BUFSIZE);
              num_ready--;
              if (bytes_read > 0)
                 process_command(buf, bytes_read);
              else {
                 fd[i] = -1;
                 num_now--;
              }
           }
     }
}
```

Programa 9.4

El control en el programa 9.4 es relativamente sencillo porque las entradas de los dos descriptores de archivo se procesan de forma independiente. Si `process_command` se bloquea por alguna razón, el procesamiento de los demás descriptores de archivo esperará hasta que `process_command` termine. Para traslapar la lectura de otro descriptor durante `process_command`, el programa requeriría una lógica bastante compleja.

9.1.4 Empleo de `poll` para eliminar la ocupación en espera

La función `poll` forma parte de la versión 4 de System V y de la especificación 1170, pero no de POSIX. Esta función es similar a `select`, pero organiza la información por descriptor, no por tipo de condición. En contraste, `select` tiene diferentes máscaras de descriptor para lectura, escritura y condiciones excepcionales.

SINOPSIS

```
#include <stropts.h>
#include <poll.h>

int poll(struct pollfd *fds, size_t nfds, int timeout);
```

Spec 1170

Cada elemento del arreglo `fds` representa la información de vigilancia de un descriptor de archivo. El parámetro `nfds` indica el número de descriptores de archivo por vigilar. La `struct pollfd` incluye los siguientes miembros:

```
int fd;          /* descriptor de archivo */
short events;    /* sucesos solicitados */
short revents;   /* sucesos devueltos */
```

`fd` es el número de descriptor de archivo, y `events` y `revents` se construyen aplicando un OR lógico a las banderas que representan diversas condiciones. Si se hace que `events` con-

tenga los sucesos que se desea vigilar, `poll` llenará `revents` con los sucesos que han ocurrido. La página del manual que corresponde a `poll` describe las banderas asociadas a los sucesos de descriptor. Aquí sólo consideraremos la lectura normal de datos de descriptores. El valor `timeout` está en milisegundos. El valor especial `INFTIM` se usa cuando no se desea que `poll` tenga límite de tiempo. La llamada `select` modifica los conjuntos de descriptores de archivos que le son pasados y el programa debe restablecer estos conjuntos de descriptores cada vez que invoca a `select`. La función `poll` emplea variables independientes para los valores de entrada y devueltos, por lo que no es necesario restablecer la lista de descriptores vigilados después de cada llamada a `poll`. Esta función devuelve el número de descriptores de archivos que están listos, o `-1` si ocurre algún error.

El programa 9.5 implementa `monitor_fd` en términos de `poll`. La función `poll` tiene varias ventajas. No es necesario restablecer las máscaras después de cada invocación. A diferencia de `select`, `poll` interpreta los errores como sucesos que hacen que `poll` regrese; además, no necesita un argumento `max_fd`.

Programa 9.5: Función para vigilar un arreglo de descriptores de archivo empleando `poll`.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stropts.h>
#include <poll.h>
#include <errno.h>

void process_command(char *, int);

void monitor_fd(int fd[], int num_fds)
{
   int i;
   int num_ready;
   int num_now;
   int bytes_read;
   char buf[BUFSIZE];
   struct pollfd *myfd;
   short errormask;

   errormask = POLLERR|POLLHUP;

                                   /* iniciar la estructura de escrutinio */
   if ((myfd = (void *)calloc(num_fds, sizeof(struct pollfd))) == NULL)
      return;
   for (i = 0; i < num_fds; i++) {
      (myfd + i)->fd = *(fd + i);
      (myfd + i)->events = POLLRDNORM;
```

```
        (myfd + i)->revents = 0;
    }

    num_now = num_fds;

         /* Seguir vigilando hasta completar la E/S en todos los descriptores */
    while (num_now > 0) {
        if ( ((num_ready = poll(myfd, num_fds, INFTIM)) == -1) &&
             (errno != EINTR) )
            break;

        for (i = 0; i < num_fds && num_ready > 0; i++)  {
            if ( (myfd + i)->events && (myfd + i)->revents ) {
                    /* Si ocurrió un error, dejar de vigilar fd */
                if ((myfd + i)->revents & errormask) {
                    num_now--;
                    (myfd + i)->events = 0;
                }
                            /* Si llegaron datos normalmente, leerlos */
                else if ((myfd + i)->revents & POLLRDNORM) {
                    bytes_read = read(fd[i], buf, BUFSIZE);
                    num_ready--;
                    if (bytes_read > 0)
                        process_command(buf, bytes_read);
                    else {
                        num_now--;
                        (myfd + i)->events = 0;
                    }
                }
            }
        }
        if (!num_now) break;
    }
}
```

Programa 9.5

9.1.5 Hilos múltiples

El empleo de hilos múltiples puede simplificar la programación de problemas como la vigilancia y procesamiento de entradas de descriptores de archivo múltiples porque un hilo dedicado puede vigilar cada descriptor. Además, los hilos ofrecen la posibilidad de traslapar E/S y procesamiento, cosa que no es posible con los otros enfoques.

La función `process_fd` del programa 9.6 vigila un descriptor de archivo empleando E/S bloqueadora. La función regresa cuando encuentra un fin de archivo o cuando detecta la ocurrencia de un error en el descriptor. El descriptor de archivo se pasa mediante un apuntador a `void` de modo que `process_fd` se puede invocar, ya sea como una función ordinaria o como un hilo.

Programa 9.6: La función `process_fd` vigila un descriptor de archivo para detectar entradas.

```
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
void process_command(char *, int);

void *process_fd(void *arg)
{
   int fd;
   int nbytes;
   char buf[BUFSIZE];

   fd = *((int *)(arg));
   for ( ; ; )  {
      if (((nbytes = read(fd, buf, BUFSIZE)) == -1) &&
           (errno != EINTR))
         break;
      if (!nbytes)
         break;
      process_command(buf, nbytes);
   }
   return NULL;
}
```

Programa 9.6

Ejemplo 9.3

El siguiente segmento de código llama a `process_fd` *como función ordinaria.*

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
void *process_fd(void *);
int fd_1;

if ((fd_1 = open("my.dat", O_RDONLY)) == -1)
   perror("Could not open my.dat");
else process_fd(&fd_1);
```

La figura 9.1 muestra el hilo de ejecución cuando se invoca el programa 9.6 como función ordinaria. El hilo de ejecución del programa invocador recorre los enunciados de la función y

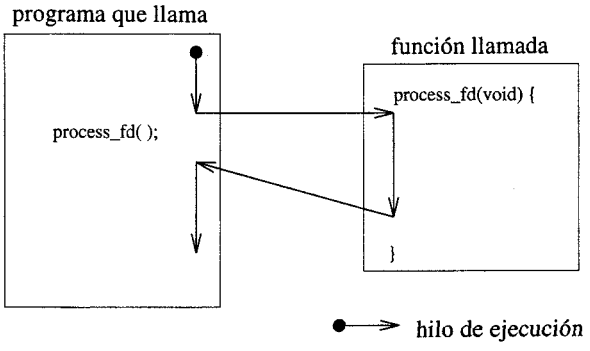


Figura 9.1: Hilo de ejecución de una llamada ordinaria a `process_fd`.

luego continúa la ejecución en el enunciado que sigue a la llamada. Puesto que `process_fd` utiliza E/S bloqueadora, el hilo de ejecución se bloquea en el `read` hasta que hay una entrada disponible en el descriptor de archivo. Recuerde que el hilo de ejecución está representado por la secuencia de enunciados que el hilo ejecuta. No hay información de temporización en esta secuencia. El hecho de que la ejecución se bloquea en un `read` es evidente.

Un programa también puede crear un hilo independiente para ejecutar `process_fd` como se muestra en la figura 9.2. El hilo se separa y ejecuta un flujo de instrucciones independiente, y nunca vuelve al punto de invocación. El programa invocador se sigue ejecutando de forma concurrente. En contraste, cuando se invoca `process_fd` como función ordinaria, el hilo de ejecución del invocador pasa por el código de la función y vuelve al punto de invocación.

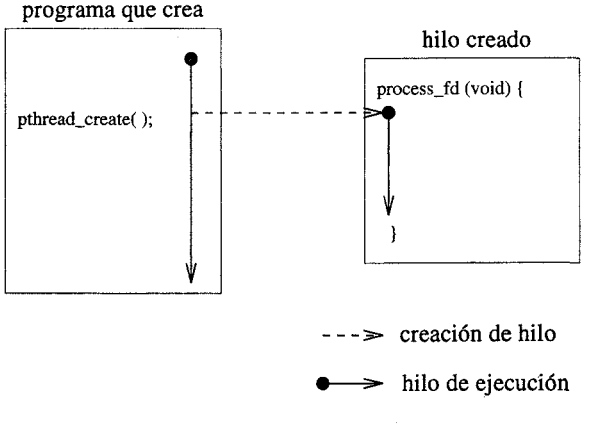


Figura 9.2: Un hilo nuevo ejecuta `process_fd`.

Ejemplo 9.4

El siguiente segmento de código crea un nuevo hilo para ejecutar `process_fd`.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
#include <string.h>
#include <errno.h>

pthread_t tid;
int fd;
void *process_fd(void *arg);

if ((fd = open("my.dat", O_RDONLY)) == -1)
   perror("Could not open my.dat");
else if (pthread_create(&tid, NULL, process_fd, (void *)&fd))
   perror("Could not create thread");
else
   pthread_join(tid, NULL);
```

Lo interesante de los hilos es que un programa puede crear muchos de ellos sin trabajar demasiado. En el resto del capítulo estudiaremos los mecanismos para crear hilos, así que no se preocupe demasiado por la sintaxis aquí. La función `monitor_fd` del programa 9.7 utiliza hilos para vigilar un arreglo de descriptores de archivo. Compare esta implementación con las de los programas 9.4 y 9.5. La versión con hilos aprovecha el paralelismo sin que el usuario tenga que intervenir. Si `process_command` hace que el hilo que llama se bloquee por alguna razón, el sistema de tiempo de ejecución de hilos programará otro hilo listo para ejecución (ready). De este modo, el procesamiento y la lectura se traslapan naturalmente. En contraste, si sólo hay un hilo de ejecución, un bloqueo en `process_command` hace que todo el proceso se bloquee.

Programa 9.7: Función para vigilar un arreglo de descriptores de archivo empleando hilos.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <pthread.h>
#include <string.h>
#include <errno.h>

void process_fd(void *arg);
```

```
void monitor_fd(int fd[], int num_fds)
{
   int i;
   pthread_t *tid;

   if ((tid = (pthread_t *)calloc(num_fds, sizeof(pthread_t))) == NULL)
      return;
                    /* crear un hilo para cada descriptor de archivo */
   for (i = 0; i < num_fds; i++) {
      if (pthread_create((tid + i), NULL, process_fd, (void *)(fd + i)))
          fprintf(stderr, "Could not create thread %i: %s\n", i,
                  strerror(errno));
   }
   for (i = 0; i < num_fds; i++)
       pthread_join(*(tid + i), NULL);
}
```

Programa 9.7

9.2 Hilos POSIX

Un paquete de hilos representativo contiene un sistema de tiempo de ejecución para gestionar los hilos de forma transparente (es decir, sin que el usuario se dé cuenta de la existencia del sistema de tiempo de ejecución). El paquete de hilos por lo regular incluye llamadas para crear y destruir hilos, exclusión mutua y variables de condición. Las bibliotecas de hilos tanto de Sun Solaris 2 como del estándar POSIX.1c cuentan con llamadas. Estos paquetes contemplan la creación y destrucción dinámicas de hilos, así que no es necesario conocer el número de hilos antes de iniciarse la ejecución. En la tabla 9.1 se resumen las funciones de hilos comunes en POSIX y en Sun Solaris 2.

La mayor parte de las funciones de hilos devuelven 0 si tuvieron éxito y un código de error distinto de cero si no fue así. La función `pthread_create` (`thr_create`) crea un hilo para ejecutar una función especificada; `pthread_exit` (`thr_exit`) hace que el hilo invocador termine sin causar que todo el proceso llegue a su fin. La función `pthread_kill` (`thr_kill`) envía una señal a un hilo especificado; `pthread_join` (`thr_join`) hace que el hilo invocador espere a que termine el hilo especificado. Esta llamada es similar a `waitpid` en el nivel de procesos. Por último, `pthread_self` (`thr_self`) devuelve la identidad del invocador. El resto de las llamadas de la tabla tienen que ver con mecanismos de sincronización que veremos en el siguiente capítulo.

Los hilos POSIX.1c y Sun Solaris son muy similares. Una diferencia importante la constituye la forma en que se asocian propiedades a los hilos. Los hilos POSIX se valen de objetos atributos para representar propiedades de los hilos. Las propiedades como el tamaño de pila o la política de programación se establecen para un objeto atributo de hilo. Varios hilos pueden estar asociados al mismo objeto atributo de hilo. Si una propiedad del objeto cambia, se refleja en todos los hilos asociados a ese objeto. Los hilos Solaris establecen explícitamente las propie-

Descripción	POSIX	Solaris 2
Gestión de hilos	pthread_create	thr_create
	pthread_exit	thr_exit
	pthread_kill	thr_kill
	pthread_join	thr_join
	pthread_self	thr_self
Exclusión mutua	pthread_mutex_init	mutex_init
	pthread_mutex_destroy	mutex_destroy
	pthread_mutex_lock	mutex_lock
	pthread_mutex_trylock	mutex_trylock
	pthread_mutex_unlock	mutex_unlock
Variables de condición	pthread_cond_init	cond_init
	pthread_cond_destroy	cond_destroy
	pthread_cond_wait	cond_wait
	pthread_cond_timedwait	cond_timedwait
	pthread_cond_signal	cond_signal
	pthread_cond_broadcast	cond_broadcast

Tabla 9.1: Comparación de llamadas para hilos POSIX.1c y para hilos Sun Solaris 2.

dades de los hilos y otras primitivas, de modo que algunas llamadas tienen largas listas de parámetros para establecer dichas propiedades. Los hilos Solaris ofrecen mayor control sobre la correspondencia entre los hilos y los recursos de procesadores, mientras que los hilos POSIX ofrecen un método más robusto de cancelación y de terminación de hilos. En este libro nos concentraremos en los hilos POSIX.

9.3 Gestión de hilos básica

Un hilo tiene un identificador (ID), una pila, una prioridad de ejecución y una dirección de inicio de la ejecución. Hacemos referencia a los hilos POSIX mediante un ID de tipo `pthread_t`. Un hilo puede averiguar su ID llamando a `pthread_self`. La estructura de datos interna del hilo también puede contener información de programación y uso. Los hilos de un proceso comparten todo el espacio de direcciones de ese proceso; pueden modificar variables globales, acceder a descriptores de archivo abiertos o interferirse mutuamente de otras maneras.

Se dice que un hilo es *dinámico* si se puede crear en cualquier instante durante la ejecución de un proceso y si no es necesario especificar por adelantado el número de hilos. En POSIX, los hilos se crean dinámicamente con la función `pthread_create`, la cual crea un hilo y lo coloca en una cola de hilos preparados.

SINOPSIS

```
#include <pthread.h>

int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
void pthread_exit(void *value_ptr);
int pthread_join(pthread_t thread, void **value_ptr);
```

POSIX.1c

El parámetro `tid` de `pthread_create` apunta al ID del hilo que se crea. Los atributos del hilo se encapsulan en el objeto atributo al que apunta `attr`. Si `attr` es `NULL`, el nuevo hilo tendrá los atributos por omisión. El tercer parámetro, `start_routine`, es el nombre de una función a la que el hilo invoca cuando inicia su ejecución. `start_routine` requiere un solo parámetro que se especifica con `arg`, un apuntador a `void`. La función `start_routine` devuelve un apuntador a `void` que `pthread_join` trata como una situación de salida. No se asuste al ver el prototipo de `pthread_create`; es muy fácil crear y utilizar hilos.

La función `pthread_exit` termina el hilo que la invoca. El valor del parámetro `value_ptr` queda disponible para `pthread_join` si ésta tuvo éxito. Sin embargo, el `value_ptr` en `pthread_exit` debe apuntar a datos que existan después de que el hilo ha terminado, así que no puede asignarse como datos locales automáticos para el hilo que está terminando.

Ejemplo 9.5

El siguiente segmento de código crea un hilo con los atributos por omisión.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
pthread_t copy_tid;
int myarg[2];
void *copy_file(void *arg);

if ((myarg[0] = open("my.in", O_RDONLY)) == -1)
   perror("Could not open my.in");
else if ((myarg[1] = open("my.out",
      O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) == -1)
   perror("Could not open my.out");
else if (pthread_create(&copy_tid, NULL, copy_file, (void *)myarg))
   perror("Thread creation was not successful");
```

En el ejemplo 9.5, `copy_tid` contiene el ID del hilo que se creó y `copy_file` (copiar archivo) es el nombre de la función que el hilo debe ejecutar. `myarg` es un apuntador al valor de parámetro que debe pasarse a la función de hilo. En este caso el arreglo `myarg` contiene descriptores de archivo abiertos para los archivos `my.in` y `my.out`.

El programa 9.8 muestra una implementación de una función `copy_file` que lee de un archivo y envía la salida a otro archivo. El parámetro `arg` contiene un apuntador a un par de descriptores abiertos que representan los archivos de origen y de destino. A las variables `infile` (archivo de entrada), `outfile` (archivo de salida), `bytes_read` (bytes leídos), `bytes_written` (bytes escritos), `bytes_copied_p` (apuntador a bytes copiados), `buffer` y `bufp` (apuntador al *buffer*) se les asigna espacio en la pila local de `copy_file` y no están accesibles directamente a otros hilos. El hilo también asigna espacio con `malloc` para devolver el número total de bytes copiados. La implementación supone que `malloc` es segura respecto a los hilos; de no ser así, debe utilizarse una versión `malloc_r`. La función `copy_file` podría devolver el apuntador `bytes_copied` en lugar de llamar a `pthread_exit`. La función `pthread_exit` invoca controladores de terminación de hilos, cosa que `return` no hace.

Programa 9.8: La función `copy_file` copia los contenidos de `infile` en `outfile`.

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>

#define BUFFERSIZE 100

void *copy_file(void *arg)
{
   int infile;
   int outfile;
   int bytes_read = 0;
   int bytes_written = 0;
   int *bytes_copied_p;
   char buffer[BUFFERSIZE];
   char *bufp;
         /* abrir descriptores para los archivos de origen y de destino */
   infile = *((int *)(arg));
   outfile = *((int *)(arg) + 1);
   if ((bytes_copied_p = (int *)malloc(sizeof(int))) == NULL)
      pthread_exit(NULL);
   *bytes_copied_p = 0;

   for ( ; ; ) {
      bytes_read = read(infile, buffer, BUFFERSIZE);
      if ((bytes_read == 0) || ((bytes_read < 0) && (errno != EINTR)))
         break;
      else if ((bytes_read < 0) && (errno == EINTR))
         continue;
      bufp = buffer;
      while (bytes_read > 0) {
         bytes_written = write(outfile, bufp, bytes_read);
```

```
            if ((bytes_written < 0) && (errno != EINTR))
               break;
            else if (bytes_written < 0)
               continue;
            *bytes_copied_p += bytes_written;
            bytes_read -= bytes_written;
            bufp += bytes_written;
         }
         if (bytes_written == -1)
            break;
      }
      close(infile);
      close(outfile);
      pthread_exit(bytes_copied_p);
}
```

Programa 9.8

El programa 9.9 muestra un programa principal con tres argumentos de línea de comandos, un nombre base de archivo de entrada, un nombre base de archivo de salida y el número de archivos copiadores. El programa crea `numcopiers` hilos. El hilo `i` copia `infile_name.i` en `outfile_name.i`.

Programa 9.9: Programa que crea hilos para copiar múltiples descriptores de archivo.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>

#define MAXNUMCOPIERS 10
#define MAXNAMESIZE 80

void *copy_file(void *arg);

void main(int argc, char *argv[])
{
   pthread_t copiertid[MAXNUMCOPIERS];
   int fd[MAXNUMCOPIERS][2];
   char filename[MAXNAMESIZE];
   int numcopiers;
   int total_bytes_copied=0;
   int *bytes_copied_p;
```

```
   int i;

   if (argc != 4) {
      fprintf(stderr, "Usage: %s infile_name outfile_name copiers\n",
         argv[0]);
      exit(1);
   }
   numcopiers = atoi(argv[3]);
   if (numcopiers < 1 || numcopiers > MAXNUMCOPIERS) {
      fprintf(stderr, "%d invalid number of copiers\n", numcopiers);
      exit(1);
   }
                                          /* crear los hilos copiadores */
   for (i = 0; i < numcopiers; i++) {
      sprintf(filename, "%s.%d", argv[1], i);
      if ((fd[i][0] = open(filename, O_RDONLY)) < 0) {
         fprintf(stderr, "Unable to open copy source file %s: %s\n",
                         filename, strerror(errno));
         continue;
      }
      sprintf(filename, "%s.%d", argv[2], i);
      if ((fd[i][1]=
         open(filename, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) < 0) {
         fprintf(stderr,
            "Unable to create copy destination file %s: %s\n",
            filename, strerror(errno));
         continue;
      }
      if (pthread_create(&copiertid[i], NULL, copy_file,
            (void *)fd[i]) != 0)
         fprintf(stderr, "Could not create thread %i: %s\n",
            i, strerror(errno));
    }
                                          /* esperar que se termine de copiar */
    for (i = 0; i < numcopiers; i++) {
       if (pthread_join(copiertid[i], (void **)&(bytes_copied_p)) != 0)
          fprintf(stderr, "No thread %d to join: %s\n",
             i, strerror(errno));
       else {
          printf("Thread %d copied %d bytes from %s.%d to %s.%d\n",
             i, *bytes_copied_p, argv[1], i, argv[2], i);
          total_bytes_copied += *bytes_copied_p;
       }
    }
    printf("Total bytes copied = %d\n", total_bytes_copied);
    exit(0);
}
```

Programa 9.9

Cuando un hilo `copy_file` completa su trabajo, termina invocando a `pthread_exit`. La situación de terminación de un hilo que llama a `pthread_exit` se conserva hasta que otro hilo se une a él o si ya es el último hilo del proceso. La función `pthread_join` es similar a `waitpid` para procesos hijo en cuanto a que el hilo invocador se bloquea hasta que el hilo indicado termina. El hilo invocador recupera el número de bytes copiado por el hilo a través del `status_value` devuelto por `pthread_join`. El hilo asigna espacio dinámicamente para `status_value` de modo que la variable persista después de que el hilo termine. Por último, el hilo principal termina.

Los hilos tienen un atributo `detachstate` (estado de desconexión) de `PTHREAD_CREATE_JOINABLE` (unible) por omisión. El otro valor que `detachstate` puede tener es `PTHREAD_CREATE_DETACHED` (desconectado). Los hilos desconectados deben llamar a `pthread_detach` en lugar de a `pthread_exit` para liberar sus recursos.

Ejercicio 9.1

¿Qué sucede en el programa 9.9 si falla la `malloc` de `copy_file`?

Respuesta:

En el programa principal después del retorno de `pthread_join`, `bytes_copied_p` es `NULL` y el programa se cae cuando trata de obtener la dirección a la que apunta este apuntador. El problema puede corregirse haciendo que el programa principal verifique si este apuntador es `NULL`.

En las implementaciones tradicionales de UNIX, `errno` es una variable global externa que se establece cuando las funciones del sistema producen un error. Esta implementación no funciona con multihilos (véase la sección 1.5), y en la mayor parte de las implementaciones de hilos `errno` es una macro que devuelve información específica para un hilo. En esencia, cada hilo tiene su propia copia de `errno`. El programa principal no tiene acceso directo al `errno` de un hilo unido, de modo que si se necesita esta información se deberá devolver a través del último parámetro de `pthread_join`.

Ejercicio 9.2

¿Qué sucede en el programa 9.9 si el `write` de `copy_file` fracasa?

Respuesta:

La función `copy_file` devuelve el número de bytes copiados con éxito y el programa principal no detecta ningún error. El problema puede corregirse haciendo que `copy_file` devuelva un apuntador a una estructura que contenga tanto el número de bytes copiados como un valor de error.

Un aspecto delicado es el paso de parámetros cuando hay múltiples hilos. En el programa 9.9 se crean varios hilos, pero a cada hilo le son pasados sus descriptores de archivo en diferentes entradas del arreglo `fd`, así que los hilos no se interfieren. Tenga cuidado al reutilizar variables que se pasan por referencia a los hilos cuando éstos se crean. Podría suceder que el hilo creado no se programara para ejecutarse a tiempo a fin de utilizar los valores antes de que se sobreescriban.

El programa 9.10 muestra una modificación del programa 9.9 que utiliza un solo par de posiciones para los descriptores de archivo. Este programa fracasa si el descriptor de archivo para el hilo `i` es sobreescrito por la siguiente iteración del ciclo `for` antes de que el hilo `i` haya tenido oportunidad de copiar el descriptor en su memoria local.

Programa 9.10: Programa que pasa incorrectamente múltiples descriptores de archivo a hilos.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>

void *copy_file(void *arg);
#define MAXNUMCOPIERS 10
#define MAXNAMESIZE 80

void main(int argc, char *argv[])
{
   pthread_t copiertid[MAXNUMCOPIERS];
   int fd[2];
   char filename[MAXNAMESIZE];
   int numcopiers;
   int total_bytes_copied=0;
   int *bytes_copied_p;
   int i;

   if (argc != 4) {
      fprintf(stderr, "Usage: %s infile_name outfile_name copiers\n",
         argv[0]);
      exit(1);
   }
   numcopiers = atoi(argv[3]);
   if (numcopiers < 1 || numcopiers > MAXNUMCOPIERS) {
      fprintf(stderr, "%d invalid number of copiers\n", numcopiers);
      exit(1);
   }

                              /* crear los hilos copiadores */
   for (i = 0; i < numcopiers; i++) {
      sprintf(filename, "%s.%d", argv[1], i);
      if ((fd[0] = open(filename, O_RDONLY)) < 0) {
          fprintf(stderr, "Unable to open copy source file %s: %s\n",
                           filename, strerror(errno));
```

```
            continue;
        }
        sprintf(filename, "%s.%d", argv[2], i);
        if ((fd[1] =
            open(filename, O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)) < 0) {
            fprintf(stderr,
                "Unable to create copy destination file %s: %s\n",
                filename, strerror(errno));
            continue;
        }
        if (pthread_create(&copiertid[i], NULL, copy_file,
                (void *)fd) != 0)
            fprintf(stderr, "Could not create thread %i: %s\n",
                i, strerror(errno));
    }
                                    /* esperar que se termine de copiar */
    for (i = 0; i < numcopiers; i++) {
        if (pthread_join(copiertid[i], (void **)&(bytes_copied_p)) != 0)
            fprintf(stderr, "No thread %d to join: %s\n",
                i, strerror(errno));
        else {
            printf("Thread %d copied %d bytes from %s.%d to %s.%d\n",
                i, *bytes_copied_p, argv[1], i, argv[2], i);
            total_bytes_copied += *bytes_copied_p;
        }
    }
    printf("Total bytes copied = %d\n", total_bytes_copied);
    exit(0);
}
```

Programa 9.10

Ejercicio 9.3

Pruebe el código del programa 9.10. Haga que se exhiba en la pantalla cada descriptor de archivo cuando sea abierto y al principio de la ejecución hilo al cual se pasó como parámetro. ¿Son iguales los dos valores del descriptor? ¿Qué sucede si se inserta un `sleep(5)` dentro del ciclo `for` después de `pthread_create`?

Respuesta:

Se abre un par diferente de descriptores de archivo para cada hilo, pero el arreglo `fd` se reutiliza con cada hilo. Todos los hilos se crean con la misma prioridad que el hilo principal y se colocan en la cola de hilos listas para ejecución (ready). Si la siguiente iteración del ciclo comienza antes de que se programe para ejecución el hilo creado por la iteración anterior, se sobreescribirán los descriptores de archivo y el hilo copiará los archivos equivocados. Si se coloca un `sleep` después de `pthread_create`, es muy probable que el hilo tenga oportunidad de ejecutarse antes de que esto suceda y el programa deberá funcionar correctamente.

Los hilos copiadores individuales del programa 9.9 trabajan con problemas independientes y no iteractúan entre sí. En aplicaciones más complicadas puede suceder que un hilo no termine después de completar su tarea asignada. En vez de ello, un hilo trabajador podría solicitar tareas adicionales o compartir información. En el capítulo 10 se explica cómo puede controlarse este tipo de interacción mediante mecanismos de sincronización como los candados mutex y las variables de condición.

Un problema poco oculto del uso de hilos es que éstos pueden invocar funciones de biblioteca o hacer llamadas de sistema que no sean seguras respecto a los hilos, produciendo tal vez resultados indeseables. Incluso funciones como `sprintf` y `fprintf` podrían no ser seguras respecto de los hilos, así que tenga cuidado. POSIX.1c especifica que todas las funciones requeridas, incluida la biblioteca de C estándar, se implemente en forma segura respecto a los hilos. Aquellas funciones cuyas interfaces tradicionales impiden hacerlas seguras respecto a los hilos deben contar con una versión alternativa segura designada con un sufijo `_r`. Las páginas del manual de Sun Solaris 2 casi siempre indican si una función es o no segura respecto de los hilos en el apartado `MT_LEVEL`.

9.4 Usuario de hilos *versus* hilos de núcleos (*kernel*)

Recuerde que un procesador funciona ejecutando su ciclo de instrucción y que el valor del contador de programa determina cuál proceso se está ejecutando. El sistema operativo tiene oportunidades de recuperar el control modificando el valor del contador de programa cuando ocurren interrupciones y cuando los programas solicitan servicios mediante llamadas de sistema. Cuando se usan hilos, surgen cuestiones de control similares en el nivel de proceso. Los dos modelos tradicionales de control de hilos son el de *hilos a nivel de usuario* y el de *hilos a nivel de núcleo (kernel)*.

Los paquetes de hilos a nivel de usuario suelen ejecutarse sobre un sistema operativo existente. Los hilos dentro del proceso son invisibles para el núcleo. Estos hilos compiten entre sí por los recursos asignados a un proceso, como se aprecia en la figura 9.3. Los hilos son programados por un sistema de hilos de tiempo de ejecución que forma parte del código del proceso. Los programas que utilizan un paquete de hilos a nivel de usuario por lo regular se ligan con una biblioteca especial en la que cada función de biblioteca y llamada de sistema está dentro de una envoltura. El código de envoltura llama al sistema de tiempo de ejecución para que se encargue de la gestión de hilos.

Las llamadas de sistema como `read` o `sleep` podrían representar un problema para hilos a nivel de usuario porque pueden hacer que el proceso se bloquee. A fin de evitar el problema de que una llamada bloqueadora haga que todo el proceso se bloquee, cada llamada potencialmente bloqueadora se sustituye en la envoltura por una versión no bloqueadora. El sistema de hilos de tiempo de ejecución verifica si la llamada puede hacer que el hilo se bloquee. Si no es así, el sistema de tiempo de ejecución realiza la llamada de inmediato. Sin embargo, si la llamada bloquearía el hilo, el sistema de tiempo de ejecución bloquea el hilo, añade la llamada a una lista de cosas que intentará más tarde y escogerá otro hilo para ejecutarlo. Todo este control queda fuera de la vista del usuario y del sistema operativo.

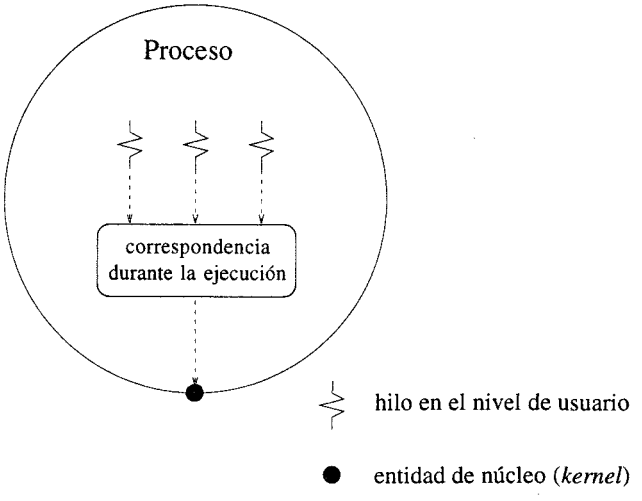


Figura 9.3: Los hilos en el nivel de usuario no pueden verse desde fuera del proceso.

Los hilos a nivel de usuario tienen muy poco gasto extra, pero presentan algunas desventajas. El modelo necesita tener hilos que permitan al sistema de hilos de tiempo de ejecución recuperar el control. Un *hilo limitado por la CPU* rara vez efectúa llamadas de sistema o de biblioteca y por tanto evita que el sistema de hilos de tiempo de ejecución recupere el control para programar otros hilos. El programador debe evitar la situación de exclusión obligando explícitamente a los hilos limitados por la CPU a ceder el control en los puntos apropiados. Un segundo problema de los hilos a nivel de usuario, más grave, es que los hilos sólo pueden compartir recursos de procesador asignados a su proceso encapsulante. Esta restricción limita el grado de paralelismo porque los hilos sólo se pueden ejecutar en un procesador a la vez. Puesto que una de las principales razones para usar hilos es aprovechar el paralelismo que ofrecen las estaciones de trabajo de multiprocesador, los hilos a nivel de usuario por sí solos no son una estrategia aceptable.

Con los hilos a nivel de núcleo *(kernel)*, el núcleo está consciente de cada hilo como una entidad programable. Los hilos compiten por los recursos de procesador en todo el sistema, como se muestra en la figura 9.4. La programación de la ejecución de los hilos a nivel de núcleo puede ser tan costosa como la programación de los procesos mismos, pero los hilos a nivel de núcleo pueden aprovechar la existencia de múltiples procesadores. La sincronización y el compartimiento de datos que los hilos a nivel de núcleo hacen posible es menos costosa que en el caso de procesos completos, pero estos hilos son mucho más costosos que los hilos a nivel de usuario.

Los *modelos de hilos híbridos* ofrecen las ventajas de los hilos tanto a nivel de usuario como a nivel de núcleo al proveer dos niveles de control. La figura 9.5 muestra un enfoque híbrido representativo. El usuario escribe el programa en términos de hilos a nivel de usuario y luego especifica cuántas entidades programables por el núcleo están asociadas al proceso. Durante la ejecución del proceso se establece una correspondencia entre los hilos a nivel de

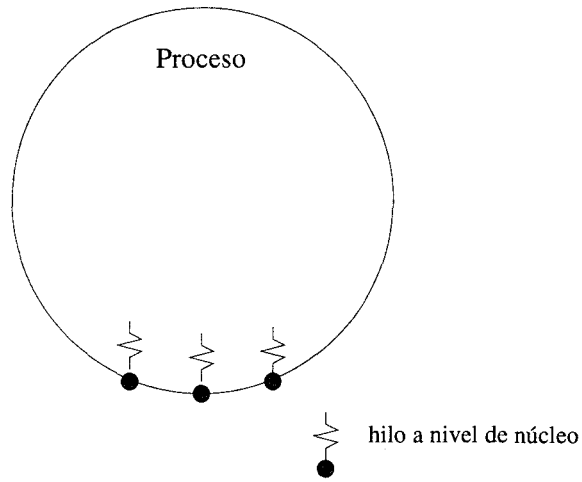


Figura 9.4: Los hilos a nivel de núcleo *(kernel)* se programan igual que los procesos individuales.

usuario y las entidades programables por el núcleo a fin de lograr el paralelismo. El grado de control que un usuario tiene sobre esta correspondencia depende de la implementación. En la implementación de hilos Sun Solaris, por ejemplo, los hilos a nivel de usuario se llaman hilos y las entidades programables por el núcleo se denominan *procesos ligeros (light weigth processes)*. El usuario puede especificar que un hilo en particular tenga un proceso ligero dedicado o que cierto grupo de hilos sea ejecutado por un conjunto de procesos ligeros.

El modelo de programación de hilos POSIX.1c es un modelo híbrido con la suficiente flexibilidad para apoyar hilos tanto a nivel de usuario como a nivel de núcleo en implementaciones específicas del estándar. El modelo consiste en dos niveles de programación: hilos y entidades de núcleo. Los hilos son análogos a aquellos a nivel de usuario. Las entidades de núcleo son programadas por el núcleo. La biblioteca de hilos decide cuántas entidades de núcleo necesita y cómo se establecerá la correspondencia con ellas.

POSIX.1c introduce la idea de *alcance de contención de programación de hilos* que proporciona al programador cierto control sobre la correspondencia entre las entidades de núcleo y los hilos. Un hilo puede tener un atributo `contentionscope` (alcance de contención) de `PTHREAD_SCOPE_PROCESS` (proceso) o `PTHREAD_SCOPE_SYSTEM` (sistema). Los hilos que son `PTHREAD_SCOPE_PROCESS` compiten con los demás hilos de su proceso por recursos de procesador. POSIX no especifica cómo un hilo de este tipo compite con los hilos externos a su propio proceso, así que los hilos `PTHREAD_SCOPE_PROCESS` pueden ser hilos estrictamente a nivel de usuario o bien pueden hacerse corresponder con un conjunto de entidades de núcleo de alguna otra forma más complicada.

Los hilos que son `PTHREAD_SCOPE_SYSTEM` compiten por los recursos de procesador dentro de todo el sistema, en forma parecida a como lo hacen los hilos a nivel de núcleo. POSIX deja la correspondencia entre los hilos `PTHREAD_SCOPE_SYSTEM` y las entidades de núcleo a

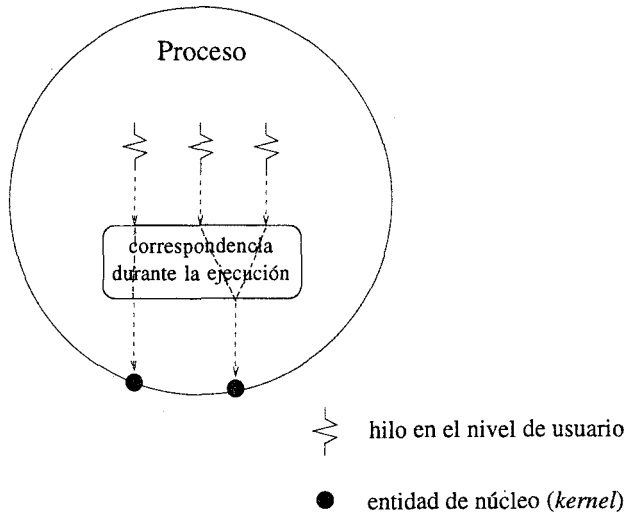


Figura 9.5: El modelo híbrido tiene dos niveles de programación, con una correspondencia entre los hilos a nivel de usuario y las entidades de núcleo.

la implementación, pero la correspondencia obvia es vincular un hilo de este tipo directamente con una entidad de núcleo. Una implementación de hilos POSIX puede apoyar hilos `PTHREAD_SCOPE_PROCESS` o `PTHREAD_SCOPE_SYSTEM`, o ambos.

En la tabla 9.2 se muestra el costo relativo de los hilos a nivel de usuario y a nivel de núcleo, como se presenta en el *Sun Solaris 2.3 Software Developer Answerbook*. Solaris 2 emplea un modelo de hilos de dos niveles similar a la especificación POSIX. En la terminología de Sun Microsystems, un *hilo no vinculado (unbound thread)* es un hilo a nivel de usuario, y un *hilo vinculado (bound thread)* es un hilo a nivel de núcleo porque está vinculado con un proceso ligero. El `fork` es el costo de la creación de un proceso completo. La sincronización se refiere a que dos hilos se sincronizan con semáforos como en el problema del productor-consumidor. Cuesta entre seis y siete veces más crear y sincronizar un hilo a nivel de núcleo que uno a nivel de usuario. Cuesta alrededor de treinta veces más crear un proceso completo con un `fork` que crear un hilo a nivel de usuario.

9.5 Atributos de los hilos

POSIX.1c adopta un enfoque orientado a objetos respecto de la representación y asignación de propiedades. Cada hilo POSIX.1c tiene un objeto atributo asociado que representa sus propiedades. Un objeto atributo de hilo puede estar asociado a varios hilos, y POSIX.1c tiene funciones para crear, configurar y destruir objetos atributo. El enfoque orientado a objetos permite a un programa agrupar entidades tales como los hilos y asociar el mismo objeto atributo a todos los miembros del grupo. Cuando cambia una propiedad del objeto atributo, todas las entidades

Operación	Microsegundos
Crear hilo no vinculado	52
Crear hilo vinculado	350
`fork()`	1700
Sincronizar hilo no vinculado	66
Sincronizar hilo vinculado	390
Sincronizar entre procesos	200

Tabla 9.2: Tiempos de los servicios de hilos con Solaris 2.3 en una SPARCstation 2.

del grupo tienen la nueva propiedad. Los objetos atributo de los hilos son de tipo `pthread_attr_t`. La tabla 9.3 muestra las propiedades de los atributos de hilo que se pueden establecer y las funciones asociadas a las propiedades. Otras entidades, como las variables de condición o los candados mutex, tienen sus propios tipos de objeto atributo y funciones, como se verá en el capítulo 10.

La función `pthread_attr_init` inicializa un objeto atributo de hilos con los valores por omisión. La función `pthread_attr_destroy` hace que el valor del objeto atributo sea no válido. POSIX no especifica el comportamiento del objeto una vez que ha sido destruido.

Propiedad	Función
Inicialización	`pthread_attr_init`
	`pthread_attr_destroy`
Tamaño de pila	`pthread_attr_setstacksize`
	`pthread_attr_getstacksize`
Dirección de pila	`pthread_attr_setstackaddr`
	`pthread_attr_getstackaddr`
Estado de desconexión	`pthread_attr_setdetachstate`
	`pthread_attr_getdetachstate`
Alcance	`pthread_attr_setscope`
	`pthread_attr_getscope`
Herencia	`pthread_setinheritsched`
	`pthread_getinheritsched`
Política de programación	`pthread_attr_setschedpolicy`
	`pthread_attr_getschedpolicy`
Parámetros de programación	`pthread_attr_setschedparam`
	`pthread_attr_getschedparam`

Tabla 9.3: Resumen de propiedades que se pueden establecer para objetos de atributo de hilos POSIX.1c.

Tanto `pthread_attr_init` como `pthread_attr_destroy` llevan un solo argumento que es un apuntador a un objeto de atributo de hilos.

Todas las funciones para obtener/establecer atributos de hilos tienen dos parámetros. El primero es un apuntador a un objeto atributo de hilos; el segundo es el valor del atributo o un apuntador a un valor. Por ejemplo, la sinopsis de las funciones para manipular la política de programación es

SINOPSIS

```
#include <pthread.h>
#include <sched.h>

int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr,
                               struct sched_param *param);
```

POSIX.1c

Un hilo tiene una pila cuya ubicación y tamaño se puede examinar o establecer mediante las llamadas `pthread_attr_getstackaddr` (obtener dirección), `pthread_attr_setstackaddr` (establecer dirección), `pthread_attr_getstacksize` (obtener tamaño) y `pthread_attr_setstacksize` (establecer tamaño). Cuando un hilo se desconecta, ya no se puede esperar con un `pthread_join`. Las funciones `pthread_attr_getdetachstate` y `pthread_attr_setdetachstate` pueden examinar y establecer el `detachstate` (estado de desconexión) de un hilo. Los posibles valores de `detachstate` son `PTHREAD_CREATE_JOINABLE` (unible) o `PTHREAD_CREATE_DETACHED` (desconectado). Por omisión, los hilos son unibles. Los hilos desconectados invocan a `pthread_detach` cuando terminan para liberar sus recursos.

Las funciones `pthread_attr_getscope` y `pthread_attr_setscope` examinan y establecen el atributo `contentionscope` (alcance de contención) que controla si el hilo compite por recursos dentro del proceso o bien en el nivel del sistema. Los posibles valores de `contentionscope` son `PTHREAD_SCOPE_PROCESS` y `PTHREAD_SCOPE_SYSTEM`.

La función `pthread_attr_getinheritsched` examina el atributo `inheritsched` que controla si los parámetros de programación se heredan del hilo creador o se especifican explícitamente. La función `pthread_attr_setinheritsched` establece este atributo. Los posibles valores de `inheritsched` son `PTHREAD_INHERIT_SCHED` (se heredan) y `PTHREAD_EXPLICIT_SCHED` (se especifican explícitamente).

La política de programación de un hilo se almacena en una estructura de tipo `struct sched_param`. El submiembro `sched_policy` de `struct sched_param` contiene la política de programación. Las posibles políticas de programación son "el primero que entra es el primero que sale" (`SCHED_FIFO`), turno circular (`SCHED_RR`) o definido por la implementación (`SCHED_OTHER`). La implementación más común de `SCHED_OTHER` es una política de prioridad apropiativa. Una implementación que se ajuste a POSIX podrá apoyar cualquiera de estas políticas de programación. El comportamiento real de la política en la implementación depende del alcance de programación y de otros factores.

La propiedad con mayores probabilidades de cambiar es la de un hilo, que forma parte de la política de programación. El submiembro `sched_priority` de `struct sched_param` contiene un valor de prioridad `int`. Un valor de prioridad más alto corresponde a una prioridad más alta.

Ejemplo 9.6

El siguiente segmento de código crea un hilo `do_it` *con los atributos por omisión y luego cambia la prioridad a* `HIGHPRIORITY` (prioridad alta).

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
#include <sched.h>

#define HIGHPRIORITY 10

pthread_attr_t my_tattr;
pthread_t my_tid;
struct sched_param param;
int fd;

if (pthread_attr_init(&my_tattr))
   perror("Could not initialize thread attribute object");
else if (pthread_create(&my_tid, &my_tattr, do_it, (void *)&fd))
   perror("Could not create copier thread");
else if (pthread_attr_getschedparam(&my_tattr, &param))
   perror("Could not get scheduling parameters");
else {
   param.sched_priority = HIGHPRIORITY;
   if (pthread_attr_setschedparam(&my_tattr, &param))
      perror("Could not set priority");
}
```

La `pthread_create` del ejemplo 9.6 asocia el objeto de atributo de hilo `my_tattr` al hilo `my_tid` durante su creación a fin de poder modificar posteriormente los atributos asociados a este hilo. Observe que la prioridad del hilo `my_tid` se cambia modificando una propiedad del objeto de atributo de hilo `my_tattr` que se asoció al hilo.

9.6 Ejercicio: Copiado de archivos en paralelo

En esta sección creamos un copiador de archivos en paralelo como una extensión de la aplicación de copiado del programa 9.9. Asegúrese de utilizar llamadas seguras respecto de los hilos en la implementación. El programa principal requiere dos argumentos de línea de comando que son

nombres de directorio y copia todos los archivos del primer directorio al segundo. El programa de copiado preserva la estructura de subdirectorios. Se utilizan los mismos nombres de archivo para el origen y el destino. Implemente el copiado de archivos en paralelo como sigue:

- Escriba una función llamada `copy_directory` que tenga el prototipo

```
void *copy_directory(void *arg)
```

  La función `copy_directory` copia todos los archivos de un directorio a otro. Los nombres de los directorios se pasan en `arg` como dos cadenas consecutivas (separadas por un `NULL`). Suponga que ambos directorios, de origen y de destino, existen en el momento en que se llama a `copy_directory`. En esta versión sólo se copian los archivos ordinarios y se ignoran los subdirectorios. Para cada archivo por copiar, cree un hilo que ejecute la función `copy_file` del programa 9.8. Espere a que el hilo termine su ejecución antes de copiar el siguiente archivo.
- Escriba un programa principal que requiera dos argumentos de línea de comandos para especificar los directorios de origen y de destino. El programa principal crea un hilo para ejecutar `copy_directory` y luego efectúa un `pthread_join` para esperar que el hilo `copy_directory` termine. Utilice el programa principal para probar la primera versión de `copy_directory`.
- Modifique la función `copy_directory` de modo que, si el directorio de destino no existe, lo cree. Pruebe la nueva versión.
- Modifique `copy_directory` de modo que, después de crear un hilo para copiar un archivo, siga creando hilos para crear los demás archivos. Conserve el ID de hilo y los descriptores de archivo abiertos para cada hilo `copy_file` en una lista enlazada con una estructura de nodos similar a

```
typedef struct copy_struct {
     char *namestring;
     int source_fd;
     int destination_fd;
     pthread_t tid;
     struct copy_struct *next_thread;
} copyinfo_t;
copyinfo_t *copy_head = NULL;
copyinfo_t *copy_tail = NULL;
```

  Implemente la lista como un objeto colocando su declaración en un archivo aparte junto con funciones de acceso para insertar, recuperar y eliminar nodos. Una vez que la función `copy_directory` haya creado hilos para copiar todos los archivos del directorio, realizará un `pthread_join` con cada hilo de su lista y liberará la estructura `copyinfo_t`.
- Modifique la función `copy_file` del programa 9.8 de modo que su argumento sea un apuntador a una estructura `copyinfo_t`. Pruebe las nuevas versiones de `copy_file` y `copy_directory`.

- Modifique `copy_directory` de modo que, si un archivo es un directorio en lugar de un archivo ordinario, la función cree un hilo para ejecutar `copy_directory` en vez de `copy_file`. Pruebe la nueva función.
- Invente un método de cronometría para comparar un copiado ordinario con el copiado por hilos.
- Si el programa se ejecuta con un directorio grande, puede tratar de abrir más descriptores de archivo de los que están permitidos para un proceso. Invente un método para manejar esta situación. Algunos intérpretes de comandos (*shells*) permiten al usuario cambiar este límite.
- Determine si cambia el tiempo de ejecución cuando los hilos tienen el alcance `PTHREAD_SCOPE_SYSTEM` en vez de `PTHREAD_SCOPE_PROCESS`.

9.7 Lecturas adicionales

El libro *Distributed Operating Systems* de Tanenbaum [93] trata el tema de los hilos en forma sumamente accesible. Las diferentes estrategias de programación de hilos se analizan en los artículos [3, 10, 27, 48]. Un libro de Kleiman *et al.* [44] que está por aparecer con el título *Programming with Threads* analiza técnicas avanzadas de programación con hilos. Por último, el estándar POSIX.1c [54] es en sí una relación sorprendentemente amena de las consideraciones y elecciones en conflicto que debieron hacerse al implementar un paquete de hilos utilizable.

Capítulo 10

Sincronización de hilos

POSIX.1c maneja un mecanismo de sincronización llamado mutex para el uso de candados a corto plazo y maneja variables de condición para esperar sucesos de duración ilimitada. Los programas multihilos también pueden utilizar los semáforos POSIX.1b para la sincronización. El manejo de señales en programas con hilos presenta complicaciones adicionales que se pueden reducir si los controladores de señales se sustituyen por hilos dedicados. En el presente capítulo ilustraremos estos conceptos de sincronización de hilos a través de variaciones de problema de productores-consumidores.

Los hilos se crean dentro del espacio de direcciones de un proceso y comparten recursos como las variables estáticas y los descriptores de archivo abiertos. Cuando los hilos utilizan recursos compartidos como éstos, deben sincronizar su interacción a fin de obtener resultados consistentes. Hay dos tipos distintos de sincronización: uso de candados y espera. El *uso de candados (locking)* se refiere por lo regular a la tenencia a corto plazo de recursos. La *espera*, que puede tener duración ilimitada, se refiere al bloqueo hasta que ocurre cierto suceso. POSIX.1c provee mutexes y variables de condición para apoyar estos dos tipos de sincronización en programas multihilos; además, permite el empleo de semáforos en este tipo de programas.

En este capítulo desarrollaremos los conceptos de sincronización de hilos en términos del problema de productores-consumidores. Los hilos o procesos productores fabrican datos (por ejemplo, mensajes) y los depositan en una cola FIFO. Los hilos consumidores retiran elementos de datos de la cola. Si la cola tiene un tamaño especificado, se dice que el problema de productores-consumidores es un *problema de buffer acotado*. Un ejemplo de aplicación de productores-consumidores es un administrador de impresoras de red en la que los productores son usuarios que generan solicitudes de impresión y los consumidores son las impresoras. Otros ejemplos incluyen las colas de programación en un sistema multiprocesador o los *buffers* de mensajes de red que se utilizan para rutear mensajes a través de nodos intermedios en una red de área extensa.

La figura 10.1 es un esquema del problema de productores-consumidores. Los hilos productores y consumidores comparten la cola y deben poner un candado a este recurso mientras insertan o sacan elementos. Los productores adquieren el candado sólo cuando tienen un elemento que insertar y conservan el candado sólo mientras están efectuando realmente la inserción. De forma similar, los consumidores ponen un candado a la cola sólo mientras están sacando los elementos, y liberan el candado antes de procesar los elementos que sacaron. Si la cola está vacía, los hilos consumidores deberán bloquearse hasta que haya elementos que sacar. Además, si la cola es de tamaño limitado, los productores deberán esperar hasta que haya espacio disponible antes de producir más datos. El uso de candados por lo regular es de corta duración, pero la espera puede ser de larga duración. He aquí algunos problemas que deben evitarse al programar una aplicación de productores-consumidores:

- Un consumidor saca un elemento mientras un productor lo está colocando en el *buffer* (*candados*).
- Un consumidor saca elementos que no están en el *buffer* (*espera*).
- Un consumidor saca elementos que ya fueron sacados (*candados*).
- Un productor coloca algo en el *buffer* cuando no hay una ranura libre (*espera*).
- Un productor sobreescribe un elemento que no se ha sacado todavía (*candado*).

Un control de flujo de productores-consumidores más complicado podría incluir *marcas de pleamar* y *de bajamar*. Cuando una cola llega a tener cierto tamaño (la marca de pleamar), los productores se bloquean hasta que la cola se vacía hasta la marca de bajamar. Se pueden usar mutexes, variables de condición y semáforos para controlar diversos aspectos del problema. En las siguientes tres secciones ilustraremos el empleo de estas primitivas para controlar y sincronizar los productores y consumidores. En la sección 10.1 presentamos los candados mutex de POSIX.1c y los utilizamos para implementar el acceso exclusivo a la cola de productores-consumidores. En la sección 10.2 usaremos semáforos de POSIX.1b para sincronizar los hilos productores y consumidores cuando se conoce el número de elementos. En la sección 10.3 presentaremos las variables de condición de POSIX.1c y las utilizaremos para implementar la sincronización de productores-consumidores con requisitos de terminación más complejos. En la sección 10.4 se tratará el manejo de señales con hilos. La sección 10.5 presenta una aplicación de un servidor de impresoras con hilos.

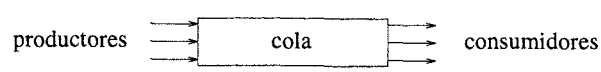


Figura 10.1: Esquema del problema de productores-consumidores.

10.1 Mutex

El *mutex* o *candado mutex (mutex lock)* es el mecanismo de sincronización de hilos más sencillo y más eficiente. Los programas emplean mutex para preservar secciones críticas y obtener acceso exclusivo a los recursos.

SINOPSIS

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *attr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

POSIX.1c

Para utilizar estas funciones, un programa debe declarar una variable de tipo `pthread_mutex_t` e inicializarla antes de utilizarla para sincronización. Por lo regular, las variables de mutex son variables estáticas accesibles para todos los hilos del proceso. Un programa puede inicializar un mutex ya sea invocando `pthread_mutex_init` o empleando el inicializador estático `PTHREAD_MUTEX_INITIALIZER`.

Ejemplo 10.1

El siguiente segmento de código inicializa el mutex `my_lock` *con los atributos por omisión. La variable* `my_lock` *debe estar accesible para todos los hilos que la usan.*

```
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <errno.h>
pthread_mutex_t my_lock;

if (!pthread_mutex_init(&my_lock, NULL))
   perror("Could not initialize my_lock");
```

El método del inicializador estático tiene dos ventajas respecto de `pthread_mutex_init` al inicializar candados mutex: suele ser más eficiente y se garantiza que se ejecutará una y sólo una vez antes de que se inicie la ejecución de cualquier hilo.

Ejemplo 10.2

El siguiente segmento de código inicializa el mutex `my_lock` *con los atributos por omisión empleando el iniciador estático.*

```
#include <pthread.h>
pthread_mutex_t my_lock = PTHREAD_MUTEX_INITIALIZER;
```

Un hilo puede utilizar un `pthread_mutex_lock` (candado mutex) para proteger sus secciones críticas. A fin de preservar la semántica de la sección crítica, el mutex debe ser liberado por el mismo hilo que lo adquirió; es decir, no debemos hacer que un hilo adquiera un candado (`pthread_mutex_lock`) y otro lo libere (`pthread_mutex_unlock`). Una vez que un hilo adquiera un mutex, deberá conservarlo durante un tiempo corto. Los hilos que estén esperando sucesos de duración impredecible deberán utilizar otros mecanismos de sincronización como los semáforos y las variables de condición.

Ejemplo 10.3

El siguiente segmento de código utiliza un mutex para proteger una sección crítica.

```
#include <pthread.h>
pthread_mutex_t my_lock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&my_lock);
    /*  sección crítica */
pthread_mutex_unlock(&my_lock);
```

La adquisición y liberación de candados son voluntarias en el sentido de que sólo se logra la exclusión mutua cuando todos los hilos adquieren correctamente el mutex antes de ingresar en sus secciones críticas. No hay nada que impida a un hilo no cooperativo ingresar en su sección crítica sin adquirir un candado. Una forma de asegurar el acceso exclusivo a objetos es permitir el acceso sólo a través de funciones bien definidas e incluir las llamadas de adquisición de candados en esas funciones. Así, el mecanismo de candados será transparente para los hilos invocadores.

En la figura 10.2 se muestra una implementación de una cola como un *buffer* circular con ocho ranuras (*slots*) y tres elementos de datos. El valor `bufout` indica el número de ranura del siguiente elemento de datos que se va a sacar, y el valor `bufin` indica la siguiente ranura que se llenará. Si los hilos productores y consumidores no actualizan `bufout` y `bufin` de una forma mutuamente exclusiva, un productor podría sobreescribir un elemento que no ha sido sacado o un consumidor podría sacar un elemento que ya se usó.

El programa 10.1 incluye código para implementar un *buffer* circular como un objeto compartido. Las estructuras de datos para el *buffer* se declaran de clase de enlace interno empleando el calificador `static` para limitar su alcance. (En la sección 2.1 se explican los dos significados del calificador `static` en C.) El código está en un archivo aparte para que el programa sólo pueda acceder a `buffer` a través de `get_item` (obtener elemento) y `put_item` (insertar elemento).

Programa 10.1: *Buffer* circular protegido con candados mutex.

```
#include <pthread.h>
#define BUFSIZE 8
static int buffer[BUFSIZE];
static int bufin = 0;
static int bufout = 0;
```

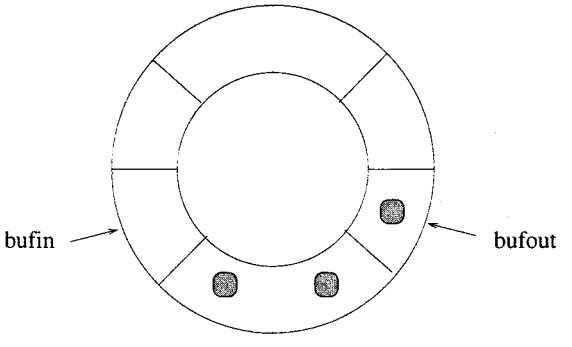


Figura 10.2: Una implementación de *buffer* circular de una cola acotada para el problema de productores-consumidores.

```
static pthread_mutex_t  buffer_lock = PTHREAD_MUTEX_INITIALIZER;

/* Obtener el siguiente elemento del buffer y colocarlo en *itemp. */
void get_item(int *itemp)
{
   pthread_mutex_lock(&buffer_lock);
   *itemp = buffer[bufout];
   bufout = (bufout + 1) % BUFSIZE;
   pthread_mutex_unlock(&buffer_lock);
   return;
}

/* Colocar un elemento en el buffer en la posición bufin y actualizar bufin. */
void put_item(int item)
{
   pthread_mutex_lock(&buffer_lock);
   buffer[bufin] = item;
   bufin = (bufin + 1) % BUFSIZE;
   pthread_mutex_unlock(&buffer_lock);
   return;
}
```

Programa 10.1

El programa 10.2 es un ejemplo sencillo en el que un hilo productor escribe los cuadrados de los enteros de 1 a 100 en el *buffer* circular y un hilo consumidor saca los valores y los suma. Aunque el *buffer* está protegido con candados mutex, la sincronización productor-consumidor no es correcta. El consumidor puede sacar elementos de ranuras vacías y el productor puede sobreescribir ranuras llenas.

Programa 10.2: Implementación incorrecta del problema de productores-consumidores. Los hilos productores y consumidores no están sincronizados.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#define SUMSIZE 100
int sum = 0;

void put_item(int);
void get_item(int *);

void *producer(void * arg1)
{
   int i;
   for (i = 1; i <= SUMSIZE; i++)
      put_item(i*i);
   return NULL;
}

void *consumer(void *arg2)
{
   int  i, myitem;
   for (i = 1; i <= SUMSIZE; i++) {
      get_item(&myitem);
      sum += myitem;
   }
   return NULL;
}

void main(void)
{
   pthread_t prodtid;
   pthread_t constid;
   int i, total;
                                                    /* comprobar valor */
   total = 0;
   for (i = 1; i <= SUMSIZE; i++)
      total += i*i;
   printf("The actual sum should be %d\n", total);

                                                  /* crear hilos */
   if (pthread_create(&constid, NULL, consumer, NULL))
      perror("Could not create consumer");
   else if (pthread_create(&prodtid, NULL, producer, NULL))
      perror("Could not create producer");
```

```
                                    /* esperar a que los hilos terminen */
    pthread_join(prodtid, NULL);
    pthread_join(constid, NULL);
    printf("The threads produced the sum %d\n", sum);
    exit(0);
}
```

Programa 10.2

Ejercicio 10.1

Describa una situación en la que el programa 10.2 produzca una respuesta incorrecta. Suponga que sólo se puede ejecutar un hilo a la vez y que se está usando programación de prioridad apropiativa.

Respuesta:

Suponga que sólo se permite ejecutar un hilo a la vez. El consumidor y el productor se crean con la misma prioridad, así que no hay apropiación para estos hilos limitados por cómputo. El hilo consumidor se crea primero y se ejecuta hasta terminar, consumiendo ceros del *buffer* ya que el productor no ha producido nada. Después el productor entra en acción y produce los valores, pero de nada sirve.

Ejercicio 10.2

En un intento por corregir el problema que presenta el código del programa 10.2, intercambie la creación del consumidor y del productor de modo que el productor comience primero. ¿Qué sucede?

Respuesta:

Esta vez el productor se ejecuta hasta terminar. Puesto que el *buffer* sólo tiene ocho ranuras, los cuadrados del 93 al 100 estarán en el *buffer* cuando el consumidor comience a ejecutarse.

Un hilo o proceso en ejecución que llama a `sched_yield` cede su lugar hasta que vuelve a estar a la cabeza de su lista de programación. En el caso de hilos programados por prioridad apropiativa, un hilo invocador cede su lugar a todos los hilos con mayor prioridad y hasta que todos los hilos con su misma prioridad se han ejecutado y han cedido su lugar, o bien se han bloqueado.

SINOPSIS

```
#include <sched.h>

int sched_yield(void);
```

POSIX.1b

La función `sched_yield` devuelve 0 si tuvo éxito; en caso contrario, devuelve `-1` y establece `errno`.

Las dificultades del programa 10.2 pueden superarse parcialmente obligando a los hilos productores y consumidores a ceder su lugar.

Ejemplo 10.4

La siguiente modificación del programa 10.2 obliga a los hilos productores y consumidores a ceder su lugar en cada iteración de sus respectivos ciclos.

```
#include <pthread.h>
#include <sched.h>
#define SUMSIZE 100
int sum = 0;

void put_item(int);
void get_item(int *);

void *producer(void *arg1)
{
   int i;
   for (i = 1; i <= SUMSIZE; i++)
      put_item(i*i);
      sched_yield();
   return NULL;
}

void *consumer(void *arg2)
{
   int  i, myitem;
   for (i = 1; i <= SUMSIZE; i++) {
      get_item(&myitem);
      sum += myitem;
      sched_yield();
   }
   return NULL;
}
```

Una vez incluido `sched_yield` en el ejemplo 10.4, el productor y el consumidor operan en alternancia estricta siempre que sólo un hilo se ejecute a la vez. Para que el programa funcione a la perfección, tiene que iniciar el hilo productor, y el hilo consumidor debe estar activo en el momento en que el productor ceda su lugar por primera vez. La alternancia estricta no resuelve el problema cuando hay números arbitrarios de productores y consumidores. `sched_yield` forma parte de POSIX.1b, mientras que los hilos forman parte de POSIX.1c. Puede ser necesario buscar la biblioteca que contiene `sched_yield`. En Sun Solaris 2, esta biblioteca se llama `libposix4`. La página del manual deberá indicar la biblioteca.

El comportamiento de los ejemplos de esta sección depende del número exacto de hilos que pueden ejecutarse de forma concurrente y del orden en que inician su ejecución. Un programa multihilo deberá funcionar correctamente, sea cual fuere el orden de ejecución y el nivel de paralelismo. Los elementos y ranuras del problema de productores y consumidores deben sincronizarse de modo que el programa sea independiente del orden de ejecución de los hilos. En la siguiente sección veremos un método tradicional de sincronización empleando semáforos de POSIX.1b.

10.2 Semáforos

La solución tradicional con semáforos para el problema de productores-consumidores utiliza semáforos contadores para representar los recursos (por ejemplo, los semáforos POSIX.1b de la sección 8.3). En el problema de productores-consumidores los recursos son los elementos en la cola y las ranuras libres (en el caso del *buffer* acotado). Cada uno de estos tipos de recurso se representa con un semáforo. Cuando un hilo necesita un recurso de cierto tipo, decrementa el semáforo correspondiente (`sem_wait`). Cuando el hilo libera un recurso, incrementa el semáforo apropiado (`sem_post`). Puesto que la variable de semáforo nunca baja de cero, los hilos no pueden emplear recursos inexistentes. Siempre se debe inicializar un semáforo contador con el número de recursos que están disponibles inicialmente.

El programa 10.3 ilustra una versión correctamente sincronizada del problema de productores-consumidores acotado empleando semáforos POSIX sin nombre. El semáforo `slots`, al cual se asigna como valor inicial `BUFSIZE`, representa el número de ranuras libres disponibles. Este semáforo es decrementado por el productor e incrementado por el consumidor. El semáforo `items`, al cual se asigna el valor inicial 0, representa el número de elementos que están en el *buffer*. Este semáforo es decrementado por el consumidor e incrementado por el productor.

Programa 10.3: Programa de productores-consumidores para hilos sincronizados mediante semáforos.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>

#define SUMSIZE 100
#define BUFSIZE 8

int sum = 0;
sem_t items;
sem_t slots;

void put_item(int);
void get_item(int *);

static void *producer(void *arg1)
{
   int i;

   for (i = 1; i <= SUMSIZE; i++) {
      sem_wait(&slots);
      put_item(i*i);
      sem_post(&items);
```

```
    }
    return NULL;
}

static void *consumer(void *arg2)
{
    int i, myitem;

    for (i = 1; i <= SUMSIZE; i++) {
        sem_wait(&items);
        get_item(&myitem);
        sem_post(&slots);
        sum += myitem;
    }
    return NULL;
}

void main(void)
{
    pthread_t prodtid;
    pthread_t constid;
    int i, total;
                                              /* comprobar valor */
    total = 0;
    for (i = 1; i <= SUMSIZE; i++)
        total += i*i;
    printf("The checksum is %d\n", total);

                              /* inicializar los semáforos */
    sem_init(&items, 0, 0);
    sem_init(&slots, 0, BUFSIZE);

                                          /* crear hilos */
    pthread_create(&prodtid, NULL, producer, NULL);
    pthread_create(&constid, NULL, consumer, NULL);

                         /* esperar a que los hilos terminen */
    pthread_join(prodtid, NULL);
    pthread_join(constid, NULL);
    printf("The threads produced the sum %d\n", sum);
}
```

Programa 10.3

Ejercicio 10.3

¿Qué sucede cuando el programa 10.3 se ejecuta en una máquina con un solo procesador, `BUFSIZE` es 8 y el productor comienza primero? ¿En qué orden se procesan los elementos si la programación de prioridad es apropiativa?

Respuesta:

La respuesta depende del nivel de concurrencia permitido. Suponga que sólo puede ejecutarse un hilo a la vez. El productor produce ocho elementos y luego se bloquea, permitiendo al consumidor entrar en acción por primera vez. El consumidor obtiene los primeros ocho elementos; luego el productor produce los siguientes ocho elementos, y así sucesivamente. Esta alternancia de bloqueos es una consecuencia de la programación de prioridad apropiativa. Un hilo no cede el control a menos que se vea obligado a hacerlo. Si se permite que dos hilos se ejecuten al mismo tiempo, el orden de ejecución dependerá del mecanismo de programación subyacente del núcleo.

Ejercicio 10.4

¿Funcionaría correctamente el programa 10.3 si se crearan dos hilos consumidores?

Respuesta:

No. Si hay más de un hilo consumidor, la modificación del `sum` es una sección crítica que se debe proteger. Además, ambos consumidores intentan procesar `SUMSIZE` elementos y tarde o temprano se bloquearán puesto que el productor termina después de producir un total de `SUMSIZE` elementos.

Los semáforos resuelven el problema de productores-consumidores cuando los productores y los consumidores entran en ciclos infinitos o se ejecutan en un ciclo cierto número de veces. Las cosas no son tan sencillas cuando los productores o consumidores están bajo el control de condiciones de terminación más complicadas. En una variación *controlada por el productor* del problema de productores consumidores hay un solo productor y un número arbitrario de hilos consumidores. El productor coloca cierto número de elementos en la cola y luego termina. Los consumidores continúan hasta que han consumido todos los elementos y el productor ha terminado. Una posible estrategia consiste en que el productor establezca una bandera para indicar que ya completó su operación. El programa 10.4 ilustra algunas de las dificultades del empleo de semáforos para manejar esta situación.

El programa 10.4 ilustra una solución incorrecta a las condiciones de terminación controladas por el productor en el problema de productores-consumidores. El productor inserta `SUMSIZE` elementos en la cola, pero los consumidores no terminan antes de que lo haga el productor. Suponga que el *buffer* está vacío y los consumidores están esperando el semáforo `items`. Si en ese momento el productor decide que ya terminó y establece la bandera `producer_done`, los consumidores se quedarán esperando indefinidamente que se produzca un elemento final.

Programa 10.4: Una solución incorrecta al problema de productores-consumidores controlado por el productor.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <pthread.h>
#include <semaphore.h>
```

```
#define BUFSIZE 8
#define SUMSIZE 100

int producer_done = 0;
int sum = 0;
sem_t items;
sem_t slots;
pthread_mutex_t my_lock = PTHREAD_MUTEX_INITIALIZER;

void put_item(int);
void get_item(int *);

void *producer(void *arg1)
{
   int i;

   for (i = 1; i <= SUMSIZE; i++) {
      sem_wait(&slots);
      put_item(i*i);
      sem_post(&items);
   }
   pthread_mutex_lock(&my_lock);
      producer_done = 1;
   pthread_mutex_unlock(&my_lock);
   return NULL;
}

void *consumer(void *arg2)
{
   int myitem;

   for ( ; ; ) {
      pthread_mutex_lock(&my_lock);
      if (producer_done) {
         pthread_mutex_unlock(&my_lock);
         if (sem_trywait(&items)) break;
      } else {
         pthread_mutex_unlock(&my_lock);
         sem_wait(&items);
      }
      get_item(&myitem);
      sem_post(&slots);
      sum += myitem;
   }
   return NULL;
}
```

Programa 10.4

Un problema que presenta la implementación con semáforos es que una vez que un consumidor se pone a esperar que un semáforo cambie, no hay forma de desbloquear ese consumidor como no sea incrementando el semáforo con `sem_post`. Un productor que ya terminó no puede ejecutar `sem_post` sin hacer que el consumidor crea que hay un elemento disponible. El productor podría probar con un `sem_destroy` (destruir semáforo), pero desafortunadamente POSIX no garantiza que los hilos en espera de que un semáforo cambie se desbloquearán cuando el semáforo se destruya.

Ejercicio 10.5

El programa 10.5 muestra un intento de corregir el problema que acabamos de describir. ¿En qué error incurre esta solución?

Respuesta:

Si el productor establece `producer_done` antes de que el consumidor haya procesado todos los elementos restantes, el consumidor se saldrá del ciclo sin consumir los últimos elementos.

Programa 10.5: Una segunda solución incorrecta al problema de productores-consumidores controlado por el productor.

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <pthread.h>
#include "psemaphore.h"

#define BUFSIZE 8
#define MAXCONSUMERS 1
#define SUMSIZE 100
int producer_done = 0;
int sum = 0;
sem_t items;
sem_t slots;
pthread_mutex_t my_lock = PTHREAD_MUTEX_INITIALIZER;

void put_item(int);
void get_item(int *);

void *producer(void *arg1)
{
   int i;

   for (i = 1; i <= SUMSIZE; i++) {
      sem_wait(&slots);
      put_item(i*i);
      sem_post(&items);
   }
   pthread_mutex_lock(&my_lock);
   producer_done = 1;
```

```
    for (i = 0; i < MAXCONSUMERS; i++)
        sem_post(&items);
    pthread_mutex_unlock(&my_lock);
    return NULL;
}

void *consumer(void *arg2)
{
    int myitem;

    for ( ; ; ) {
        sem_wait(&items);
        pthread_mutex_lock(&my_lock);
        if (!producer_done) {
            pthread_mutex_unlock(&my_lock);
            get_item(&myitem);
            sem_post(&slots);
            sum += myitem;
        } else {
            pthread_mutex_unlock(&my_lock);
            break;
        }
    }
    return NULL;
}
```

Programa 10.5

Nota: En el momento de imprimirse la versión en inglés de este libro no estaba disponible todavía una implementación de semáforos POSIX.1b. Los ejemplos de esta sección se probaron con una implementación de las funciones especificadas en las secciones 8.6 y 8.7. En la siguiente sección presentaremos una solución al problema del apagado (*shutdown*) en términos de variables de condición.

10.3 Variables de condición

La operación `sem_wait` en una variable de semáforo, `S`, espera atómicamente que se cumpla el predicado `S > 0`. Una variable de condición espera que se cumpla un predicado arbitrario y es un mecanismo conveniente para bloquearse hasta que ocurra cierta combinación de sucesos.

Las variables de condición tienen operaciones atómicas para esperar y señalizar (`cond_wait` y `cond_signal`) que son análogas, pero no idénticas, a las operaciones de semáforos `sem_wait` y `sem_post`. En el caso de los semáforos, la prueba del predicado `S > 0` forma parte de `sem_wait`, y el hilo se bloquea sólo si el predicado es falso. El punto clave es que la prueba y el bloqueo se efectúan atómicamente.

Lea los dos párrafos siguientes con mucho detenimiento. Las variables de condición son difíciles de entender a la primera. La información de estos dos párrafos se repetirá varias

veces en esta sección. Una vez que se comprenden, las variables de condición no son difíciles de usar.

Suponga que un hilo necesita esperar hasta que se cumpla cierto predicado en el que interviene un conjunto de variables compartidas (por ejemplo, que dos de esas variables sean iguales). Estas variables compartidas se protegerán con un candado mutex, pues cualquier segmento de código que las utilice forma parte de una sección crítica. Una variable de condición adicional ofrece un mecanismo para que los hilos esperen el cumplimiento de predicados en los que intervienen esas variables. Cada vez que un hilo modifica una de estas variables compartidas, señaliza mediante la variable de condición que se ha realizado un cambio. Esta señal activa un hilo en espera, que entonces determina si ahora su predicado se cumple.

Cuando se envía una señal a un hilo en espera, éste debe cerrar el mutex antes de probar su predicado. Si el predicado es falso, el hilo deberá liberar el candado y bloquearse de nuevo. El mutex debe liberarse antes de que el hilo se bloquee para que otro hilo pueda tener acceso al mutex y modificar las variables protegidas. La liberación de mutex y el bloqueo deben ser atómicos para evitar que otro hilo modifique las variables entre estas dos operaciones. Puesto que la señal sólo indica que las variables pueden haber cambiado, no que se ha cumplido el predicado y el hilo bloqueado deberá volver a probar el predicado cada vez que reciba una señal.

Siga estos pasos al utilizar una variable de condición para sincronizar con base en un predicado arbitrario:

a) Adquiera el mutex.

b) Pruebe el predicado.

c) Si el predicado se cumple, realice trabajo y libere el mutex.

d) Si el predicado no se cumple, llame a `cond_wait` y pase a b) cuando regrese.

La operación `cond_wait` bloquea atómicamente el hilo en espera y libera el mutex. Así, el mutex se libera explícitamente si el predicado se cumple e implícitamente si no se cumple. Cuando un hilo que espera una variable de condición se desbloquea, vuelve a adquirir el mutex automáticamente como parte del proceso de desbloqueo. Si es necesario, deja de esperar la variable de condición y se pone a esperar el mutex. Siempre debe utilizarse el mismo mutex con una variable de condición en particular.

Ejemplo 10.5

El siguiente pseudocódigo ilustra el empleo de una variable de condición, `v`*, y su candado mutex asociado,* `m`*, para obligar a un hilo a esperar hasta que las variables* `x` *y* `y` *tienen el mismo valor.*

```
a:| lock_mutex(&m);
b:|   while (x != y)
c:|      cond_wait(&v, &m);
d:|   /* do stuff related to x and y */
e:| unlock_mutex(&m);
```

En el ejemplo 10.5 el predicado o condición que debe cumplirse para que el hilo continúe es `x == y`. (Observe que esta es una negación de la prueba real que aparece en el ciclo `while`.) El hilo cierra el mutex, `m`, antes de probar `x != y`. El hilo que ejecuta este segmento de código no ejecuta el enunciado *d* hasta que `x` sea igual a `y`. La prueba se realiza en un ciclo `while` y no con un solo enunciado `if` para asegurarse de que el predicado deseado realmente se cumpla.

Cuando un hilo que espera el cambio de un semáforo simple `S` se activa, está garantizado que `S > 0`, a menos que la espera haya sido interrumpida por una señal. Las variables de condición no están asociadas a predicados específicos, de modo que un programa no sabe si el predicado se cumple; sólo sabe que algún hilo señalizó mediante esa variable de condición.

Ejemplo 10.6

El siguiente pseudocódigo ilustra la señalización relacionada con las variables de condición.

```
f:| lock_mutex(&m);
g:|    x++;
h:|    cond_signal(&v);
i:| unlock_mutex(&m);
```

El hilo del ejemplo 10.6 modifica el valor de `x` y luego señaliza mediante `v` para que un hilo en espera pueda probar el predicado. El sangrado en los segmentos de pseudocódigo de los ejemplos 10.5 y 10.6 indica cuáles enunciados se ejecutan mientras se tiene el candado mutex. El mecanismo de bloqueo real es un tanto complicado. Si el hilo sigue teniendo el candado cuando se bloquea en la `cond_wait` del enunciado *c*, otros procesos no podrán adquirir el candado mutex en el enunciado *f* para señalizar mediante la variable de condición y desbloquear el hilo. Se presentaría un bloqueo mortal. Para resolver este problema, la implementación garantiza que si el hilo se bloquea en una variable de condición, `cond_wait` liberará atómicamente el mutex y se bloqueará. Puesto que estas operaciones son atómicas, ningún otro hilo podrá modificar una variable compartida y señalizar antes de que este hilo se bloquee. Cuando el hilo se desbloquea, readquiere el candado mutex antes de ejecutar cualquier otro enunciado. El segundo argumento de `cond_wait` le indica al hilo cuál candado mutex es el que debe adquirir antes de continuar.

Ejercicio 10.6

Suponga que el hilo 1 ejecuta el código del ejemplo 10.5 y el hilo 2 ejecuta el código del ejemplo 10.6. Los valores iniciales de `x` y `y` son 0 y 2, respectivamente. ¿Qué sucede cuando la ejecución de los dos hilos se intercala como sigue: a_1, b_1, c_1, f_2, g_2, h_2, i_2. ¿Cuál enunciado ejecuta en seguida el hilo 1?

Respuesta:

En esta intercalación el hilo 1 adquiere el candado mutex `m` y prueba `x != y`. Puesto que `x` es 0 y `y` es 2, la prueba tiene éxito y el hilo 1 se bloquea en la variable de condición `v` y libera el candado `m`. A continuación, el hilo 2 adquiere el candado, incrementa la variable `x`, envía una señal a la variable de condición `v` y libera el mutex `m`. La señal desbloquea el hilo 1, el cual readquiere el candado mutex que se libera en el enunciado i_2. El hilo 1 vuelve a probar el predicado. Puesto que todavía se cumple `x != y`, el hilo 1 ejecuta `cond_wait` y se bloquea otra vez. Los siguientes enunciados que el hilo 1 ejecuta son b_1 y c_1. Estos enunciados pueden intercalarse con enunciados de otros hilos.

Ejercicio 10.7

¿Son posibles las siguientes intercalaciones del pseudocódigo de los ejemplos 10.5 y 10.6?

$$a_1, b_1, c_1, f_2, g_2, h_2, b_1, c_1, i_2$$
$$a_1, b_1, f_2, g_2, c_1, h_2$$

Respuesta:

Ninguna de estas intercalaciones es posible. En la primera, el hilo 1 se activa después de una `cond_wait` y debe readquirir el candado mutex antes de continuar. El hilo 2 tiene el candado cuando emite su señal. Por tanto, i_2 debe ejecutarse antes que el segundo b_1 de la intercalación. La segunda intercalación no es posible porque el hilo 1 no libera el mutex antes del `cond_wait` del enunciado c_1. Por tanto, el hilo 2 no puede adquirir el candado en el enunciado f_2 antes de que sea liberado en c_1.

La `cond_signal` no garantiza que el predicado ya se cumple; simplemente es una forma de indicarle al proceso bloqueado que las variables que intervienen en el predicado pueden haber cambiado y que el hilo debe volver a probar el predicado. El ciclo `while` del enunciado *b* del ejemplo 10.5 es necesario para asegurar que el predicado realmente se cumpla. Si la prueba del ciclo `while` tiene éxito (el predicado es falso), el hilo se bloquea otra vez realizando una `cond_wait`. Cuando otro hilo modifica las variables del predicado, puede activar el hilo bloqueado invocando `cond_signal` otra vez.

10.3.1 Variables de condición para hilos POSIX.1c

Los hilos POSIX.1c proveen sincronización mediante variables de condición de forma similar a la descrita al principio de esta sección. Una variable de condición se inicializa ya sea empleando un iniciador estático o llamando `apthread_cond_init`. Si `attr` es `NULL`, `pthread_cond_init` utiliza los atributos de variable de condición por omisión.

SINOPSIS

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex,
                      const struct timespec *abstime);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

POSIX.1c

Las variables de condición se llaman así por el hecho de que siempre se usan con una condición, es decir, un predicado. Un hilo prueba un predicado y llama a `pthread_cond_wait` si el predicado es falso. Cuando otro hilo modifica variables que podrían hacer que se cumpliera el predicado, activa al hilo bloqueado ejecutando `pthread_cond_signal`. Otras acciones, como la recepción de una señal UNIX, también pueden hacer que el hilo bloqueado regrese de `pthread_cond_wait`. El hilo que estaba bloqueado normalmente vuelve a probar el predicado y llama a `pthread_cond_wait` otra vez si el predicado sigue siendo falso.

A fin de garantizar que la prueba del predicado y la espera sean atómicas, el hilo invocador debe obtener un mutex antes de probar el predicado. La implementación garantiza que si el hilo se bloquea en una variable de condición, `pthread_cond_wait` liberará atómicamente el mutex y se bloqueará. Otro hilo no podrá emitir una señal antes de que este hilo se bloquee.

Ejemplo 10.7

Sea `v` *una variable de condición y* `m` *un candado mutex. El siguiente es un ejemplo del uso correcto de la variable de condición para acceder a un recurso si se cumple el predicado definido por* `test_condition()`.

```
#include <pthread.h>
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t v = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&m);
while (!test_condition())
    pthread_cond_wait(&v, &m);
       /* obtener recurso (hacer que test_condition devuelva false) */
pthread_mutex_unlock(&m);
       /* haga lo que quiera */
pthread_mutex_lock(&m);
       /* liberar recurso (hacer que test_condition devuelva true) */
pthread_cond_signal(&v);
pthread_mutex_unlock(&m);
```

En el ejemplo 10.7 el hilo debe adquirir el mutex `m` especificado en la función `pthread_cond_wait` antes de invocar a `pthread_cond_wait`. Si `test_condition` (probar condición) devuelve false, el hilo ejecutará `pthread_cond_wait`, liberará el mutex `m` y se bloqueará en la variable de condición `v`.

Cuando un hilo ejecuta la `pthread_cond_wait` en el ejemplo 10.7, está en posesión del mutex `m`. El hilo se bloquea atómicamente y libera el mutex, permitiendo que otro hilo adquiera el mutex y modifique las variables del predicado. Cuando un hilo regresa con éxito de una `pthread_cond_wait`, ha adquirido el mutex y puede volver a probar el predicado sin volver a adquirir explícitamente el mutex. Incluso si el programa señaliza en una variable de condición específica sólo cuando cierto predicado se cumple, los hilos en espera tendrán que probar de todos modos el predicado.

La función `pthread_cond_wait` activa un hilo que está esperando una variable de condición. Si hay más hilos esperando, se escoge uno siguiendo un criterio congruente con el algoritmo de programación. La función `pthread_cond_broadcast` activa todos los hilos que esperan una variable de condición. Estos hilos activados entran en contención por el candado mutex antes de regresar de `pthread_cond_wait`.

He aquí algunas reglas para usar variables de condición:

- Adquiera el mutex antes de probar el predicado.
- Vuelva a probar el predicado después de regresar de una `pthread_cond_wait`, pues el retorno podría deberse a algún suceso no relacionado o a una `pthread_cond_signal` que no implica que el predicado ya se cumple.
- Adquiera el mutex antes de modificar alguna de las variables que aparecen en el predicado.
- Adquiera el mutex antes de llamar a las funciones `pthread_cond_signal` o `pthread_cond_broadcast`.
- Retenga el mutex durante un periodo corto, por lo regular sólo mientras prueba el predicado. Libere el mutex lo antes posible, ya sea de forma explícita (con `pthread_mutex_unlock`) o implícita (con `pthread_cond_wait`).

El programa 10.6 muestra una solución con variables de condición al problema de *buffer* acotado controlado por el productor. El productor termina después de producir una cantidad fija de elementos. El consumidor continúa hasta que procesa todos los elementos y detecta que el productor ya terminó.

Programa 10.6: Una solución con variables de condición al problema de *buffer* acotado controlado por el productor.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define SUMSIZE 100
```

```
#define BUFSIZE 8

int sum = 0;
pthread_cond_t slots = PTHREAD_COND_INITIALIZER;
pthread_cond_t items = PTHREAD_COND_INITIALIZER;
pthread_mutex_t slot_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t item_lock = PTHREAD_MUTEX_INITIALIZER;
int nslots = BUFSIZE;
int producer_done = 0;
int nitems = 0;

void get_item(int *itemp);
void put_item(int item);

void *producer(void * arg1)
{
   int i;

   for (i = 1; i <= SUMSIZE; i++) {
      pthread_mutex_lock(&slot_lock);     /* adquirir derecho a una ranura */
         while (nslots <= 0)
            pthread_cond_wait (&slots, &slot_lock);
         nslots--;
      pthread_mutex_unlock(&slot_lock);

      put_item(i*i);
      pthread_mutex_lock(&item_lock);    /* renunciar derecho a un elemento */
         nitems++;
         pthread_cond_signal(&items);
      pthread_mutex_unlock(&item_lock);
   }
   pthread_mutex_lock(&item_lock);
      producer_done = 1;
      pthread_cond_broadcast(&items);
   pthread_mutex_unlock(&item_lock);
   return NULL;
}

void *consumer(void *arg2)
{
   int myitem;

   for ( ; ; ) {
      pthread_mutex_lock(&item_lock);    /* adquirir derecho a un elemento */
         while ((nitems <=0) && !producer_done)
            pthread_cond_wait(&items, &item_lock);
         if ((nitems <= 0) && producer_done) {
            pthread_mutex_unlock(&item_lock);
            break;
```

```
            }
            nitems--;
        pthread_mutex_unlock(&item_lock);
        get_item(&myitem);
        sum += myitem;
        pthread_mutex_lock(&slot_lock);    /* renunciar derecho a una ranura */
            nslots++;
            cond_signal(&slots);
        pthread_mutex_unlock(&slot_lock);
    }
    return NULL;
}

void main(void)
{
    pthread_t prodtid;
    pthread_t constid;
    int i, total;
                                                    /* comprobar valor */
    total = 0;
    for (i = 1; i <= SUMSIZE; i++)
        total += i*i;
    printf("The checksum is %d\n", total);
                                                  /* crear hilos */
    pthread_create(&prodtid, NULL, producer, NULL);
    pthread_create(&constid, NULL, consumer, NULL);
                              /* esperar a que los hilos terminen */
    pthread_join(prodtid, NULL);
    pthread_join(constid, NULL);
    printf("The threads produced the sum %d\n", sum);
    exit(0);
}
```

Programa 10.6

El productor del programa 10.6 se bloquea hasta que se cumple el predicado `nslots > 0`. Su prueba puede escribirse `while(!(nslots > 0))`, o bien `while(nslots <= 0)`. El consumidor se bloquea hasta que se cumple el predicado `(nitems > 0) || producer_done` (es decir, el consumidor debe actuar si hay un elemento disponible o si el productor ya terminó). Por tanto, la prueba aquí es `while((nitems <= 0) && !producer_done)`. El consumidor terminará sólo si el productor ya acabó y no quedan más elementos.

10.4 Manejo de señales e hilos

La interacción de los hilos con las señales implica varias complicaciones. Todos los hilos comparten los controladores de señales del proceso, pero cada hilo puede tener su propia

máscara de señal. Además, los distintos tipos de señales se manejan de diferente manera. En la tabla 10.1 se resumen los tipos de señales y sus métodos de manejo.

Tipo	Acción de entrega
Asíncrona	Se entrega a un hilo que la tiene desbloqueada.
Síncrona	Se entrega al hilo que la causó.
Dirigida	Se entrega al hilo identificado (`pthread_kill`).

Tabla 10.1: Entrega de señales en hilos.

Las señales como `SIGFPE` (excepción de punto flotante) son síncronas con el hilo que las causó (esto es, siempre se generan en el mismo punto de la ejecución del hilo). Las señales síncronas también suelen llamarse trampas (traps). Éstas son manejadas por el hilo que las causó. Otras señales son asíncronas en cuanto a que no se generan en un momento predecible y no están asociadas a un hilo en particular. Si varios hilos tienen desbloqueada una señal asíncrona, se escoge uno de ellos para manejar la señal. Las señales también pueden dirigirse a un hilo en particular con `pthread_kill`. Un hilo puede examinar o establecer su máscara de señal con la función `pthread_sigmask`.

SINOPSIS

```
#include <signal.h>
#include <pthread.h>

int pthread_kill(pthread_t thread, int sig);
```

POSIX.1c

SINOPSIS

```
#include <pthread.h>
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *set,
                    sigset_t *oset);
```

POSIX.1c

La llamada `pthread_sigmask` es similar a `sigprocmask`. El parámetro `how` (cómo) puede ser `SIG_BLOCK` (bloquear señal), `SIG_UNBLOCK` (desbloquear señal) o `SIG_SETMASK` (establecer máscara de señal). Los controladores de señales abarcan todo el proceso y se instalan con llamadas a `sigaction` igual que en los procesos de un solo hilo. La distinción entre los controladores de señales que abarcan todo el proceso y los que son específicos para un hilo es importante.

Recuérdese que cuando hablamos de las señales dijimos que cuando se ingresa en el controlador de señales la señal que causó el suceso se bloquea automáticamente. En una

aplicación multihilo no hay nada que impida que otra señal del mismo tipo se entregue a otro hilo que tenga desbloqueada la señal. Es posible tener a varios hilos ejecutándose dentro del mismo controlador de señales. Es indispensable que la llamada a `sigaction` bloquee explícitamente la señal que se está manejando. La instalación del controlador de señales para bloquear la señal hace que todos los hilos tengan la señal bloqueada cuando el controlador de señales está activo.

Una estrategia alternativa para manejar las señales en procesos multihilo es dedicar hilos específicos al manejo de señales. El hilo principal bloquea todas las señales antes de crear los hilos. La máscara de señal se hereda del hilo creador, de modo que todos los hilos tienen la señal bloqueada. Entonces, el hilo dedicado a manejar la señal ejecuta una `sigwait` con esa señal (véase la página 391).

A fin de ilustrar los dos enfoques (controlador de señales *vs.* hilo dedicado), consideremos una variación del problema de *buffer* acotado en la que el productor inserta elementos en el *buffer* hasta que el programa recibe una señal `SIGUSR1`. En ese momento, el productor acaba el elemento que estaba produciendo y sale. El programa no puede simplemente eliminar el hilo productor porque podría estar en posesión de un candado mutex, el cual entonces quedaría en un estado indeterminado.

El programa 10.7 ilustra la estrategia de controlador de señales para resolver el problema de *buffer* acotado controlado por señales. El controlador establece la variable `producer_shutdown` para indicar que el hilo productor debe terminar. Como en el programa 10.7 el controlador de señales y el hilo productor comparten la variable `producer_shutdown`, ésta debe protegerse en el controlador con el mismo mutex que en el hilo productor. Si llega una señal `SIGUSR1` mientras el productor tiene este candado, el controlador de señales entra en un bloqueo mortal. A fin de evitar esto, el productor bloquea la señal cuando tiene el mutex. En esta solución se supone que todos los demás hilos también bloquearon a `SIGUSR1`. Dado que el productor bloquea la señal mientras está esperando la condición, no detecta que debe terminar hasta que hay una ranura (*slot*) disponible y produce el siguiente elemento.

Las funciones del programa 10.7 se instrumentaron de modo que pueden probarse. El programa principal exhibe su ID de proceso y luego duerme durante cinco segundos antes de crear los hilos. Esto permite al usuario pasar a otra ventana y prepararse para ejecutar un comando que envíe la señal `SIGUSR1` al proceso. Se exhiben los ID de los hilos productor y consumidor para indicar cuándo se inician los hilos. En la mayor parte de las máquinas, el cálculo de la suma de los cuadrados en este sencillo programa causará un desbordamiento de un entero de 32 bits en muy poco tiempo (después de 1861 elementos), por lo que se ha añadido un pequeño retardo en varios lugares para hacerlo más lento. La duración del retardo se controla con la constante `SPIN` y se debe modificar para las diferentes máquinas. Aparece una advertencia cuando el programa termina porque se detecta un desbordamiento. El consumidor del programa 10.7 es casi idéntico al del programa 10.6 excepto por el retardo. El programa principal bloquea la señal `SIGUSR1` antes de crear los hilos para que éstos hereden una máscara de señal con esa señal bloqueada. El productor necesita tener la señal bloqueada mientras está en posesión del mutex `slot_lock`. El hilo productor se vale de `pthread_sigmask` para desbloquear esta señal a fin de que esté desbloqueada sólo para ese hilo.

Programa 10.7: Estrategia de controlador de señales para resolver el problema del *buffer* acotado controlado por señales.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>

#define SUMSIZE 1861
#define BUFSIZE 8
#define SPIN 10000

int sum = 0;
pthread_cond_t slots = PTHREAD_COND_INITIALIZER;
pthread_cond_t items = PTHREAD_COND_INITIALIZER;
pthread_mutex_t slot_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t item_lock = PTHREAD_MUTEX_INITIALIZER;
int nslots = BUFSIZE;
int producer_done = 0;
int nitems = 0;
int totalproduced = 0;
int producer_shutdown = 0;

void get_item(int *itemp);
void put_item(int item);

/* spinit entra en un ciclo para hacer tiempo */
void spinit(void)
{
   int i;
   for (i = 0; i < SPIN; i++) ;
}

/* controlador de señales para la desactivación */
void catch_sigusr1(int signo)
{
   pthread_mutex_lock(&slot_lock);
      producer_shutdown = 1;
   pthread_mutex_unlock(&slot_lock);
}

void *producer(void * arg1)
{
   int i;
   sigset_t intmask;

   sigemptyset(&intmask);
   sigaddset(&intmask, SIGUSR1);
```

```
   for (i = 1;  ; i++){
      spinit();
      pthread_sigmask(SIG_BLOCK, &intmask, NULL);
      pthread_mutex_lock(&slot_lock);     /* adquirir el derecho a una ranura */
      spinit();
      while ((nslots <= 0) && (!producer_shutdown))
         pthread_cond_wait (&slots, &slot_lock);
      if (producer_shutdown) {
         pthread_mutex_unlock(&slot_lock);
         break;
      }
      nslots--;
      pthread_mutex_unlock(&slot_lock);
      pthread_sigmask(SIG_UNBLOCK, &intmask, NULL);

      spinit();
      put_item(i*i);
      pthread_mutex_lock(&item_lock);    /* renunciar al derecho a un elemento */
         nitems++;
         totalproduced++;
         pthread_cond_signal(&items);
      pthread_mutex_unlock(&item_lock);
      spinit();
   }
   pthread_mutex_lock(&item_lock);
      producer_done = 1;
      pthread_cond_broadcast(&items);
   pthread_mutex_unlock(&item_lock);
   return NULL;
}

void *consumer(void *arg2)
{
   int myitem;

   for ( ; ; ) {
      pthread_mutex_lock(&item_lock);    /* adquirir el derecho a un elemento */
         while ((nitems <= 0) && !producer_done)
            cond_wait(&items, &item_lock);
         if ((nitems <= 0) && producer_done) {
            pthread_mutex_unlock(&item_lock);
            break;
         }
         nitems--;
      pthread_mutex_unlock(&item_lock);
      spinit();
      get_item(&myitem);
      spinit();
      sum += myitem;
```

```
        pthread_mutex_lock(&slot_lock);  /* renunciar al derecho a una ranura */
            nslots++;
            cond_signal(&slots);
        pthread_mutex_unlock(&slot_lock);
    }
    return NULL;
}

void main(void)
{
    pthread_t prodtid;
    pthread_t cons id;
    double total;
    double tp;
    struct sigaction act;
    sigset_t set;

    fprintf(stderr, "Process ID is %ld\n", (long)getpid());
    sleep(5);
                                    /* instalar controlador de señales y bloquear */
    act.sa_handler = catch_sigusr1;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGUSR1, &act, NULL);
    sigemptyset(&set);
    sigaddset(&set,SIGUSR1);
    sigprocmask(SIG_BLOCK, &set, NULL);
                                                        /* crear hilos */
    pthread_create(&prodtid, NULL, producer, NULL);
    pthread_create(&constid, NULL, consumer, NULL);
    fprintf(stderr,"Producer ID = %d, Consumer ID = %d\n",
        (int)prodtid, (int)constid);
                                      /* esperar a que los hilos terminen */
    pthread_join(prodtid, NULL);
    pthread_join(constid, NULL);
    printf("The threads produced the sum   %d\n", sum);
                                                /* mostrar valor correcto */
    total = 0.0;
    tp = (double) totalproduced;
    total = tp*(tp+1)*(2*tp+1)/6.0;
    if (tp > SUMSIZE)
        fprintf(stderr,"*** Overflow occurs for more than %d items\n",
            SUMSIZE);
    printf("The checksum for %4d items is %1.0f\n",
        totalproduced, total);
    exit(0);
}
```

Programa 10.7

Las funciones `pthread_mutex_lock` y `pthread_mutex_unlock` son *seguras respecto de señales asíncronas*, lo que significa que se pueden invocar desde un controlador de señales. No obstante, todos los hilos deben bloquear la señal antes de adquirir un candado mutex de este tipo, pues de lo contrario el programa podría entrar en un bloqueo mortal.

Un enfoque alternativo consiste en usar un hilo dedicado para manejar las señales. Todos los hilos excepto el dedicado bloquean la señal en cuestión. El hilo dedicado realiza una `sigwait` con la señal especificada.

SINOPSIS

```
#include <signal.h>

int sigwait(sigset_t *sigmask, int *signo);
```

POSIX.1c

La función `sigwait` se bloquea hasta que el hilo recibe cualquiera de las señales especificadas por `*sigmask`. El valor `*signo` es el número de la señal que causó el retorno desde `sigwait`. La función `sigwait` devuelve 0 si la llamada tuvo éxito y `-1` en caso contrario; en caso de error, establece `errno`.

Tome nota de las diferencias entre `sigwait` y `sigsuspend`. Ambas funciones tienen un primer parámetro que es un apuntador a un conjunto de señales (`sigset_t *`). En el caso de `sigsuspend`, este conjunto contiene la nueva máscara de señal, de modo que las señales que *no están* en el conjunto son las que pueden hacer que `sigsuspend` regrese. En el caso de `sigwait`, este parámetro contiene el conjunto de señales que deben esperarse, de modo que las señales que *están* en el conjunto son las que pueden hacer que `sigwait` regrese. En ambos casos, el programa bloquea las señales de interés antes de la llamada. Con `sigsuspend`, la señal se entrega al proceso y `sigususpend` sólo regresa después de un retorno normal del controlador de señales. Con `sigwait`, la señal pendiente se quita sin entregarse, por lo que no es necesario el controlador de señales.

El programa 10.8 muestra una solución al problema del *buffer* acotado controlado por señales que utiliza un hilo aparte para esperar la señal `SIGUSR1`. La estrategia del hilo dedicado es más sencilla que la del controlador de señales en varios sentidos. Los hilos no están limitados a invocar funciones seguras respecto de señales asíncronas y se puede informar al hilo productor que debe terminar mientras está esperando que se libere una ranura.

El programa principal del programa 10.8 bloquea la señal `SIGUSR1` antes de crear hilos. Puesto que los hilos creados heredan la máscara de señal, todos los hilos tendrán `SIGUSR1` bloqueada. El programa principal crea un hilo dedicado para manejar la señal. El programa 10.8 se probó en un sistema que sólo maneja programación de prioridad apropiativa. El hilo que espera la señal, `sigusr1_thread`, tiene prioridad superior a la prioridad por omisión a fin de garantizar que `sigusr1_thread` pueda atrapar la señal. Si se ejecuta el programa 10.8 en el mismo sistema sin incrementar primero la prioridad de `sigusr1_thread`, el programa funciona correctamente, pero en algunas ocasiones transcurren varios segundos entre el momento en que se genera la señal y el instante en que se atrapa. La prioridad se establece inicializando un atributo de hilo con el valor por omisión mediante `pthread_attr_init`,

colocando la prioridad en una variable, aumentando la prioridad en esa variable, restableciendo la prioridad del atributo y creando por último un hilo con este nuevo atributo. Si está disponible una política de programación de turno circular (*round-robin*), todos los hilos podrían tener la misma prioridad.

El hilo dedicado al manejo de señales, `sigusr1_thread`, exhibe su prioridad para confirmar que ésta se estableció correctamente y luego llama a `sigwait` al recibir la señal `SIGUSR1`. No se requiere controlador de señales, puesto que `sigwait` retira la señal de las que están pendientes. La señal siempre está bloqueada, así que nunca se ingresa en el controlador de `SIGUSR1` por omisión (que terminaría el proceso).

El programa 10.8 se diseñó de modo que pueda probarse enviando la señal `SIGUSR1` al proceso desde otra ventana mediante el comando `kill`. El programa comienza exhibiendo su ID de proceso y luego inicia los cálculos como en el programa 10.7. Sin intervención del usuario, el programa causará un desbordamiento del entero empleado para almacenar la suma después de 1861 elementos en una fracción de segundo si se ejecuta en una máquina rápida. La función `spinit` hace lento el programa en varios puntos. El parámetro `SPIN` determina cuánto tiempo pasará el programa sin hacer nada y debe ajustarse según la velocidad de la máquina objetivo. Una función tipo *sleep* habría hecho más transportable el programa, pero no se utilizó porque podría afectar la programación de los hilos.

Programa 10.8: Solución al programa del *buffer* acotado controlado por señales empleando un hilo dedicado.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <pthread.h>
#include <sched.h>
#define SUMSIZE 1861
#define BUFSIZE 8
#define SPIN 1000000

int sum = 0;
pthread_cond_t slots = PTHREAD_COND_INITIALIZER;
pthread_cond_t items = PTHREAD_COND_INITIALIZER;
pthread_mutex_t slot_lock = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t item_lock = PTHREAD_MUTEX_INITIALIZER;
int nslots = BUFSIZE;
int producer_done = 0;
int nitems = 0;
int totalproduced = 0;
int producer_shutdown = 0;

void get_item(int *itemp);
void put_item(int item);
```

```
void spinit(void)
{
   int i;
   for (i = 0; i < SPIN; i++) ;
}

void *sigusr1_thread(void *arg)
{
   sigset_t intmask;
   struct sched_param param;
   int policy;

   sigemptyset(&intmask);
   sigaddset(&intmask, SIGUSR1);

   pthread_getschedparam(pthread_self(), &policy, &param);
   fprintf(stderr,
      "sigusr1_thread entered with policy %d and priority %d\n",
      policy, param.sched_priority);

   sigwait(&intmask);
   fprintf(stderr, "sigusr1_thread returned from sigwait\n");
   pthread_mutex_lock(&slot_lock);
      producer_shutdown = 1;
      pthread_cond_broadcast(&slots);
   pthread_mutex_unlock(&slot_lock);
   return NULL;
}

void *producer(void *arg1)
{
   int i;

   for (i = 1;   ; i++) {
      spinit();
      pthread_mutex_lock(&slot_lock);    /* adquirir el derecho a una ranura */
         while ((nslots <= 0) && (!producer_shutdown))
            pthread_cond_wait (&slots, &slot_lock);
         if (producer_shutdown) {
            pthread_mutex_unlock(&slot_lock);
            break;
         }
         nslots--;
      pthread_mutex_unlock(&slot_lock);
      spinit();

      put_item(i*i);
      pthread_mutex_lock(&item_lock);   /* renunciar al derecho a un elemento */
         nitems++;
```

```
            pthread_cond_signal(&items);
        pthread_mutex_unlock(&item_lock);
        spinit();
        totalproduced = i;
    }
    pthread_mutex_lock(&item_lock);
        producer_done = 1;
        pthread_cond_broadcast(&items);
    pthread_mutex_unlock(&item_lock);
    return NULL;
}

void *consumer(void *arg2)
{
    int myitem;

    for ( ; ; ) {
        pthread_mutex_lock(&item_lock);    /* adquirir el derecho a un elemento */
            while ((nitems <=0) && !producer_done)
                cond_wait(&items, &item_lock);
            if ((nitems <= 0) && producer_done) {
                pthread_mutex_unlock(&item_lock);
                break;
            }
            nitems--;
        pthread_mutex_unlock(&item_lock);
        get_item(&myitem);
        sum += myitem;
        pthread_mutex_lock(&slot_lock);    /* renunciar al derecho a una ranura */
            nslots++;
            cond_signal(&slots);
        pthread_mutex_unlock(&slot_lock);
    }
    return NULL;
}

void main(void)
{
    pthread_t prodtid;
    pthread_t constid;
    pthread_t sighandid;
    double total;
    double tp;
    sigset_t set;
    pthread_attr_t high_prio_attr;
    struct sched_param param;
    fprintf(stderr, "Process ID is %ld\n", (long)getpid());
                                                    /* bloquear la señal */
    sigemptyset(&set);
```

```
    sigaddset(&set, SIGUSR1);
    sigprocmask(SIG_BLOCK, &set, NULL);
    fprintf(stderr, "Signal blocked\n");
                                                    /* crear hilos */
    pthread_attr_init(&high_prio_attr);
    pthread_attr_getschedparam(&high_prio_attr, &param);
    param.sched_priority++;
    pthread_attr_setschedparam(&high_prio_attr, &param);
    pthread_create(&sighandid, &high_prio_attr, sigusr1_thread, NULL);
    pthread_create(&prodtid, NULL, producer, NULL);
    pthread_create(&constid, NULL, consumer, NULL);
                                    /* esperar a que los hilos terminen */
    pthread_join(prodtid, NULL);
    pthread_join(constid, NULL);
    printf("The threads produced the sum   %d\n", sum);
                                          /* mostrar valor correcto */
    tp = (double) totalproduced;
    total = tp*(tp + 1)*(2*tp + 1)/6.0;
    if (tp > SUMSIZE)
       fprintf(stderr, "*** Overflow occurs for more than %d items\n",
          SUMSIZE);
    printf("The checksum for %4d items is %1.0f\n",
       totalproduced, total);
    exit(0);
}
```

Programa 10.8

10.5 Ejercicio: Servidor de impresión con hilos

En la mayor parte de los sistemas, el comando `lp` no envía un archivo directamente a la impresora especificada. En vez de ello, `lp` envía la solicitud a un proceso llamado *servidor de impresión* o *daemon de impresión*. El servidor de impresión coloca la solicitud en una cola y provee un número de identificación por si el usuario decide cancelar el trabajo de impresión. Cuando se desocupa una impresora, el servidor de impresión comienza a copiar el archivo en el dispositivo de impresión. Es posible que el archivo por imprimir no se copie en un dispositivo de *spool* temporal a menos que el usuario especifique explícitamente que se haga. Muchas implementaciones de `lp` intentan crear un enlace firme con el archivo mientras está en espera de imprimirse para que el archivo no pueda eliminarse por completo. `lp` no siempre puede enlazarse con el archivo, y la página del manual advierte al usuario que no modifique el archivo antes de que se imprima.

Ejemplo 10.8

El siguiente comando `lp` *de UNIX envía el archivo* `myfile.ps` *a la impresora designada con* `ps`.

```
lp -dps myfile.ps
```

El comando `lp` *podría responder con un número de solicitud similar al siguiente.*

```
Request ps-358 queued
```

Utilice el número `ps-358` *en un comando* `cancel` *para eliminar el trabajo.*

Las impresoras son dispositivos lentos en comparación con los tiempos de ejecución de los procesos, así que un proceso servidor de impresión puede controlar muchas impresoras. Al igual que el problema de manejar entradas de múltiples descriptores, el control de impresión es una aplicación natural del enfoque multihilo. La figura 10.3 es un esquema de la organización de un servidor de impresión con hilos. El servidor se vale de un hilo dedicado para leer las solicitudes de los usuarios de una fuente de entrada. El hilo de solicitudes asigna espacio para la solicitud y la agrega a la cola de solicitudes.

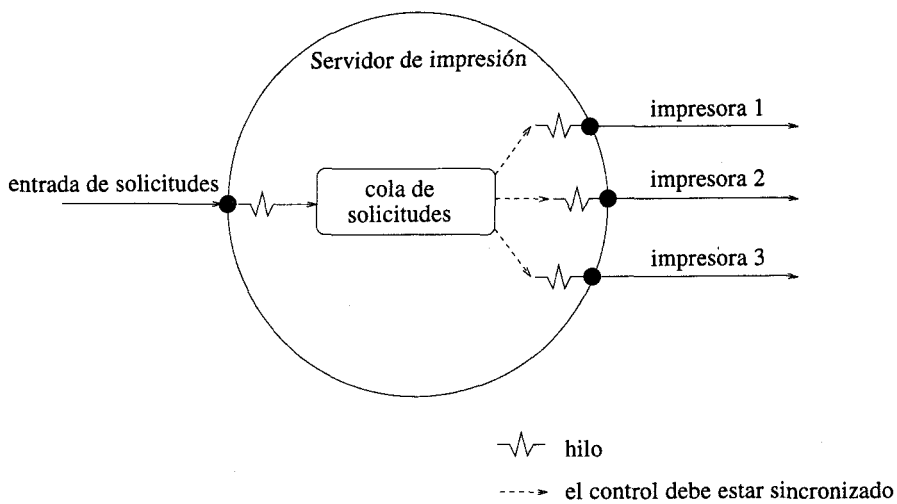


Figura 10.3: Esquema de un servidor de impresión con hilos.

El servidor de impresión de la figura 10.3 también tiene hilos dedicados para controlar sus impresoras. Cada hilo de impresora saca una solicitud de la cola de solicitudes y copia en la impresora el archivo especificado en la solicitud. Una vez que termina el copiado, el hilo de impresora libera la solicitud y atiende otra.

Los hilos dentro del servidor de impresión requieren una sincronización productor-consumidor con un solo productor (el hilo de solicitudes) y múltiples consumidores (los hilos de impresora). La cola en sí debe protegerse con mutex para que los elementos se agreguen y eliminarlos de forma consistente. Los consumidores deben sincronizarse respecto de las solicitudes disponibles en la cola para que no intenten sacar solicitudes inexistentes. La cola de solicitudes no está acotada porque el hilo de solicitudes asigna espacio dinámicamente a las solicitudes conforme van llegando. El hilo de solicitudes también podría usar una *marca de*

pleamar para limitar el número de solicitudes que pondrá en una cola antes de bloquearse. En esta situación, más complicada, el hilo de solicitudes se sincroniza con base en un predicado en el que interviene el tamaño de la cola.

Varios aspectos del servidor de impresión se han simplificado para este ejercicio. En un servidor real, la entrada se redirige desde cierto puerto de la red o bien las solicitudes se realizan mediante llamadas a procedimientos remotos. No es necesario que las impresoras sean idénticas, y las solicitudes de impresión realistas ofrecen diversas opciones a los usuarios para especificar cómo debe realizarse la impresión. El administrador del sistema puede instalar filtros por omisión que actúen sobre archivos de ciertos tipos. El servidor de impresión puede analizar el tipo de la solicitud y dirigirla a la impresora que mejor pueda realizar el trabajo. Se puede permitir que las solicitudes de impresión tengan prioridades u otras características que afecten la forma como se atienden. Los hilos de impresora individuales deben responder a condiciones de error y a informes de situación de los controladores de dispositivo de las impresoras.

Este ejercicio describe el servidor de impresión representado de forma esquemática en la figura 10.3. Las solicitudes pendientes se mantienen en una cola de solicitudes. Se debe sincronizar el número de solicitudes pendientes con una variable de condición llamada `items` siguiendo una estrategia similar al problema de productores-consumidores estándar. Este ejercicio no requiere una variable de condición para las ranuras (*slots*), ya que no hay límite para el crecimiento de la cola de solicitudes.

- Represente las solicitudes de impresión con un ID de usuario entero seguido de una cadena que especifique el nombre de trayectoria completo del archivo por imprimir.
- Represente la cola de solicitudes con una lista enlazada de nodos de tipo `prcmd_t`. Un ejemplo de definición es

```
typedef struct pr_struct {
    int owner;
    char filename[PATH_MAX];
    struct pr_struct *next_prcmd;
} prcmd_t;
static prcmd_t *pr_head = NULL;
static prcmd_t *pr_tail = NULL;
static int pending_requests = 0;
static pthread_mutex_t prmutex = PTHREAD_MUTEX_INITIALIZER;
```

Coloque las estructuras de datos de la cola de solicitudes en un archivo aparte al cual sólo se accederá por medio de las siguientes funciones.

➤ La función `add_queue` agrega un nodo a la cola de solicitudes. El prototipo es

```
int add_queue(prcmd_t *node);
```

La función `add_queue` incrementa `pending_requests` (solicitudes pendientes) e inserta el `node` al final de la cola de solicitudes. La función devuelve 0 si tiene éxito y `-1` si fracasa.

- La función `remove_queue` saca un nodo de la cola de solicitudes. El prototipo es

```
int remove_queue(prcmd_t **node);
```

  Si la cola no está vacía, la función `remove_queue` decrementa `pending_requests` y saca el primer nodo de la cola de solicitudes, estableciendo `*node` de modo que apunte al nodo retirado. La función `remove_queue` devuelve 0 si sacó con éxito un nodo o `-1` si la cola está vacía.
- La función `get_number_requests` devuelve el tamaño de la cola de solicitudes, que es el valor de `pending_requests`. El prototipo es

```
int get_number_requests(void);
```

- Defina una variable de condición llamada `items` y un candado mutex asociado llamado `items_lock`.
- Escriba la siguiente función que se invoca como un hilo para poner en una cola solicitudes de impresión:

```
void *get_requests(void *arg);
```

  La función `get_requests` agrega las solicitudes que llegan a la cola de solicitudes de impresión. El parámetro `arg` apunta a un descriptor de archivo abierto del cual se leen las solicitudes. `get_requests` lee el ID de usuario y el nombre de trayectoria del archivo por imprimir, crea un nodo `prcmd_t` para contener la información y llama a `add_queue` para agregar la solicitud a la lista de solicitudes de impresión. A continuación, la función `get_request` emite una señal para la variable de condición `items` con objeto de informar a los hilos de impresora en espera que hay solicitudes disponibles. Si `get_request` no puede asignar espacio para `prcmd_t` o si detecta un fin de archivo, regresa; de lo contrario, sigue vigilando el descriptor de archivo abierto para detectar la siguiente solicitud.
- Escriba un programa principal para probar `get_requests`. El programa principal crea el hilo `get_requests` con `STDIN_FILENO` como archivo de entrada; luego entra en un ciclo en el que espera hasta que `pending_requests` tiene un valor distinto de cero. Utilice la variable de condición `items` y su candado mutex asociado para lograr la sincronización con el hilo `get_requests`. El hilo principal saca la siguiente solicitud de la cola y envía el ID de usuario y el nombre de archivo a la salida estándar. Ejecute el programa con solicitudes de entrada introducidas desde el teclado. También pruebe el programa con entrada estándar redirigida desde un archivo.
- Escriba una función llamada `printer` que saque una solicitud de la cola de solicitudes de impresión y la "imprima". El prototipo de `printer` es

```
void *printer(void *arg);
```

  El parámetro `arg` apunta a un descriptor de archivo abierto al cual `printer` envía el archivo por imprimir. La función `printer` espera a que el contador `pending_requests`

sea distinto de cero en un ciclo similar al del programa principal, como se explicó en el paso anterior. Utilice la variable de condición `items` y su mutex asociado para acceder correctamente a `pending_requests`. Cuando esté disponible una solicitud, sáquela de la cola, abra para lectura el archivo especificado en el campo `filename` y copie el contenido del archivo en el archivo de salida. Después, cierre el archivo de entrada, libere el espacio ocupado por el nodo de solicitud y siga esperando la llegada de más solicitudes. Si ocurre un error durante la lectura del archivo de entrada, exhiba un mensaje de error apropiado, cierre el archivo de entrada y siga esperando más solicitudes. Puesto que el archivo de salida desempeña el papel de la impresora en este ejercicio, un error de salida corresponderá a una falla de impresora. Si `printer` encuentra un error de salida, debe cerrar el archivo de salida, exhibir un mensaje de error apropiado y regresar.

- Escriba un nuevo programa principal para implementar el servidor de impresión. El servidor podrá manejar un máximo de `MAX_PRINT` impresoras. (Cinco son suficientes para las pruebas.) El programa principal requiere dos argumentos de línea de comandos: el nombre base del archivo de salida y el número de impresoras. Las solicitudes de entrada se toman de la entrada estándar que puede redirigirse para tomar solicitudes de un archivo. La salida a cada impresora se envía a un archivo distinto cuyo nombre comienza con el nombre base del archivo de salida. Por ejemplo, si el nombre base es `printer.out`, los archivos de salida son `printer.out1`, `printer.out.2`, y así sucesivamente. El programa principal crea un hilo para ejecutar `get_requests` y un hilo `printer` para cada impresora que se puede controlar. A continuación, el programa principal espera que todos los hilos terminen antes de terminar él mismo. Pruebe exhaustivamente el servidor de impresión.
- Agregue funciones para que cada hilo `printer` mantenga datos estadísticos como el número de archivos impresos y el número total de bytes impresos. Cuando el servidor reciba una señal `SIGUSR1`, enviará los datos estadísticos de todas las impresoras a error estándar. Maneje la señal `SIGUSR1` añadiendo un hilo que ejecute `sigwait` para `SIGUSR1`, adquiera los candados apropiados y envíe a la salida los datos estadísticos. Todos los demás hilos, incluido el principal, deberán bloquear `SIGUSR1`.
- Agregue funciones de modo que la entrada incluya un comando además de un ID de usuario y un nombre de archivo. Los comandos son

`lp:`	Agregar la solicitud a la cola y enviar un ID de solicitud a la salida estándar.
`cancel:`	Sacar la solicitud de la cola si está ahí.
`lpstat:`	Enviar un resumen de todas las solicitudes pendientes y de las solicitudes que se están imprimiendo en cada impresora.

  Agregue funciones apropiadas al archivo de cola según sea necesario. El comando `lp` envía un número de solicitud a la salida estándar que puede utilizarse en comandos `cancel` posteriores.
- Agregue sincronización de modo que `get_request` utilice una marca de pleamar (`high_water_mark`) y una marca de bajamar (`low_water_mark`) para controlar

el tamaño de la cola de solicitudes. Si el número de solicitudes llega al valor de `high_water_mark`, `get_request` se bloqueará hasta que el tamaño de la cola de solicitudes sea menor que `low_water_mark`.

10.6 Lecturas adicionales

La mayor parte de las obras clásicas sobre sistemas operativos analizan alguna variación del problema de productores-consumidores. Véase, por ejemplo, [10, 92]. Desafortunadamente, en casi todos los tratamientos clásicos, los productores y consumidores entran en ciclos infinitos, sin ser interrumpidos por señales u otras complicaciones que surgen de un universo finito. Pronto aparecerá el libro *Programming with Threads* [44], que incluye una sección muy completa sobre manejo de señales con hilos. La *Solaris Multithreaded Programming Guide* [100], aunque se ocupa primordialmente de los hilos Solaris, contiene algunos ejemplos de sincronización interesantes.

Capítulo 11

Proyecto: *La máquina virtual no muy paralela*

Grace Murray Hopper, una de las primeras y más expresivas partidarias de la computación en paralelo, gustaba de recordar a quienes la escuchaban que la forma de acarrear una carga más pesada no era criando un buey más grande. La idea de juntar estaciones de trabajo baratas para resolver problemas de gran envergadura se ha hecho cada vez más atractiva, pero las dificultades de proveer *software* que coordine las actividades de estas máquinas siguen siendo el principal obstáculo para que se difunda su uso. La PVM (*Parallel Virtual Machine*, máquina virtual paralela) proporciona un sistema de alto nivel, pero no transparente, que permite a un usuario coordinar tareas dispersas entre varias estaciones de trabajo en una red. Este proyecto compara dos estrategias para implementar un despachador de una máquina virtual no muy paralela (NTPVM, Not Too Parallel Virtual Machine), que es un sistema PVM simplificado. El enfoque de un solo hilo utiliza `select` o `poll` para gestionar descriptores de archivo, mientras que el enfoque multihilo asigna un hilo dedicado a cada descriptor de entrada. La segunda estrategia simplifica la lógica y mejora el paralelismo.

Los sistemas de programación como PVM (máquina virtual paralela) [87] y Linda [18] se valen de grupos de máquinas heterogéneas interconectadas para proporcionar un entorno de computación paralela transparente. Estos sistemas permiten a los usuarios resolver problemas de gran tamaño en redes de estaciones de trabajo creando la ilusión de una sola máquina paralela. Linda se basa en un modelo de programación de espacio de tuplas y proporciona una abstracción de memoria compartida distribuida, mientras que PVM opera en el nivel de tareas y proporciona una abstracción de transferencia de mensajes. En ambos casos, el usuario percibe un solo sistema unificado, y los detalles de la red y las máquinas individuales que constituyen

la *máquina virtual* no son visibles directamente. En este capítulo nos concentraremos en una implementación tipo PVM, y en el capítulo 15 exploraremos el paradigma de Linda.

La unidad básica de cómputo en PVM se llama *tarea (task)* y es análoga a un proceso UNIX. Un programa PVM invoca funciones de biblioteca PVM para crear y coordinar tareas. Las tareas pueden comunicarse pasando mensajes a otras tareas mediante llamadas a funciones de biblioteca PVM. Las tareas que cooperan ya sea mediante comunicación o sincronización se organizan en grupos llamados *cómputos (computation)*. PVM maneja comunicación directa, difusión y barreras dentro de un cómputo.

En la figura 11.1 se muestra un diagrama lógico de una aplicación PVM representativa. Una aplicación PVM generalmente se inicia con una tarea de entrada y partición de tarea que controla la forma en que se resolverá el problema. En esta tarea, el usuario especifica cómo otras tareas cooperarán para resolver un problema. La tarea de entrada y partición de tarea crea varios cómputos. Las tareas dentro de cada cómputo comparten datos y se comunican entre sí. La aplicación PVM tiene además una tarea dedicada, se encarga de la salida y la exhibición para el usuario. Las demás tareas de la aplicación PVM envían sus salidas a esta tarea para que se exhiban en la consola de la aplicación.

Para ejecutar una aplicación PVM, lo primero que debe hacer el usuario es designar el conjunto de máquinas o nodo que constituyen la *máquina virtual* y luego inicia el demonio de control de PVM, `pvmd`, en cada uno de estos nodos. El demonio de control se comunica con la consola del usuario y se encarga de la comunicación y control de las tareas en su máquina. Cuando la PVM necesita enviar entradas a una tarea en particular, envía los datos al demonio `pvmd` del nodo de destino, el cual a su vez los reenvía a la tarea apropiada. De forma similar, una tarea produce salidas enviando un mensaje a su `pvmd`, el cual a su vez reenvía el mensaje

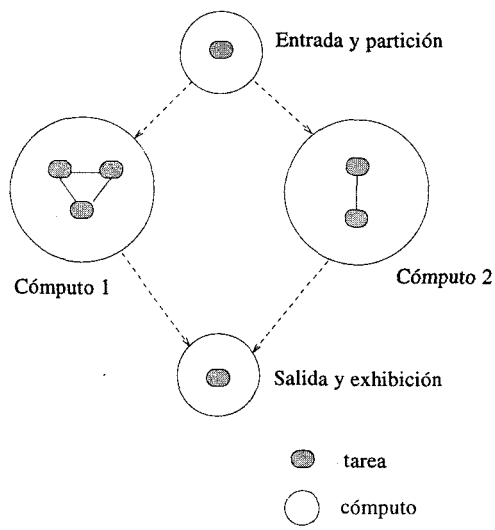


Figura 11.1: Diagrama lógico de una aplicación que se ejecuta en una máquina virtual PVM.

al pvmd de la consola, que entonces envía el mensaje a la tarea de salida de la aplicación. La transferencia de mensajes subyacente es transparente, y lo único que el usuario ve es que cierta tarea envió un mensaje a la consola. En la figura 11.2 se muestra cómo se establece la correspondencia entre una aplicación y la máquina virtual. Las tareas que constituyen un cómputo lógico no necesariamente se asignan al mismo nodo; podrían distribuirse entre todos los nodos de la máquina virtual.

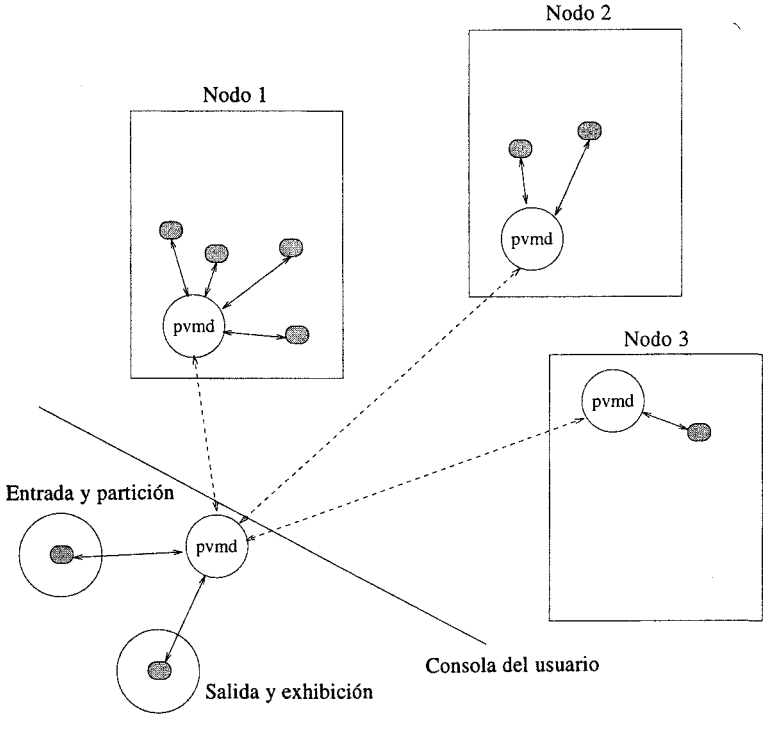


Figura 11.2: Diagrama de una máquina virtual PVM.

11.1 La máquina virtual no muy paralela

La máquina virtual no muy paralela (NTPVM) es un despachador que posee muchas de las características de un demonio de control de PVM, pvmd. El despachador NTPVM se encarga de crear y controlar tareas, como se muestra esquemáticamente en la figura 11.3. El despachador recibe solicitudes a través de su entrada estándar y responde a través de su salida estándar. (Posteriormente, la entrada y la salida estándar se pueden redirigir a puertos de comunicación

de la red.) El despachador podría recibir una solicitud pidiéndole crear una tarea o enviar de nuevo datos a una tarea que está bajo su control.

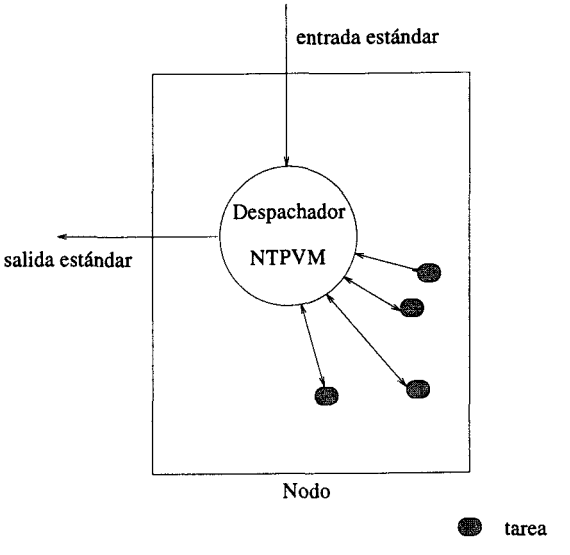


Figura 11.3: Diagrama del despachador NTPVM.

Una tarea no es más que un proceso que ejecuta un programa especificado. Cada tarea se identifica mediante un ID de cómputo y un ID de tarea. Cuando el despachador recibe una solicitud para crear una tarea con un ID de cómputo y un ID de tarea determinados, crea un par de entubamientos y bifurca un hijo para ejecutar la tarea. En la figura 11.4 se muestra la capa de comunicación entre una tarea y su despachador. El entubamiento que lleva la comunicación del despachador a la tarea hija se etiqueta con `writefd` en el extremo del despachador. El hijo redirige su entrada estándar a este entubamiento. De forma similar, el entubamiento que lleva la comunicación del hijo al despachador se rotula con `readfd` en el extremo del despachador. El hijo redirige su salida estándar a este entubamiento.

El despachador se encarga de la entrega de datos a las tareas, la entrega de salidas de las tareas, la difusión a las tareas que tienen el mismo ID de cómputo y la cancelación. NTPVM es más sencillo que el PVM real en varios aspectos. PVM cuenta con entrega de mensajes en orden y permite a cualquier tarea comunicarse con otras tareas de su cómputo; además, cuenta con un mecanismo de memoria temporal para retener mensajes. PVM también ofrece avanzadas herramientas para vigilar los cómputos. NTPVM entrega mensajes cada vez que los recibe, no apoya la comunicación punto por punto entre tareas y cuenta con capacidades de vigilancia rudimentarias.

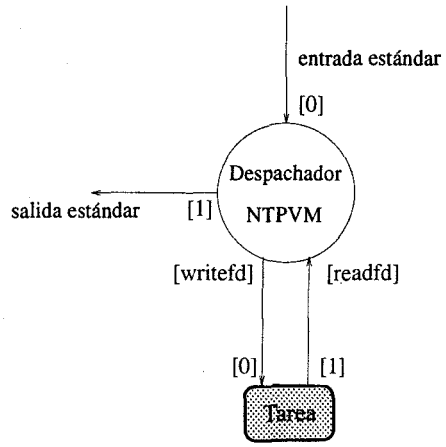


Figura 11.4: El despachador NTPVM se comunica con sus hijos mediante entubamientos (*pipes*).

11.2 Panorama general del proyecto NTPVM

En este capítulo se especificarán dos implementaciones de la máquina virtual no muy paralela. Una versión, designada por A, tiene un solo hilo de ejecución y utiliza `select` o `poll` para controlar múltiples descriptores de archivo. La otra versión, designada por B, es multihilo. Trate de realizar las primeras partes de ambas versiones y luego elija partes adicionales por implementar ya sea de la serie A o de la serie B.

Las tareas de la NTPVM son procesos independientes agrupados en unidades llamadas cómputos. El despachador se encarga de crear y gestionar las tareas. En general, las tareas de un cómputo no tienen que residir en la misma máquina, y la especificación del proyecto se diseñó pensando en esta extensión. Sin embargo, un solo despachador controla todos los cómputos en el proyecto descrito en el presente capítulo.

El despachador se comunica con el mundo exterior leyendo paquetes de su entrada estándar y escribiendo paquetes en su salida estándar. El despachador podría recibir un paquete indicándole que debe crear una tarea nueva, o un paquete de datos destinado a una tarea que está bajo su control. El despachador envía a su vez las salidas generadas por las tareas a su propia salida estándar en forma de paquetes. En las primeras partes del proyecto, las tareas envían datos ASCII y el despachador envuelve los datos en un paquete. Posteriormente, las tareas generarán los paquetes ellas mismas.

El programa 11.1 muestra el archivo de encabezado `ntpvm.h` que contiene las definiciones de tipo pertinentes para el despachador. Incluya este archivo en todos los programas de este proyecto.

Programa 11.1: El `ntpvm.h` archivo encabezado.

```
typedef enum ptype {START_TASK, DATA, BROADCAST, DONE,
                    TERMINATE, BARRIER} packet_t;

typedef struct {
     int comp_id;
     int task_id;
     packet_t type;
     int length;
} task_packet_t;

typedef struct {
     int comp_id;                    /* ID de cómputo para la tarea */
     int task_id;                      /* ID de tarea para la tarea */
     int writefd;                   /* contiene fd despachador->hijo */
     int readfd;                    /* contiene fd hijo->despachador */
     int total_packs_sent;
     int total_bytes_sent;
     int total_packs_recv;
     int total_bytes_recv;
     pid_t task_pid;          //* ID de proceso de la tarea bifurcada */
     int barrier; /* número de barrera o -1 si no está en la barrera */
     int end_of_input;    /* true si ya no hay entradas para la tarea */
} task_t;

#define MAX_PACK_SIZE 1024
#define MAX_TASKS 10
```

Programa 11.1

Los paquetes del despachador tienen el siguiente formato:

- Un ID de cómputo.
- Un ID de tarea.
- Un tipo de paquete.
- La longitud de la información del paquete.
- La información del paquete.

Los primeros cuatro elementos de un paquete constituyen el *encabezado del paquete*, de longitud fija, que se almacena en una estructura de tipo `task_packet_t`. Suponga que la porción de información del paquete no contiene más de `MAX_PACK_SIZE` bytes.

El despachador mantiene información acerca de cada una de las tareas activas en un arreglo global `tasks` de tipo `task_t`. No permita que el despachador ejecute más de `MAX_TASKS` tareas simultáneas. El arreglo `tasks` se declara mediante

```
task_t tasks[MAX_TASKS];
```

Asigne inicialmente –1 al miembro `comp_id` (ID de cómputo) de cada elemento del arreglo `tasks` para indicar que la ranura está vacía. Cuando el despachador crea una tarea, encuentra una ranura vacía en el arreglo `tasks` para contener la información pertinente acerca de la nueva tarea.

En total, hay seis tipos de paquetes de despachador: `START_TASK`, `DATA`, `BROADCAST`, `DONE`, `TERMINATE` y `BARRIER`. El despachador interpreta estos tipos de paquetes como sigue:

- Cuando el despachador recibe un `START_TASK` por la entrada estándar, inicia una tarea nueva. La porción de información de este paquete contiene la línea de comandos que la tarea hija bifurcada ejecutará mediante `execvp`. El despachador crea dos entubamientos (`readfd` y `writefd` en la figura 11.4) y bifurca un hijo para que ejecute mediante `execvp` el comando especificado.
- Los paquetes `DATA` que se pasan al despachador desde la entrada estándar se tratan como datos de entrada para la tarea identificada mediante los miembros ID de cómputo e ID de tarea del encabezado del paquete. En las primeras partes del proyecto, el despachador quitará el encabezado del paquete y escribirá los datos reales del paquete en `writefd` de la tarea apropiada.
- Cuando una tarea escribe datos en su salida estándar, a su vez el despachador los reenvía a la salida estándar. En las primeras partes de este proyecto las tareas serán utilerías UNIX estándar. Puesto que estos comandos sólo producen texto ASCII como salida, el paquete empacará los datos en paquetes `DATA` antes de enviarlos a la salida estándar.
- Cuando el despachador recibe un paquete `DONE` por la entrada estándar, cierra el descriptor de archivo `writefd` para la tarea identificada por los miembros ID de cómputo e ID de tarea del encabezado del paquete. La tarea correspondiente detectará entonces un fin de archivo en su entrada estándar.
- Cuando el despachador detecta un fin de archivo en el `readfd` de una tarea, realiza las operaciones de limpieza apropiadas y envía un paquete `DONE` por la salida estándar para indicar que la tarea ya terminó.
- El despachador envía todos los paquetes `BROADCAST` (difusión) que recibe por la entrada estándar a todas las tareas del cómputo especificado.
- Si una tarea envía un paquete `BROADCAST` al despachador, éste enviará la solicitud a todas las tareas del mismo cómputo, y también enviará la solicitud por su salida estándar. De este modo, todas las tareas de un cómputo recibirán el mensaje.
- Si el despachador recibe un paquete `TERMINATE` por su entrada estándar, matará la tarea identificada por el ID de cómputo y el ID de tarea del paquete.
- Los paquetes `BARRIER` (barrera) sincronizan las tareas de un cómputo en un punto determinado de su ejecución.

El proyecto NTPVM tiene las siguientes partes:

Parte I:	Preparación de E/S y pruebas [sección 11.3].
Parte II:	Una sola tarea sin entradas (manejar `START_TASK` y datos que salen) [sección 11.4].
Parte III:	Una tarea a la vez (manejar paquetes `START_TASK`, `DATA` y `DONE`) [sección 11.5].
Parte IV:	Múltiples tareas y cómputos (manejar paquetes `START_TASK`, `DATA` y `DONE`) [sección 11.6].
Parte V:	Manejar paquetes `BROADCAST` y `BARRIER` [sección 11.7].
Parte VI:	Manejar paquetes `TERMINATION` y señales [sección 11.8].

En las primeras partes del proyecto, las tareas hijas no se comunicarán mediante paquetes, y el despachador quitará los encabezados de los paquetes antes de escribirlos en `writefd`. Este formato permite al despachador ejecutar como tareas utilerías UNIX ordinarias como `cat` o `ls`. En la parte V las tareas se comunican con el despachador empleando paquetes. A esas alturas, el programa requerirá programas de tarea específicos para pruebas de NTPVM. En el resto de esta sección daremos ejemplos de los distintos tipos de paquetes y de cómo debe manejarlos el despachador.

11.2.1 Paquetes `START_TASK`

El despachador de NTPVM recibe comandos por su entrada estándar (denotada por `[0]`) e informa de los resultados por su salida estándar (denotada por `[1]`). El despachador espera la llegada de un paquete `START_TASK` por su entrada estándar. Este tipo de paquete incluye un ID de cómputo, un ID de tarea y una cadena de línea de comandos.

Ejemplo 11.1

El siguiente paquete `START_TASK` *solicita la creación de la tarea 2 en el cómputo 3 para ejecutar* `ls -l`.

ID de cómputo:	`3`
ID de tarea:	`2`
Tipo de paquete:	`START_TASK`
Longitud de datos del paquete:	`5`
Información del paquete:	`ls -l`

Los datos contenidos en el paquete del ejemplo 11.1 no son una cadena terminada por `NULL`. El despachador debe convertir los datos en una cadena de ese tipo antes de pasarlos a `makeargv` o a `execvp`.

El despachador selecciona una entrada desocupada en el arreglo `tasks` para almacenar la información relativa a la nueva tarea y asigna el valor inicial 0 a los miembros `total_packs_sent` (total de paquetes enviados), `total_bytes_sent` (total de bytes enviados), `total_packs_recv`

(total de paquetes recibidos), `total_bytes_recv` (total de bytes recibidos) y `end_of_input` (fin de entradas) de la entrada del arreglo `tasks`. El despachador también asigna el valor inicial –1 al miembro `barrier` para indicar que la tarea no está esperando en una barrera.

A continuación, el despachador crea dos entubamientos para la comunicación bidireccional con una tarea hija. El despachador utiliza dos de los cuatro descriptores de archivo de entubamiento resultantes para comunicarse con la tarea hija. Estos descriptores se almacenan en los miembros `readfd` y `writefd` de la entrada del arreglo `tasks`. El despachador bifurca un hijo y almacena su ID de proceso en el miembro `task_pid` de la entrada de `tasks`; luego cierra los descriptores de archivo de entubamiento que no utilizará y espera E/S ya sea de su entrada estándar o de los descriptores `readfd` de sus tareas.

Cuando se bifurca un hijo, éste redirige su entrada y su salida estándar a los entubamientos y cierra los descriptores de archivo no utilizados. A continuación, el hijo puede ejecutar, mediante `execvp`, el comando que realizará la tarea. Utilice la función `makeargv` del programa 1.2 para crear un arreglo de argumentos que será la entrada de `execvp`.

11.2.2 Paquetes DATA

Cuando llega un paquete `DATA` por la entrada estándar, el despachador verifica si tiene una tarea cuyos ID de tarea e ID de cómputo coincidan con los que vienen en la cabecera del paquete. Si no es así, el paquete se desecha. Si los identificadores coinciden, el despachador actualiza los miembros `total_packs_recv` y `total_bytes_recv` de la entrada de la tarea en el arreglo `tasks`. En las primeras partes del proyecto, las tareas son utilerías UNIX estándar que aceptan entradas en ASCII. El despachador reenvía la porción de información del paquete a la tarea en el descriptor `writefd` de la tarea. En las partes V y VI, las tareas reciben los paquetes de datos completos directamente.

Ejemplo 11.2

Después de recibir el siguiente paquete `DATA`*, el despachador envía las palabras* `This is my data` *a la tarea 2 en el cómputo 3.*

ID de cómputo:	`3`
ID de tarea:	`2`
Tipo de paquete:	`DATA`
Longitud de datos del paquete:	`15`
Información del paquete:	`Estos son mis datos`

El despachador también reenvía los datos recibidos de tareas individuales a su salida estándar en forma de paquetes `DATA`. En las primeras partes del proyecto, el despachador interpreta las entradas de `readfd` como entradas en bruto de la tarea, crea un paquete `DATA` con el ID de cómputo y el ID de tarea de la tarea y utiliza la información que leyó de `readfd` como porción de información del paquete. A continuación, el despachador escribe el paquete `DATA` en la salida estándar. A partir de la sección 11.7 cada tarea leerá y escribirá sus datos en formato de paquete.

11.2.3 Paquetes DONE

Cuando el despachador recibe un paquete DONE por la entrada estándar, establece el miembro `end_of_input` (fin de entrada) de la entrada que corresponde a la tarea en cuestión en el arreglo `tasks` y cierra el descriptor `writefd` de esa tarea. El despachador desechará todos los paquetes DONE o DATA adicionales que lleguen para esa tarea.

Ejemplo 11.3

El siguiente paquete DONE *indica que ya no hay más datos de entrada para la tarea 2 del cómputo 3.*

ID de cómputo:	3
ID de tarea:	2
Tipo de paquete:	DONE
Longitud de datos del paquete:	0
Información del paquete:	NULL

Cuando el despachador recibe una indicación de fin de archivo por `readfd`, cierra ese descriptor y envía un paquete DONE por la salida estándar. Si el descriptor `writefd` para esa tarea sigue abierto, el despachador lo cierra. En algún momento, el despachador tendrá que ejecutar un `wait` para el proceso de tarea hijo y asignar el valor –1 al miembro `comp_id` de la entrada correspondiente del arreglo `tasks` para que esa entrada del arreglo pueda reutilizarse.

Si el despachador recibe una indicación de fin de archivo por su entrada estándar, cierra los descriptores `writefd` de todas las tareas activas y asigna 1 al miembro `end_of_input` de cada una de las tareas activas. Una vez que reciba una indicación de fin de archivo por los descriptores `readfd` de todas las tareas activas, el despachador esperará cada una de las tareas y terminará.

11.3 E/S y prueba del despachador

En esta sección se describirán las funciones de E/S del despachador y se preparará el diseño de depuración. El despachador recibe datos de entrada de la entrada estándar llamando a `get_packet` (obtener paquete) y envía datos de salida a la salida estándar llamando a `put_packet` (colocar paquete), como se muestra en la figura 11.5. Los datos siempre se transfieren en dos partes. Primero, el despachador lee o escribe un encabezado de tipo `task_packet_t`. Después, el despachador utiliza el miembro de longitud del encabezado para determinar cuántos bytes de información de paquete debe leer o escribir. Por último, el despachador lee o escribe la porción de datos del paquete. Suponga que el campo de información del paquete contiene cuando mucho `MAX_PACK_SIZE` bytes, así que el despachador puede utilizar un *buffer* de longitud fija (`MAX_PACK_SIZE` bytes) para contener la información del paquete durante las operaciones de entrada y salida.

La función `get_packet` tiene el siguiente prototipo:

```
int get_packet(int fd, int *comp_idp, int *task_idp,
               packet_t *typep, int *lenp, unsigned char *buf);
```

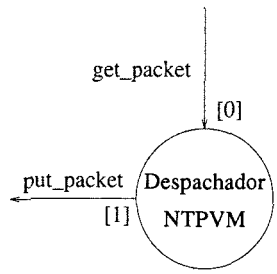


Figura 11.5: E/S básica del despachador.

La función `get_packet` lee una cabecera `task_packet_t` de `fd` y luego lee y coloca en `buf` el número de bytes especificado por el miembro `length`. `get_packet` devuelve 0 si tiene éxito o –1 si hay un error. La función `get_packet` asigna valores a `*comp_idp`, `*task_idp`, `*typep` y `*lenp` de los miembros `comp_id`, `task_id`, `type` y `length` del encabezado del paquete, respectivamente.

La función `put_packet` tiene el siguiente prototipo:

```
int put_packet(int fd, int comp_id, int task_id,
               packet_t type, int len, unsigned char *buf);
```

La función `put_packet` arma una cabecera `task_packet_t` a partir de `comp_id`, `task_id`, `type` y `len`. A continuación, escribe el encabezado del paquete en `fd` seguida de `len` bytes tomados de `buf`. La función `put_packet` devuelve 0 si tiene éxito o –1 si hay un error.

Ejemplo 11.4

El siguiente programa utiliza `get_packet` *y* `put_packet` *para copiar paquetes de la entrada estándar a la salida estándar.*

```
#include <stdio.h>
#include <unistd.h>
#include "ntpvm.h"

int get_packet(int, int *, int *, packet_t *, int *, unsigned char *);
int put_packet(int, int, int, packet_t, int, unsigned char *);

void main(void)
{
   int in, out;
   int comp_id;
   int task_id;
   packet_t type;
   int len;
   unsigned char buf[MAX_PACK_SIZE];
```

```
    in = STDIN_FILENO;
    out = STDOUT_FILENO;
    while (get_packet(in, &comp_id, &task_id, &type, &len, buf) != -1) {
        if (put_packet(out, comp_id, task_id, type, len, buf) == -1)
            break;
    }
}
```

La especificación para la parte I del proyecto es

- Convertir el segmento de código del ejemplo 11.4 en un programa principal.
- Escribir las funciones `get_packet` y `put_packet`.
- Compilar y enlazar el programa para asegurarse de que no haya errores de sintaxis.
- Probar el programa empleando entubamientos con nombre como se describe a continuación.
- Agregar mensajes para depuración al ciclo del programa principal indicando los valores que se están leyendo y escribiendo. Todos los mensajes de depuración deben enviarse al error estándar.

La parte más difícil del proyecto NTPVM es la prueba del despachador. El despachador se comunica con la entrada estándar y la salida estándar por medio de paquetes que incluyen componentes no ASCII. Durante la depuración, el despachador deberá enviar al error estándar mensajes que indiquen su grado de avance. Se requiere cierto trabajo para aislar la salida y la entrada del despachador de los mensajes informativos dirigiendo los tres tipos de E/S para que aparezcan en formato ASCII en diferentes pantallas.

Comience con dos filtros sencillos, `a2ts` y `ts2a`, que se muestran en los programas 11.2 y 11.3, respectivamente. El filtro `a2ts` lee caracteres ASCII de la entrada estándar, construye un paquete de tarea y lo escribe en la salida estándar. En uso interactivo, `a2ts` solicita al usuario la información requerida, enviando las indicaciones al error estándar. El filtro `ts2a` lee un paquete de tarea de la entrada estándar y escribe el contenido del paquete en la salida estándar en formato ASCII. Para este proyecto, suponga que la porción de información de un paquete de tarea siempre contiene información en ASCII.

Programa 11.2: El filtro `a2ts` solicita información y envía un paquete a la salida estándar.

```
/*
 * Este filtro convierte texto ASCII en paquetes de tarea.
 * Se solicitan tres enteros empleando el error estándar:
 *     comp_id, task_id, type,
 * Los tres enteros están en formato libre.
 *
 * El texto ASCII debe ser un conjunto de líneas
 * terminado por una línea que contenga sólo 5 «!».
 * El texto ASCII debe consistir en líneas completas.
```

```
 *
 * Esta función está diseñada para usuarios con experiencia,
 *  de modo que la verificación de errores es mínima.
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include "ntpvm.h"
#define MAX_LINE_SIZE 100
#define TERMINATE_STRING "!!!!!"

void main(void)
{
   task_packet_t pack;
   int type_int;
   int wsize;
   int line_len;
   char buf[MAX_PACK_SIZE + MAX_LINE_SIZE];
   char *bufptr;


   wsize = sizeof(task_packet_t);
   fprintf(stderr, "Ready for first packet\n");
   for( ; ; ) {
      fprintf(stderr, "Enter comp_id:");
      if (scanf("%d", &pack.comp_id) == EOF) {
         fprintf(stderr, "Exiting ...\n");
         break;
      }
      fprintf(stderr, "Enter task_id:");
      scanf("%d", &pack.task_id);
      fprintf(stderr, "Enter task type:\n");
      fprintf(stderr, "    0 = START_TASK\n");
      fprintf(stderr, "    1 = DATA\n");
      fprintf(stderr, "    2 = BROADCAST\n");
      fprintf(stderr, "    3 = DONE\n");
      fprintf(stderr, "    4 = TERMINATE\n");
      fprintf(stderr, "    5 = BARRIER\n");
      scanf("%d", &type_int);
      pack.type = type_int;
      gets(buf);        /* saltar una nueva línea después del tercer int*/
      pack.length = 0;
      bufptr = buf;
      *bufptr = 0;
      fprintf(stderr, "Enter first line of data (%s to end):\n",
         TERMINATE_STRING);

      while (fgets(bufptr, MAX_LINE_SIZE+ 1, stdin) != NULL) {
```

```
            line_len = (int)strlen(bufptr);
            if (line_len == 0)
               continue;
            if (*(bufptr + line_len - 1) != '\n') {
               *(bufptr + line_len - 1) = '\n';
               *(bufptr + line_len) = 0;
               line_len++;
            }
            if ( (line_len == (strlen(TERMINATE_STRING) + 1)) &&
                 !strncmp(bufptr,TERMINATE_STRING,line_len - 1) )
               break;
            bufptr = bufptr + line_len;
            pack.length = pack.length + line_len;
            if (pack.length >= MAX_PACK_SIZE) {
               fprintf(stderr, "**** Maximum packet size exceeded\n");
               exit(1);
            }
            fprintf(stderr,

               "Length %d received, total=%d, Enter line (%s to end):\n",
                line_len, pack.length, TERMINATE_STRING);
         }
         fprintf(stderr, "Writing packet header: %d %d %d %d\n",
             pack.comp_id, pack.task_id, (int)pack.type, pack.length);
         if (write(STDOUT_FILENO, &pack, wsize) != wsize) {
            fprintf(stderr, "Error writing packet\n");
            exit(1);
         }
         fprintf(stderr, "Writing %d bytes\n", pack.length);
         if (write(STDOUT_FILENO, buf, pack.length) != pack.length) {
            fprintf(stderr,"Error writing packet\n");
            exit(1);
         }
         fprintf(stderr, "Ready for next packet\n");
      }
      exit(0);
}
```

Programa 11.2

Programa 11.3: El filtro `ts2a` lee un paquete de la entrada estándar y exhibe el encabezado y los datos en la salida estándar en formato ASCII.

```
/*
 * Este filtro lee paquetes de tarea de la entrada estándar
 * y los exhibe en formato ASCII en la salida estándar.
 * Los mensajes que esperan aparecen en el error estándar.
```

```
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "ntpvm.h"
#define MAX_LINE_SIZE 100

char *get_type_string(packet_t ptype)
{
   switch (ptype) {
      case START_TASK:
         return "Start Task";
      case DATA:
         return "Data";
      case BROADCAST:
         return "Broadcast";
      case DONE:
         return "Done";
      case TERMINATE:
         return "Terminate";
      case BARRIER:
         return "Barrier";
      default:
         return "Unknown";
   }
}

void main(void)
{
   task_packet_t pack;
   int wsize;
   char buf[MAX_PACK_SIZE + MAX_LINE_SIZE];
   char *type_string;
   int bytes_read;

   wsize = sizeof(task_packet_t);
   fprintf(stderr, "***** Waiting for first packet\n");
   for( ; ; ) {
      bytes_read =  read(STDIN_FILENO, &pack, wsize);
      if (bytes_read == 0) {
         fprintf(stderr, "End of File received\n");
         exit(0);
      }
      if (bytes_read != wsize) {
         fprintf(stderr, "Error reading packet header\n");
         exit(1);
      }
      type_string = get_type_string(pack.type);
      printf("Received packet header of type %s\n",type_string);
```

```
        printf("   comp_id = %d, task_id = %d, length = %d\n",
               pack.comp_id, pack.task_id, pack.length);
        fflush(stdout);
        if (pack.length > MAX_PACK_SIZE) {
           fprintf(stderr, "Task data is too long\n");
           exit(1);
        }
        if (read(STDIN_FILENO, buf, pack.length) != pack.length) {
           fprintf(stderr, "Error reading packet data\n");
           exit(1);
        }
        write(STDOUT_FILENO, buf, pack.length);
        fprintf(stderr, "***** Waiting for next packet\n");
    }
}
```

Programa 11.3

Ejemplo 11.5

El siguiente comando solicita los campos de un paquete y luego hace eco del paquete en la salida estándar en formato ASCII.

```
a2ts | ts2a
```

En el ejemplo 11.5 el programa `a2ts` solicita interactivamente información del paquete y la envía a su salida estándar en formato binario. La salida estándar de `a2ts` se entuba hacia la entrada estándar de `ts2a`. El programa `ts2a` lee paquetes binarios de su entrada estándar y los envía en formato ASCII a su salida estándar.

Ejemplo 11.6

El siguiente comando muestra un posible método para probar el despachador de forma interactiva.

```
a2ts | dispatcher | ts2a
```

El ejemplo 11.6 entuba la salida estándar de `a2ts` hacia la entrada estándar del despachador y la salida estándar del despachador hacia `ts2a`. Si bien el comando del ejemplo 11.6 permite a un usuario introducir datos ASCII y exhibir la salida del paquete de tarea en ASCII, no es tan útil como podría ser porque la pantalla recibe demasiados datos de muchas fuentes distintas. Es difícil distinguir cuál información proviene de cuál programa. Una mejor solución sería utilizar un sistema de ventanas con tres de éstas asignadas a la prueba del despachador, como se muestra en la figura 11.6. Utilice la *ventana de entrada* para introducir paquetes en formato ASCII y para enviarlos a la entrada estándar del despachador. Utilice la *ventana de salida* para

exhibir los paquetes que salen por la salida estándar del despachador. La *ventana del despachador* muestra la salida de error estándar del despachador e indica lo que éste está haciendo.

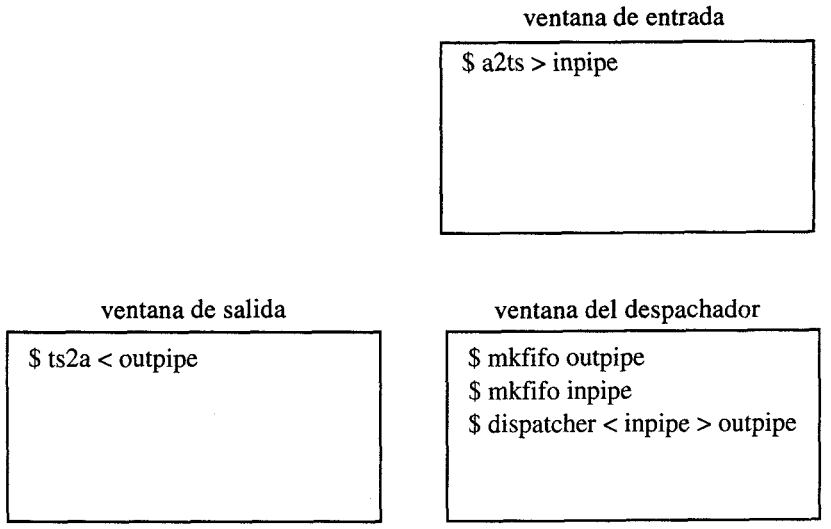


Figura 11.6: Uso de tres ventanas para depurar el despachador NTPVM.

En la figura 11.6 se muestra la configuración para tres ventanas. Asegúrese de utilizar el mismo directorio de trabajo en las tres. El procedimiento para ejecutar el despachador es el siguiente:

- Cree dos tubos con nombre en la ventana del despachador ejecutando los siguientes comandos:

```
mkfifo outpipe
mkfifo inpipe
```

- Inicie el despachador en la ventana del despachador ejecutando el siguiente comando:

```
dispatcher < inpipe > outpipe
```

  Esta ventana sólo exhibirá los mensajes que el despachador envíe al error estándar, ya que tanto la entrada estándar como la salida estándar se han redirigido.

- En la ventana de salida, ejecute el siguiente comando:

```
ts2a < outpipe
```

  Esta ventana exhibe los paquetes que salen por la salida estándar del despachador.

- En la ventana de entrada, ejecute el siguiente comando:

```
a2ts > inpipe
```

  Esta ventana exhibe las indicaciones para que el usuario introduzca paquetes. El programa `a2ts` convierte la información introducida de ASCII a formato de paquete y la escribe en la entrada estándar del despachador.

En la figura 11.7 se muestra la disposición de las ventanas para la depuración. Si no cuenta con una estación de trabajo que maneje múltiples ventanas, trate de convencer al administrador de su sistema que instale un programa como `screen`, que puede crear múltiples pantallas en una terminal ASCII.

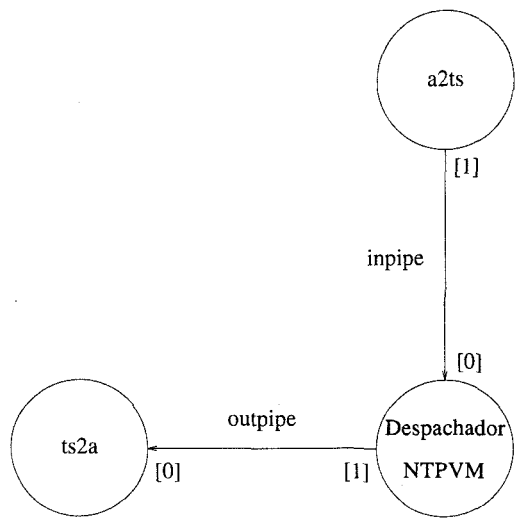


Figura 11.7: Organización de los procesos lógicos para depurar el despachador.

11.4 Una sola tarea sin entradas

Esta parte del proyecto utiliza una sola tarea que no tiene entradas a fin de probar el código que crea las tareas y los entubamientos que hacen posible la comunicación sin la complicación adicional de vigilar varios descriptores de archivo para la entrada.

El despachador lee un solo paquete `START_TASK` de la entrada estándar, crea los entubamientos necesarios y bifurca la tarea hija. A continuación, el despachador vigila el descriptor de archivo de tubo `readfd` para detectar las salidas de la tarea y reenvía lo que lee como pa-

quetes DATA a la salida estándar. Cuando el despachador detecta un fin de archivo en readfd, espera a que la tarea hija termine y luego termina. El despachador tiene la siguiente estructura:

```
get_packet(STDIN_FILENO, &comp_id, &task_id, &type, &len, buf);
if (type != START_TASK)
   /* manejar la espera de un error de START_TASK */
else {
   /* crear los entubamientos ... */
   /* bifurcar el hijo para la tarea ... */
   /* cerrar los descriptores innecesarios ... */
   while ((nbytes = read(readfd, buf, MAX_PACK_SIZE)) > 0) {
      if (put_packet(STDOUT_FILENO, comp_id, task_id,
                  DATA, nbytes, buf) == -1)
         break;
   }
}
```

Implemente el despachador NTPVM como se indica. El despachador tiene la siguiente especificación:

- Leer un paquete de la entrada estándar mediante get_packet. Si el paquete no es del tipo START_TASK, terminar después de exhibir un mensaje de error.
- Crear los entubamientos necesarios para comunicarse con una tarea hija.
- Bifurcar un hijo para que ejecute mediante execvp el comando proporcionado en el paquete START_TASK del paso 1. Utilice la función makeargv del programa 1.2 para construir el arreglo de argumentos en el hijo.
- Cerrar todos los descriptores de tubo innecesarios para que el padre pueda detectar un fin de archivo en readfd.
- Esperar salidas del hijo por readfd. Para las pruebas, suponga que el hijo sólo produce texto. El despachador lee la salida de la tarea hijo en readfd, envuelve esta salida en un paquete DATA y envía el paquete a la salida estándar mediante una llamada a put_packet.
- Si se presenta un fin de archivo en readfd, cerrar los descriptores readfd y writefd de la tarea y enviar un paquete DONE con el ID de la tarea a la salida estándar.

La tarea hija redirige su entrada estándar y su salida estándar a los entubamientos apropiados antes de ejecutar mediante execvp el comando solicitado. Asegúrese de cerrar todos los descriptores de archivo innecesarios antes de llamar a execvp.

El despachador deberá utilizar el error estándar sin restricciones para exhibir mensajes informativos sobre lo que está haciendo. Por ejemplo, cuando el despachador reciba algo de un readfd, deberá exhibir información acerca de la tarea que lo originó, el número de bytes leídos y el mensaje que se leyó. También vale la pena dedicar tiempo a diseñar mensajes informativos claros de modo que toda la información pertinente esté disponible con sólo un vistazo. Pruebe el programa utilizando ls -l como comando que se ejecutará con execvp'd.

11.5 Tareas secuenciales

En esta sección se describirá el comportamiento del despachador cuando la tarea hija tiene tanto entradas como salidas. Aunque el despachador controla una sola tarea a la vez, debe vigilar dos descriptores de archivo de entrada. Complete la sección 11.4 antes de iniciar esta parte.

El despachador mantiene información referente a la tarea hija en el arreglo `tasks`. Para que sea más sencillo, nuestra explicación se referirá a los miembros del arreglo `task_t` como `readfd` sin su estructura calificadora. Implemente el arreglo `tasks` como un objeto con funciones de acceso apropiadas. El arreglo `tasks` y sus funciones de acceso deben estar en un archivo aparte del programa principal del despachador. Nos referiremos al arreglo junto con sus funciones de acceso como el objeto `tasks`, y a un elemento individual del arreglo como una entrada del objeto `tasks`.

11.5.1 Versión A: Implementación de un despachador sin hilos

Implemente el despachador NTPVM de una sola tarea a la vez ejecutando el siguiente algoritmo en un ciclo:

- Vigile los descriptores de archivo de entrada para detectar datos. Inicialmente, el despachador tendrá su entrada estándar como su único descriptor de archivo de entrada. Una vez que se inicie una tarea hija, el despachador deberá vigilar también las salidas del hijo. Utilice `select` o `poll` para vigilar los descriptores de archivo `STDIN_FILENO` y `readfd` de forma eficiente.
- Si hay entradas disponibles en la entrada estándar, lea un paquete llamando a la función `get_packet`. Maneje los paquetes de la siguiente manera:
  * Si el paquete es del tipo `START_TASK` y el despachador ya está ejecutando una tarea, deseche el paquete y exhiba un mensaje de error. Si el despachador no tiene actualmente una tarea, cree los entubamientos apropiados, actualice el objeto `tasks` y ramifique un hijo para que ejecute con `execvp` el comando proporcionado en el paquete. Utilice la función `makeargv` del programa 1.2 para construir el arreglo de argumentos.
  * Si el paquete es del tipo `DATA`:
    - Si los ID del paquete no coinciden con los de la tarea en ejecución o si el `end_of_input` de la tarea es `true`, exhiba un mensaje de error y deseche el paquete.
    - En los demás casos, copie la porción de datos en `writefd`.
    - Actualice los miembros `total_packs_recv` y `total_bytes_recv` de la entrada apropiada del objeto `tasks`.
  * Si el paquete es del tipo `DONE`:
    - Si el ID de tarea y el ID de cómputo del paquete no coinciden con los de la tarea en ejecución, exhiba un mensaje de error y deseche el paquete.

- En caso contrario, cierre el descriptor `writefd` si todavía está abierto.
- Establezca el miembro `end_of_input` para esta tarea.

* Si el paquete es de tipo `BROADCAST`, `BARRIER` o `TERMINATE`, exhiba un mensaje de error y deseche el paquete.

- Si hay entradas de la tarea hija disponibles en `readfd`:

  * Lea la información y llame a `put_packet` para enviar un paquete `DATA` por la salida estándar.
  * Actualice `total_packs_sent` y `total_bytes_sent`.

  Para las pruebas, suponga que el hijo sólo produce texto, así que el hijo puede ejecutar con `execvp` comandos estándar como `ls` o `cat`.

- Si ocurre un fin de archivo en la entrada estándar del despachador y hay una tarea activa, cierre el descriptor `writefd`, espere a que la tarea termine y luego haga que el despachador termine.
- Si ocurre un fin de archivo en `readfd`:
  * Cierre los descriptores `readfd` y `writefd` de la tarea.
  * Espere a que la tarea termine.
  * Envíe a la salida estándar un paquete `DONE` con el ID de la tarea.
  * Exhiba a través del error estándar la siguiente información referente a la tarea terminada:

    - El ID del cómputo.
    - El ID de la tarea.
    - El total de bytes enviados por la tarea.
    - El total de paquetes enviados por la tarea.
    - El total de bytes recibidos por la tarea.
    - El total de paquetes recibidos por la tarea.

  * Si ya se recibió un fin de archivo por la entrada estándar, espere a que la tarea termine y luego haga que el despachador termine.
  * Si no se ha recibido un fin de archivo por la entrada estándar del despachador, borre la entrada de `tasks` correspondiente a esta tarea asignando –1 al miembro `comp_id`, regrese y espere la llegada de un paquete `START_TASK`.

Pruebe el programa iniciando tareas para ejecutar diversos comandos `cat` y `ls -l`. Pruebe otros filtros como `sort` para probar el análisis sintáctico de la línea de comandos.

11.5.2 Versión B: Implementación de un despachador con hilos

La versión A de esta parte del proyecto utiliza `select` o `poll` para bloquear el despachador hasta que llegan entradas de la entrada estándar o de la tarea (a través de `readfd`). La versión B utiliza hilos.

Una pregunta obvia es cómo la llamada de sistema `fork` interactúa con los hilos. Recuérdese que `fork` crea una copia del proceso. ¿Cuál hilo comienza a ejecutarse en el hijo de un proceso multihilo? POSIX.1c especifica que el hijo sólo tiene un hilo de ejecución, el hilo que efectuó la llamada `fork`. (Los hilos Sun Solaris utilizan `fork1` para designar un `fork` que produce un proceso hijo de un solo hilo.)

En la figura 11.8 se muestra la estructura de la versión con hilos. El diseño de hilos que se sugiere aquí tiene un *hilo de entrada* que vigila la entrada estándar y escribe en el descriptor `writefd` apropiado. Un *hilo de salida* vigila el descriptor `readfd` para detectar entradas de la tarea hija y envía sus salidas a la salida estándar.

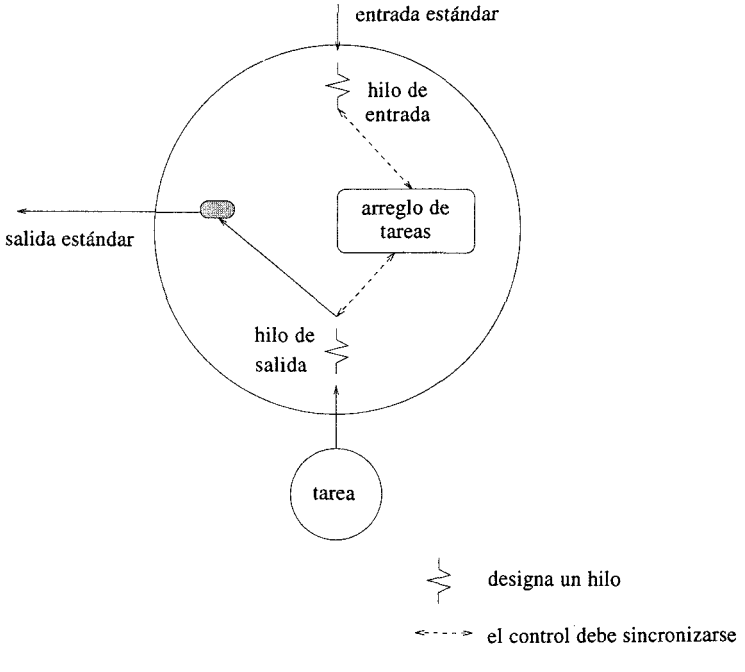


Figura 11.8: Diagrama de un despachador NTPVM con hilos.

Los hilos de entrada y de salida comparten el objeto `tasks` y deben sincronizar su acceso a esta estructura. Un posible mecanismo para sincronizar los hilos es proteger el objeto `tasks` completo con un candado mutex. Esta opción reduce el paralelismo potencial, porque sólo un hilo a la vez puede acceder al objeto `tasks`. Puesto que los candados mutex no cuestan mu-

cho, agregamos un miembro de candado mutex a la definición de `task_t`. Así, cada entrada del objeto `tasks` tiene un miembro que contiene un candado mutex y un miembro que contiene el ID de hilo del hilo de salida para esta tarea. El nuevo `task_t` es:

```
typedef struct {
   int comp_id;                    /* ID de cómputo para la tarea */
   int task_id;                      /* ID de tarea para la tarea */
   int writefd;                   /* contiene fd despachador->hijo */
   int readfd;                    /* contiene fd hijo->despachador */
   int total_packs_sent;
   int total_bytes_sent;
   int total_packs_recv;
   int total_bytes_recv;
   pid_t task_pid;          /* ID de proceso de la tarea bifurcada */
   pthread_t task_tid;  /* ID de hilo del hilo de salida de la tarea */
   int barrier;/*número de barrera o -1 si no está en la barrera */
   int end_of_input;  /* true si ya no hay entradas para la tarea */
   pthread_mutex_t mlock;       /* candado mutex para el elemento */
} task_t;
```

La función `input_thread` vigila la entrada estándar y emprende acciones según las entradas que recibe. El prototipo de `input_thread` es

```
void *input_thread(void *arg);
```

El parámetro `arg` no se utiliza. Escriba una función `input_thread` que ejecute los siguientes pasos en un ciclo hasta que encuentre un fin de archivo en la entrada estándar:

- Lea un paquete de la entrada estándar mediante `get_packet`.
- Si el paquete es del tipo `START_TASK` y no hay una tarea activa:
  - * Inicialice la entrada del objeto `tasks`.
  - * Cree los entubamientos apropiados.
  - * Actualice la entrada del objeto `tasks`.
  - * Bifurque un hijo para que ejecute con `execvp` el comando proporcionado en el paquete. Utilice la función `makeargv` del programa 1.2 para construir el arreglo de argumentos.
  - * Cree el hilo `output_thread` mediante una llamada a `pthread_create`. Pase a `output_thread` el número de elemento de la entrada del objeto `tasks` como argumento.
  - * Actualice todos los campos pertinentes de `tasks`.
- Si el paquete es del tipo `START_TASK` y ya está activa otra tarea, exhiba un mensaje de error y deseche el paquete.

- Si el paquete es del tipo DATA y el ID de cómputo y el ID de tarea coinciden con los ID correspondientes de la tarea que está activa, envíe la porción de información del paquete a la tarea copiándola en `writefd`. Actualice las entradas `total_pack_recv` y `total_bytes_recv` del objeto `tasks`.
- Si el paquete es del tipo DONE y el ID de cómputo y el ID de tarea coinciden con los ID correspondientes de la tarea que está activa, cierre el descriptor `writefd` si todavía está abierto y establezca el miembro `end_of_input` correspondiente a esta tarea.
- Si se recibe cualquier otro tipo de paquete, exhiba un mensaje de error y deseche el paquete.

Después de salir del ciclo, cierre el `writefd` e invoque a `pthread_exit`.

El hilo `output_thread` maneja la salida del `readfd` de una tarea en particular; su prototipo es:

```
void *output_thread(void *arg);
```

El parámetro `arg` es un apuntador a la entrada de `tasks` que corresponde a la tarea en cuestión. Escriba una función `output_thread` que ejecute los siguientes pasos en un ciclo hasta que encuentre un fin de archivo en `readfd`.

- Leer datos de `readfd`. Cuando `input_thread` cree el hilo, le pasará como parámetro el número de entrada del objeto `tasks` correspondiente a la tarea de la cual se encargará este hilo. El hilo determinará el `readfd` buscándolo en la entrada del objeto `tasks`.
- Llamar a `put_packet` para construir un paquete DATA y enviar el paquete a la salida estándar.
- Actualizar los miembros `total_packs_sent` y `total_bytes_sent` de la entrada apropiada del objeto `tasks`.

Después de salir del ciclo debido a un fin de archivo o a un error en `readfd`, el hilo de salida realice lo siguiente:

- Cierre los descriptores `readfd` y `writefd` de la tarea.
- Realice un `wait` con la tarea hija.
- Escriba en la salida estándar un paquete DONE con el ID de cómputo y el ID de tarea apropiados.
- Desactive la entrada de `tasks` asignando −1 al ID de cómputo. (Cuando `input_thread` reutilice la entrada, esperará para este `output_thread`).
- Llame a `pthread_exit`.

El `input_thread` y el `output_thread` comparten los recursos del objeto `tasks`, así que cada hilo debe realizar un `pthread_mutex_lock` con la entrada del objeto `tasks` antes de acceder a los recursos. Escriba un programa principal que inicialice el objeto `tasks` con los candados mutex correspondientes y cree el `input_thread`.

Los recursos de un hilo que ya terminó no se liberan hasta que otro hilo se une a él o el proceso termina. El hilo de entrada se une al hilo de salida antes de reutilizar una entrada `tasks`, a fin de limitar el número de hilos "zombies" durante la ejecución del despachador.

11.6 Tareas concurrentes

Modifique el programa para manejar múltiples cómputos y tareas. Utilice un valor `MAX_TASKS` de 10 en esta parte. Se permite que llegue un paquete `START_TASK` nuevo antes de que se hayan transmitido por completo los datos de tareas anteriores, así que el despachador podría estar vigilando varios descriptores de archivo.

Cuando llegue un paquete `START_TASK` nuevo, encuentre una entrada disponible en el objeto `tasks`, cree un nuevo juego de entubamientos y bifurque un hijo para que ejecute el comando con `execvp`. Suponga que no hay ID de cómputo o de tarea repetidos en los encabezados de los paquetes.

11.6.1 Versión A: Implementación con `select` o `poll`

A fin de atender múltiples tareas, la versión A requiere que el despachador agregue descriptores de archivo nuevos al descriptor de archivo de entrada que se establece cuando se crean tareas nuevas. Modifique la implementación de versión A de la sección 11.5 de modo que permita al despachador manejar hasta `MAX_TASKS` tareas simultáneamente. Considere con cuidado cuándo el despachador debe esperar a que las tareas hijas terminen y en qué punto es válido liberar una entrada del objeto `tasks`.

11.6.2 Versión B: Implementación con hilos

En la figura 11.9 se muestra un diagrama de un despachador NTPVM con hilos. Cuando llega otra solicitud de cálculo, el hilo de entrada crea un nuevo hilo de salida. Puesto que varios hilos de salida escriben en la salida estándar, defina un candado mutex adicional para sincronizar las salidas por la salida estándar del despachador. Otro aspecto que debe considerarse en la implementación concurrente es el proceso de limpieza. En algún momento, el despachador deberá esperar a que las tareas hijas terminen, pues de lo contrario el programa acumulará procesos que han muerto. De forma similar, el despachador deberá unirse con cada `output_thread` antes de liberar la entrada correspondiente en el objeto `tasks`.

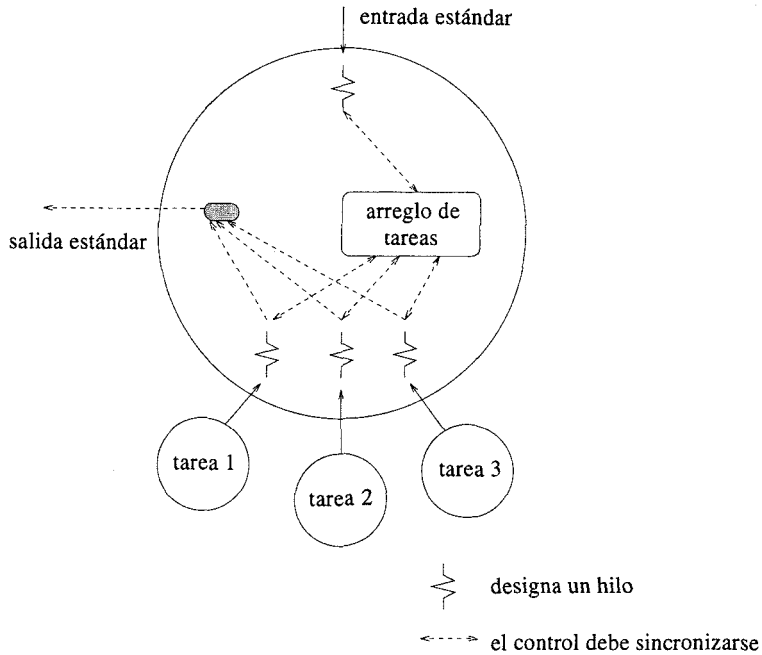


Figura 11.9: Diagrama de un despachador NTPVM con hilos.

11.7 Difusión y barreras

Una vez que el despachador maneje varias tareas simultáneas, implemente el manejo de los paquetes `BROADCAST` y `BARRIER`. Ahora, las tareas hijas tienen que comunicarse con el despachador utilizando el formato de paquetes para que el despachador y sus tareas puedan distinguir la información de control (difundida o de barrera) de la información de datos.

Cuando el despachador recibe una solicitud `BROADCAST` por la entrada estándar, envía una vez más el paquete por los descriptores `writefd` de cada una de las tareas cuyo ID de cómputo coincide con el que viene en el paquete `BROADCAST`. Si el despachador recibe una solicitud `BROADCAST` por uno de los descriptores `readfd`, envía de nuevo el paquete por los descriptores `writefd` de cada tarea cuyo ID de cómputo coincida con el contenido en el paquete `BROADCAST`. Puesto que en una extensión futura las tareas del cómputo pueden residir en otros nodo, el despachador también vuelve a enviar el paquete por su salida estándar.

Cuando el despachador recibe un paquete `BARRIER` de una tarea, asigna al miembro `barrier` correspondiente a esa tarea el número de barrera indicado en los datos del paquete. Una vez que todas las tareas de un cómputo hayan indicado que están esperando en la barrera, el despachador enviará un mensaje `BARRIER` por la salida estándar.

Cuando el despachador lea un mensaje `BARRIER` para ese número de barrera en la entrada estándar, restablecerá el miembro `barrier` a –1 y enviará una señal `SIGUSR1` a todas las tareas del cómputo. El paquete `BARRIER` proveniente de la entrada estándar indica que todas las tareas del cómputo están esperando en la barrera designada y que pueden liberarse. Suponga que el despachador nunca recibe un mensaje `BARRIER` por la entrada estándar antes de haber enviado nuevamente un paquete `BARRIER` correspondiente por la salida estándar.

Implemente la barrera en el lado de las tareas bloqueando la señal `SIGUSR1`, escribiendo un paquete `BARRIER` en la salida estándar y ejecutando luego `sigsuspend` en un ciclo hasta que llegue la señal `SIGUSR1`. El ejemplo 5.20 muestra cómo se hace esto.

Escriba un programa para una tarea muy sencilla para lograr una emisión apropiada y mensajes de obstáculos.

11.8 Terminación y señales

Implemente el manejo de señales de modo que el despachador termine de forma ordenada cuando reciba un `ctrl-c`. Además, agregue código para que el despachador mate una tarea cuando reciba un paquete `TERMINATE` para una tarea específica.

11.9 Lecturas adicionales

El sistema PVM fue desarrollado por el Oak Ridge National Laboratory y la Emory University. El artículo "PVM: A framework for parallel distributed computing", de V. S. Sunderam [87], ofrece un panorama general sobre el desarrollo y la implementación del sistema PVM. Entre otros artículos de interés se encuentran "Visualization and debugging in a heterogeneous environment", de Beguelin *et al.* [7], y "Experiences with network-based concurrent computing on the PVM system", de Geist y Sunderam [29]. La distribución PVM está disponible electrónicamente en `netlib@ornl.gov`.

Parte IV

Comunicación

Capítulo 12

Comunicación cliente-servidor

En el modelo cliente-servidor, los procesos llamados servidores proporcionan servicios a clientes en una red. La mayor parte de las redes de área local cuenta con servidores de archivos que gestionan el espacio en disco común, facilitando el compartimiento de archivos y la preparación de copias de respaldo. Los servicios de red estándar de UNIX, como el correo y la transferencia de archivos, también utilizan el paradigma cliente-servidor. En este capítulo crearemos una biblioteca especial llamada Interfaz Universal de Comunicaciones de Internet (UICI, *Universal Internet Communication Interface*), que simplifica la programación cliente-servidor e ilustra la forma de programar diferentes estrategias cliente-servidor en términos de UICI. A continuación, realizaremos implementaciones de UICI en términos de tres mecanismos de comunicación de UNIX distintos: *sockets*, TLI y STREAMS. Una comparación paralela de las implementaciones aclarará algunas de las diferencias entre los mecanismos. POSIX todavía no estandariza la comunicación en redes, y en este capítulo seguiremos la versión de los *sockets* y TLI descrita en la especificación 1170 (Spec 1170).

Muchas aplicaciones y servicios de red como el correo, la transferencia de archivos (`ftp`), la verificación de autenticidad (Kerberos), el ingreso remoto (`telnet`) y el acceso a sistemas de archivos remotos (NFS) se basan en el modelo cliente-servidor. En este modelo computacional, un cliente solicita un servicio a un servidor y éste proporciona el servicio a los clientes. El servidor puede estar en la misma máquina que el cliente o en una distinta, en cuyo caso la comunicación se realizará a través de una red.

Dos clases de protocolos de comunicación de bajo nivel que manejan el paradigma cliente-servidor son los protocolos orientados a las conexiones y los protocolos sin conexiones. En el

modelo de comunicación orientado a las conexiones, un servidor espera que un cliente solicite una conexión. Una vez que se establece ésta, la comunicación se realiza utilizando asas (descriptores de archivo) y la dirección del servidor no se incluye en los mensajes del usuario. Los protocolos orientados a las conexiones tienen un gasto extra de establecimiento. Una estrategia alternativa es utilizar un protocolo sin conexiones. El cliente envía un solo mensaje a un servidor; el servidor presta el servicio y devuelve una respuesta. En este capítulo haremos hincapié en la comunicación cliente-servidor de bajo nivel orientada a las conexiones.

Los protocolos, tanto aquellos sin conexiones como los orientados a las conexiones, se consideran de bajo nivel en cuanto a que la solicitud de servicio implica una comunicación visible. El programador está consciente de la existencia y la ubicación del servidor, y debe nombrar explícitamente el servidor específico al que desea acceder.

La asignación de nombres a los servidores en un entorno de red es un problema difícil. El método obvio consiste en designar un servidor por su ID de proceso y su ID de nodo. Sin embargo, como el ID de proceso se asigna cronológicamente en el momento en que el proceso inicia su ejecución, es difícil saber con antelación el ID de proceso de un proceso específico en un nodo. La convención de asignación de nombres más común consiste en los servidores números enteros pequeños llamados puertos. Un servidor "escucha" en un puerto bien conocido que se ha designado previamente para un servicio en particular. Al establecer la conexión, el cliente especifica claramente una dirección de nodo y un número de puerto en ese nodo.

En el capítulo 14 estudiaremos los servicios de asignación de nombres de nivel intermedio que están disponibles mediante llamadas a procedimientos remotos (RPC, *remote procedure calls*). Aunque éstas también requieren la especificación clara del nodo, permiten al usuario solicitar un servicio determinado en el nodo por nombre, sin tener que especificar el número de puerto. Uno de los objetivos de los sistemas operativos distribuidos es proporcionar una interfaz unificada entre el usuario y los servicios de sistema en toda la red. En una situación ideal, más de un servidor en la red podría prestar un servicio determinado. Sería cómodo solicitar el servicio por nombre y dejar que el sistema designe el servidor. Desafortunadamente, este tipo de interfaz de alto nivel todavía no forma parte de UNIX.

12.1 Estrategias cliente-servidor

La comunicación cliente-servidor más sencilla se efectúa a través de un solo puerto de comunicación como el que se muestra en la figura 12.1. Si el cliente y el servidor comparten un sistema de archivos y se están ejecutando en la misma máquina, el puerto único puede ser FIFO (primero que entra, primero que sale). En una red, ese puerto puede ser un *socket* o una conexión TLI.

Cuando el servidor se inicia, abre su FIFO bien conocido (o su conexión de *socket* con un puerto bien conocido) y espera solicitudes de los clientes. Cuando un cliente necesita un servicio, abre el FIFO (o una conexión de *socket* con el puerto bien conocido del servidor) y escribe su solicitud. A continuación, el servidor da el servicio. Este enfoque funciona muy bien si sólo hay un cliente y no requiere una respuesta. Si hay más de un cliente, es necesario establecer una convención para enviar el ID de proceso del cliente a fin de poder distinguir las solicitudes

de cada uno de los clientes. (Desde luego, sería conveniente que el mecanismo de identificación impidiera que los clientes falsificaran el ID de otro usuario al pretender hacerse.)

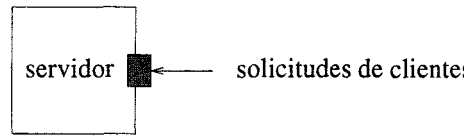


Figura 12.1: Un servidor recibe solicitudes de los clientes por un solo puerto bien conocido.

Un solo puerto no es suficiente en el caso de que el cliente necesita recibir una respuesta del servidor, pues no se cuenta con un mecanismo para que el cliente obtenga la respuesta correcta. Supongamos que el servidor utiliza un solo FIFO para las respuestas. Puesto que los elementos se sacan del FIFO cuando se leen, y no hay nada que impida a un cliente leer la respuesta dirigida a otro, cada uno necesita su propio canal para las respuestas del servidor, como se aprecia en la figura 12.2.

En la implementación FIFO, el servidor puede abrir un FIFO específico para un cliente añadiendo el ID de proceso del cliente al nombre del FIFO de respuesta. El cliente envía una solicitud que incluye su ID (no falsificable) y abre un FIFO convenido para leer la respuesta del servidor. El servidor debe asegurarse de crear semejante FIFO de respuesta con los permisos apropiados de modo que un cliente, aunque lo intente, no pueda robar las respuestas dirigidas a otro.

La implementación con *sockets* cuenta con una llamada `recvfrom` que permite al servidor escuchar en un puerto de *socket* bien conocido para detectar solicitudes. Cada solicitud incluye la identidad de quien la envió. El servidor simplemente utiliza esta identificación en una respuesta `sendto` al cliente. Las llamadas `recvfrom` (recibir de) y `sendto` (enviar a) constituyen la base del protocolo de *sockets sin conexiones*.

Si el cliente y el servidor necesitan seguir interactuando durante el procesamiento de la solicitud, resulta útil contar con un canal de comunicación bidireccional que sea privado pero que no requiera un intercambio de información de ID de proceso en cada mensaje. La figura 12.3 ilustra un mecanismo de transferencia en el que la solicitud inicial del cliente sólo sirve para establecer el canal de comunicación bidireccional que es privado pero no específico para un cliente. El canal es privado porque no es accesible a otros procesos, y no es específico para un cliente porque no se identifica con el ID de proceso del cliente. Los protocolos orientados a conexiones utilizan el mecanismo de transferencia para establecer un canal de comunicación entre el cliente y el servidor. En el resto del capítulo utilizaremos protocolos orientados a las conexiones.

Una vez que un servidor recibe una solicitud y establece un canal de comunicación, puede adoptar varias estrategias distintas para atender las solicitudes. Una posibilidad es que cuando un servidor recibe una solicitud, se dedique íntegramente a atender esa solicitud antes de aceptar solicitudes adicionales. En la figura 12.3 se ilustra la estrategia de *servidor en serie*.

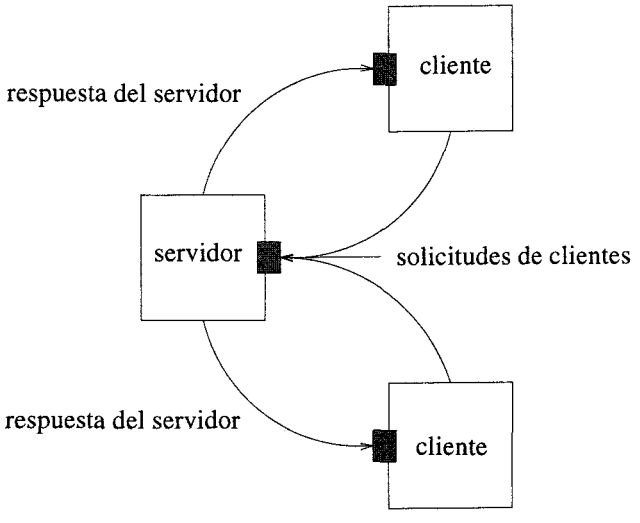


Figura 12.2: En un protocolo sin conexiones, el servidor tiene un puerto bien conocido para las solicitudes de los clientes y puertos específicos para cada cliente para las respuestas.

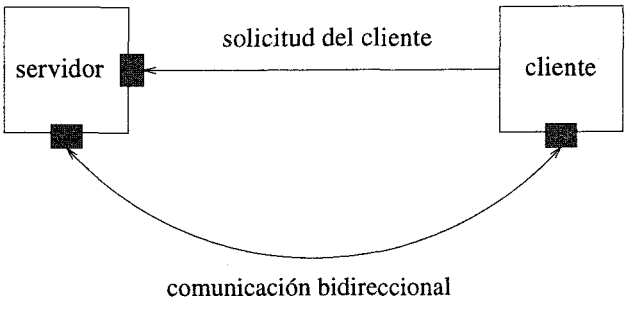


Figura 12.3: El cliente solicita un servicio a un servidor accediendo a un puerto bien conocido. El servidor responde proporcionando un canal de comunicación bidireccional privado.

Ejemplo 12.1

El siguiente pseudocódigo ilustra la estrategia de servidor en serie.

```
for ( ; ; ) {
    escuchar solicitud de cliente
    crear canal privado de comunicación bidireccional
    while(no hay error en el canal de comunicación)
        leer solicitud del cliente
        atender solicitud y responder al cliente
    cerrar el canal de comunicación
}
```

Un servidor ocupado que atiende solicitudes de larga duración, como transferencias de archivos, no puede utilizar la estrategia de servidor en serie porque sólo permite atender una solicitud a la vez. En la estrategia de *servidor padre*, el servidor bifurca un hijo que preste el servicio real al cliente mientras el servidor vuelve a escuchar para detectar solicitudes adicionales. La figura 12.4 es una representación de la estrategia de servidor padre, y es ideal para servicios como la transferencia de archivos que tardan un tiempo relativamente largo e implican gran cantidad de bloqueos.

Ejemplo 12.2

El siguiente pseudocódigo ilustra la estrategia de servidor padre.

```
for ( ; ; ) {
    escuchar solicitud de cliente
    crear canal privado de comunicación bidireccional
    bifurcar un hijo que atienda la solicitud
    cerrar el canal de comunicación
    eliminar "zombies"
}
```

Puesto que el servidor hijo se encarga de prestar el servicio real en la estrategia de servidor padre, el servidor puede aceptar múltiples solicitudes de clientes en rápida sucesión. La estrategia es análoga a la de los anticuados tableros de conmutador telefónico que todavía hay en algunos hoteles. Un cliente llama al número principal del hotel (la solicitud de conexión). El operador del tablero (servidor) contesta la llamada, puentea la conexión a la habitación apropiada (el servidor hijo), se desliga de la conversación y continúa escuchando para detectar llamadas adicionales.

La estrategia de *servidor con hilos* representada en la figura 12.5 es una alternativa con bajo gasto extra que puede utilizarse en lugar de la estrategia de servidor padre. En lugar de bifurcar un hijo para que atienda la solicitud, el servidor crea un hilo en su propio espacio de proceso. Los hilos tienen un gasto extra mucho menor y el enfoque puede ser muy eficiente, sobre todo si la solicitud es pequeña o en ella predomina la E/S. Una desventaja de la estrategia de servidor con hilos es la posible interferencia entre varias solicitudes en virtud del espacio de direcciones compartido. En el caso de servicios en los que predominan los cálculos, los hilos adicionales pueden reducir la eficiencia del hilo principal del servidor o incluso bloquearlo. El diseño del servidor debe tener suficiente paralelismo a nivel de núcleo para tener un buen rendimiento.

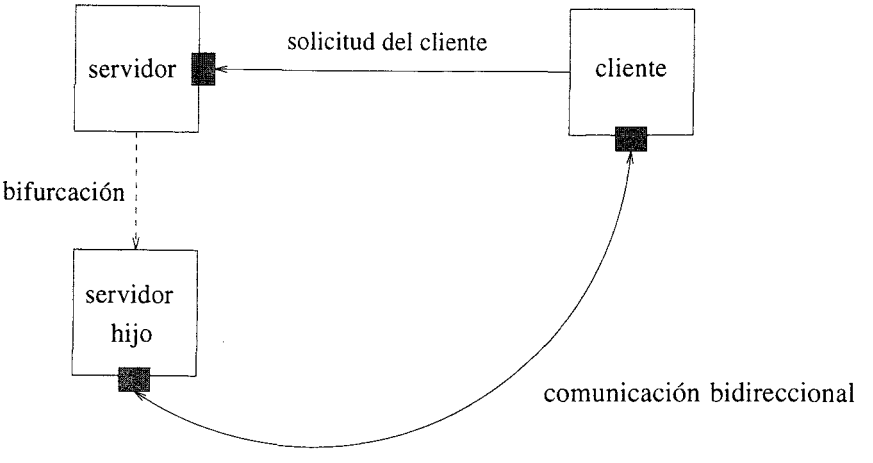


Figura 12.4: La estrategia de servidor padre para la implementación cliente-servidor.

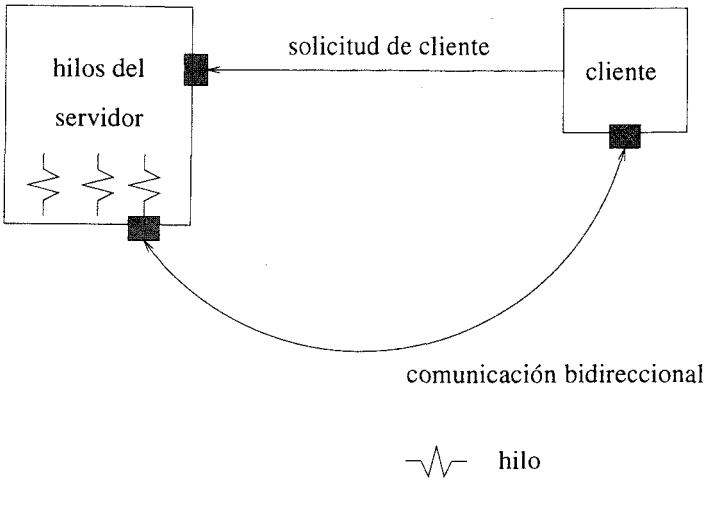


Figura 12.5: La estrategia de servidor con hilos para la implementación cliente-servidor.

12.2 La interfaz universal de comunicaciones de Internet (UICI)

Es posible implementar el paradigma de comunicación cliente-servidor orientado a conexiones empleando *sockets*, TLI o STREAMS. Las tres implementaciones tienen algunos elementos en común. El servidor escucha en un puerto bien conocido para detectar solicitudes de conexión. Por lo regular, el servidor traduce el puerto a una asa (por ejemplo, un descriptor de archivo) apropiada antes de escuchar. Cuando el servidor detecta una solicitud, genera una nueva asa para la comunicación. De aquí en adelante, el cliente y el servidor se pueden comunicar utilizando la asa. En el lado del cliente, éste envía una solicitud de conexión al puerto bien conocido del servidor. La solicitud de conexión devuelve una asa mango para la comunicación.

En esta sección se desarrollará una biblioteca de interfaz universal de comunicaciones de Internet (UICI), misma que se resume en la tabla 12.1. La biblioteca UICI proporciona una interfaz simplificada, independiente del transporte, para la comunicación orientada a conexiones en UNIX. UICI no forma parte de ningún estándar de UNIX, y este capítulo presenta la biblioteca con objeto de aclarar la comunicación cliente-servidor. En el apéndice B se proporcionan implementaciones completas de UICI en términos de *sockets*, TLI y STREAMS. En esta sección presentaremos un ejemplo de cliente-servidor en términos de UICI. En las secciones subsecuentes estudiaremos la implementación subyacente empleando cada uno de los tres mecanismos de comunicación de UNIX.

La función `u_error` devuelve `void`, mientras que el resto de las funciones UICI devuelven `-1` cuando hay un error. Utilice `u_error` para exhibir en el error estándar el mensaje de

Prototipo UICI	Descripción
`int u_open(u_port_t port)`	Abre descriptor de archivo ligado a `port`. Devuelve descriptor de archivo que escucha.
`int u_listen(int fd, char *hostn)`	Escucha solicitudes de conexión por `fd`. Devuelve descriptor de archivo de comunicación.
`int u_connect(u_port_t port, char *the_host)`	Inicia conexión al servidor que está en el puerto `port` y el nodo `the_host`. Devuelve descriptor de archivo de comunicación.
`int u_close(fd)`	Cierra el descriptor de archivo `fd`.
`ssize_t u_read(int fd, char *buf, size_t nbyte)`	Lee hasta `nbyte` bytes de `fd` y los coloca en `buf`. Devuelve número de bytes leídos realmente.
`ssize_t u_write(int fd, char *buf, size_t nbyte)`	Escribe `nbyte` bytes de `buf` a `fd`. Devuelve número de bytes escritos realmente.
`void u_error(char *errmsg)`	Produce `errmsg` seguido de un mensaje de error UICI.
`int u_sync(int fd)`	Actualiza la información de núcleo pertinente después de llamadas a `exec`.

Tabla 12.1: Resumen de llamadas de UICI.

error asociado con un error de UICI, de forma similar al empleo de `perror` en el caso de las llamadas de sistema. El servidor escucha para detectar una conexión (`u_open`, `u_listen`). El cliente solicita la conexión (`u_connect`). Una vez que el cliente y el servidor han establecido una conexión, se comunican por la red utilizando `u_read` y `u_write` de manera análoga a las llamadas de sistema `read` y `write` ordinarias.

La función `copy_from_network_to_file` del programa 12.1 copia datos de un descriptor de archivo de comunicación de red en un archivo y devuelve el número total de bytes copiados. La función `copy_from_network_to_file` supone que ya se estableció una conexión de red con una asa `communfd` y que `filefd` (descriptor del archivo en el que se va a escribir) está abierto. La función llama a `u_error` si `u_read` devuelve un error y a `perror` si `write` devuelve un error.

Programa 12.1: La función `copy_from_network_to_file`.

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include "uici.h"
#define BLKSIZE 1024
int copy_from_network_to_file(int communfd, int filefd)
{
   int bytes_read;
   int bytes_written;
   int bytes_to_write;
   int total_bytes = 0;
   char buf[BLKSIZE];
   char *bufp;

   for ( ; ; ) {
      if ((bytes_read = u_read(communfd, buf, BLKSIZE)) < 0) {
         u_error("Server read error");
         break;
      } else if (bytes_read == 0) {
         fprintf(stderr, "Network end-of-file\n");
         break;
      } else {   /* prever interrupción de la escritura por una señal */
         for (bufp = buf, bytes_to_write = bytes_read;
            bytes_to_write > 0;
            bufp += bytes_written, bytes_to_write -= bytes_written){
            bytes_written = write(filefd, bufp, bytes_to_write);
            if ((bytes_written) == -1 && (errno != EINTR)){
               perror("Server write error");
               break;
            } else if (bytes_written == -1)
               bytes_written = 0;
            total_bytes += bytes_written;
         }
```

```
            if (bytes_written == -1)
               break;
         }
      }
      return total_bytes;
}
```

Programa 12.1

12.2.1 Servidores UICI

Los servidores UICI están organizados como sigue:

- Abren un puerto bien conocido para escuchar (`u_open`). La función `u_open` devuelve un *descriptor de archivo de escuchar*.
- Escuchan en el descriptor de archivo devuelto para detectar una solicitud de conexión (`u_listen`). La función `u_listen` se bloquea hasta que un cliente solicita una conexión y luego devuelve un *descriptor de archivo de comunicación* que se usará como asa para la comunicación privada, bidireccional, entre el cliente y el servidor.
- Se comunican directamente o a través de un sustituto (hijo o hilo) con el cliente empleando el descriptor de archivo para comunicación (`u_read` y `u_write`).
- Cierran el descriptor de archivo para comunicación (`u_close`).

El programa 12.2 es un programa de servidor en serie para copiar archivos de un cliente empleando la biblioteca UICI. El archivo "`uici.h`" define los prototipos UICI necesarios. El servidor requiere un solo argumento de línea de comandos que especifique el número de puerto en el que el servidor escuchará. El servidor obtiene un descriptor de archivo para escuchar para ese puerto mediante `u_open`; a continuación, escucha para detectar solicitudes de los clientes invocando a `u_listen`. La función `u_listen` devuelve un descriptor de archivo para comunicación. Después, el servidor llama a `copy_from_network_to_file` del programa 12.1 para que realice el copiado. Cuando se termina de copiar, el servidor envía a la salida el número de bytes copiados y continúa escuchando.

Programa 12.2: Servidor en serie implementado con UICI.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include "uici.h"
int copy_from_network_to_file(int communfd, int filefd);

/*
 *                    Servidor UICI
 *  Abre un puerto UICI especificado como argumento de línea de comandos
 *  y escucha para detectar solicitudes. Cuando llega una solicitud,
```

```
 *  usa el descriptor de archivo para comunicación provisto para leer
 *  de la conexión UICI y hacer eco en la salida estándar hasta que
 *  se cancela la conexión. Luego el servidor sigue escuchando
 *  para detectar solicitudes adicionales.
 */
void main(int argc, char *argv[])
{
   u_port_t portnumber;
   int listenfd;
   int communfd;
   char client[MAX_CANON];
   int bytes_copied;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s port\n", argv[0]);
      exit(1);
   }
   portnumber = (u_port_t) atoi(argv[1]);
   if ((listenfd = u_open(portnumber)) == -1) {
      u_error("Unable to establish a port connection");
      exit(1);
   }
   while ((communfd = u_listen(listenfd, client)) != -1) {
      fprintf(stderr, "A connection has been made to %s\n",client);
      bytes_copied = copy_from_network_to_file(communfd, STDOUT_FILENO);
      fprintf(stderr, "Bytes transferred = %d\n", bytes_copied);
      close(communfd);
   }
   exit(0);
}
```

Programa 12.2

En la estrategia de servidor en serie del programa 12.2, el servidor copia el archivo completo antes de aceptar otra solicitud de cliente. Esa estrategia puede producir largos retardos para otros clientes, lo que sugiere que la estrategia de servidor padre es más apropiada. En la estrategia de servidor padre las solicitudes adicionales de los clientes se retardan sólo mientras el servidor bifurca un hijo que procese la solicitud. El programa 12.3 implementa la estrategia de servidor padre. Si el padre no cierra `communfd` (el descriptor de archivo para comunicación), el hijo no podrá detectar un fin de archivo en su conexión con la red.

Programa 12.3: Programa servidor que bifurca un hijo para que maneje la comunicación.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
```

```
#include <sys/wait.h>
#include "uici.h"

int copy_from_network_to_file(int communfd, int filefd);

/*
 *                    Servidor UICI
 *  Abre un puerto UICI especificado como argumento de línea de comandos
 *  y escucha para detectar solicitudes. Cuando llega una solicitud,
 *  bifurca un hijo para manejar la comunicación y
 *   escucha otra vez.
 */
void main(int argc, char *argv[])
{
   u_port_t portnumber;
   int listenfd;
   int communfd;
   char client[MAX_CANON];
   int bytes_copied;
   int child;

   if (argc != 2) {
      fprintf(stderr, "Usage: %s port\n", argv[0]);
      exit(1);
   }

   portnumber = (u_port_t) atoi(argv[1]);
   if ((listenfd = u_open(portnumber)) == -1) {
      u_error("Unable to establish a port connection");
      exit(1);
   }

   while ((communfd = u_listen(listenfd, client)) != -1) {
      fprintf(stderr, "[%ld]: A connection has been made to %s\n",
              (long) getpid(), client);
      if ((child = fork()) == -1) {
         fprintf(stderr, "Could not fork a child\n");
         break;
      }

      if (child == 0) {                              /* código del hijo */
         close(listenfd);
         fprintf(stderr, "[%ld]: A connection has been made to %s\n",
                         (long) getpid(), client);
         bytes_copied =
             copy_from_network_to_file(communfd, STDOUT_FILENO);
         close(communfd);
         fprintf(stderr, "[%ld]:Bytes transferred = %d\n",
                         (long) getpid(), bytes_copied);
```

```
            exit(0);

        } else {                                    /* código del padre */
            close(communfd);
            while (waitpid(-1, NULL, WNOHANG) > 0)
                ;
        }
    }
    exit(0);
}
```

Programa 12.3

12.2.2 Clientes UICI

Los clientes de UICI están organizados como sigue:

- Se conectan con un nodo y un puerto especificados (`u_connect`). La solicitud de conexión devuelve el descriptor de archivo para comunicación.
- Se comunican con el servidor (`u_read` y `u_write`) utilizando el descriptor de archivo para comunicación.
- Cierran el descriptor de archivo para comunicación.

El programa 12.4 muestra el lado del cliente en el copiado de archivos. El cliente establece una conexión con un puerto especificado en un nodo determinado llamando a `u_connect`. La función `u_connect` devuelve el descriptor de archivo para comunicación. El cliente lee el archivo de su entrada estándar, transfiere el archivo y termina.

Programa 12.4: Cliente implementado con UICI.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include "uici.h"

#define BLKSIZE 1024

/*
 *                    Cliente de UICI
 *  Solicitar una conexión UICI al nodo y puerto especificados en los
 *  argumentos de línea de comandos. Leer de la entrada estándar y
 *  escribir en el descriptor de archivo para comunicación UICI hasta
 *  el fin de archivo.
 */
```

```
void main(int argc, char *argv[])
{
   u_port_t portnumber;
   int communfd;
   ssize_t bytesread;
   char buf[BLKSIZE];

   if (argc != 3) {
      fprintf(stderr, "Usage: %s host port\n", argv[0]);
      exit(1);
   }
   portnumber = (u_port_t)atoi(argv[2]);
   if ((communfd = u_connect(portnumber, argv[1])) < 0) {
      u_error("Unable to establish an Internet connection");
      exit(1);
   }
   fprintf(stderr, "A connection has been made to %s\n",argv[1]);
   for ( ; ; ) {
      if ((bytesread = read(STDIN_FILENO, buf, BLKSIZE)) < 0) {
         perror("Client read error");
         break;
      } else if (bytesread == 0) {
         fprintf(stderr, "Client detected end-of-file on input\n");
         break;
      } else if (bytesread !=
                  u_write(communfd, buf, (size_t)bytesread)) {
         u_error("Client write_error");
         break;
      }
   }
   u_close(communfd);
   exit(0);
}
```

Programa 12.4

Ejemplo 12.3

Si el código ejecutable para el servidor dado en el programa 12.2 se llama `uici_server`, *el siguiente comando hará que el servidor escuche en el puerto* `8652` *para detectar solicitudes de conexión de los clientes.*

```
uici_server 8652
```

Ejemplo 12.4

Si el código ejecutable para el cliente del programa 12.4 se llama `uici_client` *y el servidor está en un* nodo *llamado* `vip`, *el siguiente comando hace que el cliente solicite una conexión.*

```
uici_client vip 8652
```

UICI	Sockets	TLI	STREAMS
u_open	socket bind listen	t_open t_bind	create pipe meterconnld fattach
u_listen	accept	t_alloc t_listen t_bind t_accept	ioctl de I_RECVFD
u_connect	socket connect	t_open t_bind t_alloc t_connect	open
u_read	read	t_rcv	read
u_write	write	t_snd	write
u_sync		t_sync	

Tabla 12.2: Resumen del protocolo de conexión UICI y sus implementaciones subyacentes.

Ejercicio 12.1

Inicie el servidor y el cliente en distintas ventanas de diferentes máquinas utilizando los comandos de los ejemplos 12.3 y 12.4. Una vez establecida la conexión, introduzca texto para el cliente y vea cómo el servidor lo envía a la salida.

12.2.3 Implementación de UICI

En la tabla 12.2 se muestra los pasos necesarios para implementar cada función UICI en términos de tres distintos paradigmas de comunicación: *sockets*, TLI o STREAMS. Las implementaciones con *sockets* y TLI contemplan la comunicación en red. La implementación con STREAMS sólo funciona si el cliente y el servidor están en la misma máquina.

Todos los mecanismos subyacentes (*sockets*, TLI y STREAMS) contemplan la posibilidad de E/S no bloqueadora, pero la interfaz UICI sólo proporciona E/S bloqueadora; es decir, una llamada `u_read` o `u_write` hace que el invocador se bloquee. Una `u_read` bloquea hasta que la información está disponible en la conexión con la red.

Lo que el bloqueo significa para `u_write` es menos obvio. En el contexto actual significa que `u_write` regresa una vez que la salida se ha transferido a un *buffer* utilizado por el mecanismo de transporte. Las escrituras también pueden bloquear si, a causa de problemas de entrega en las capas inferiores del protocolo, todos los *buffers* para los protocolos de red están llenos. Por fortuna, los aspectos de bloqueo y almacenamiento temporal son transparentes para la mayor parte de las aplicaciones.

En la siguiente sección presentaremos un breve repaso de las capas de red y de la comunicación en la Internet. Las siguientes secciones del capítulo ofrecen un panorama general de cada uno de los mecanismos subyacentes y desarrollan implementaciones de las funciones `u_open`, `u_listen` y `u_connect` en términos de estos mecanismos. En la sección 12.4 presentaremos las llamadas de *sockets* y veremos una implementación de UICI empleando *sockets*. En la sección 12.5 presentaremos algunos de los detalles técnicos de TLI y estudiaremos una implementación de UICI utilizando TLI. La sección 12.6 es una introducción a STREAMS y en la sección 12.7 se proporciona una implementación STREAMS de UICI. En el apéndice B se presentan implementaciones completas de UICI para *sockets*, TLI y STREAMS. La interfaz UICI se diseñó de modo que parezca una versión simplificada de TLI o de *sockets*. En vista de los problemas para detectar errores, es difícil lograr que dicha interfaz sea segura para usarse con hilos. En la sección 12.8 veremos una versión segura respecto de los hilos de la interfaz UICI.

12.3 Comunicación en red

La Organización Internacional de Normas (ISO, por sus siglas en inglés) tiene un estándar para el diseño de redes llamado modelo de referencia de interconexión de sistemas abiertos (OSI, *Open Systems Interconnection*). En la figura 12.6 se muestran las siete capas de protocolo del modelo OSI. Cada capa consta de un conjunto de funciones que se encargan de un aspecto específico de la comunicación en red. Las funciones de una capa se comunican sólo con las capas que están inmediatamente arriba y abajo. Las capas más bajas del modelo OSI se ocupan de aspectos íntimamente relacionados con el *hardware*, mientras que las capas superiores se ocupan de la interfaz con el usuario.

En la comunicación par a par con protocolos de capas, cada capa de un nodo parece comunicarse con la misma capa del otro nodo. Esta perspectiva lógica de la comunicación simplifica y aísla la implementación de las funciones de una capa. Por ejemplo, la perspectiva lógica de la comunicación en la capa de transporte en la figura 12.6 podría ser una corriente de bytes libre de errores. El mecanismo real es que la capa de transporte pasa sus datos a la capa que está abajo. Esa capa realiza su función y pasa la información a la capa inmediata inferior. Finalmente, la información fluye por la red física hacia el otro nodo, donde se transfiere hacia arriba a través de capas sucesivas, hasta la capa de transporte de ese nodo.

La *capa física* y la *capa de enlace de datos* se ocupan de la transmisión de datos de punto a punto. Ethernet es una implementación común de bajo costo de estas capas. Cada nodo de la red cuenta con un adaptador de Ethernet en *hardware* conectado con un enlace de comunicación que consiste en cable coaxial o alambre de par trenzado. El nodo se identifica mediante una dirección de Ethernet única de seis bytes que está alambrada permanentemente en el *hardware* del adaptador. Otras posibilidades de uso común para estas dos capas incluyen *token ring*, *token bus*, ISDN, ATM y FDDI.

La *capa de red* se encarga del direccionamiento de red y del ruteo a través de los puentes y ruteadores que conectan las redes entre sí. El protocolo de capa de red más común que se utiliza en los sistemas UNIX se llama IP, el Protocolo de Internet. Cada nodo tiene una o más

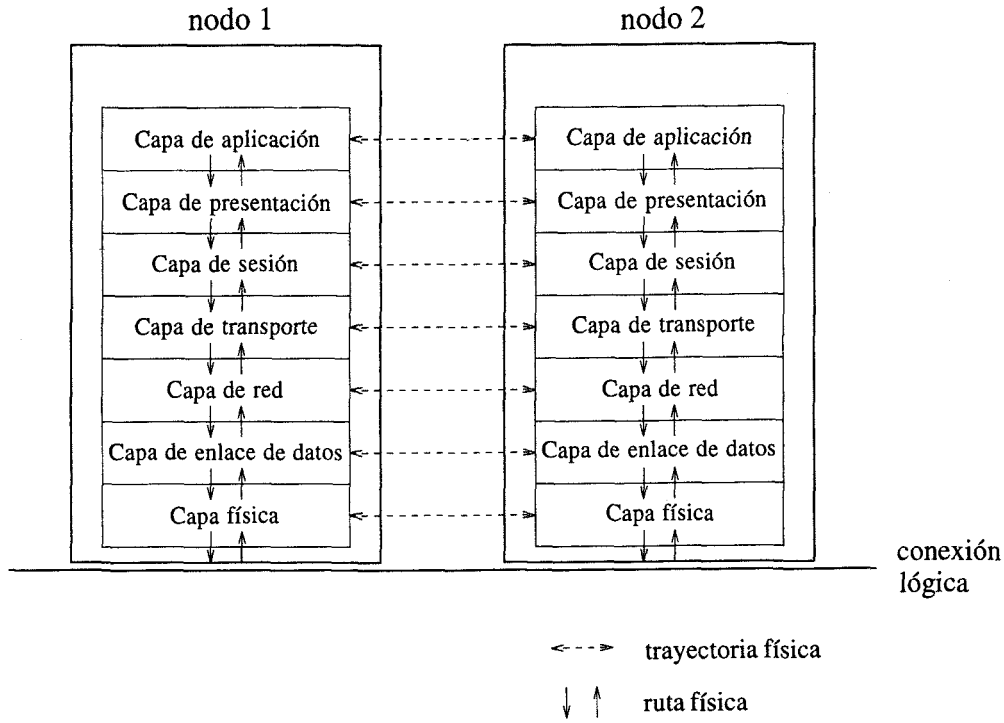


Figura 12.6: Comunicación par a par en el modelo OSI.

direcciones de IP, cada una de las cuales comprende cuatro bytes. Se acostumbra hacer referencia a los cuatro bytes de la dirección de IP representándolos en decimal con puntos decimales entre los bytes (por ejemplo, `198.86.40.81`).

Aunque el *software* de IP sólo maneja direcciones numéricas de IP, los usuarios suelen referirse a una máquina por nombre (por ejemplo, `sunsite.unc.edu`). La conversión de nombre a dirección se realiza con `gethostbyname`, y de dirección a nombre, con `gethostbyaddr`. Por lo regular existe una correspondencia entre la primera parte de la dirección de IP (1, 2 o 3 bytes), llamada dominio de Internet, y la parte final del nombre de la máquina, pero los detalles exactos de la correspondencia son complicados y carecen de importancia en nuestro contexto.

La *capa de transporte* se encarga de la comunicación de extremo a extremo entre nodos. Los dos principales protocolos utilizados en esta capa son TCP (protocolo de control de transmisión) y UDP (protocolo de datagrama de usuario). Las redes UNIX utilizan tanto TCP como UDP. UDP es un protocolo sin conexiones que no ofrece garantía de entrega. El programa `tftp` (protocolo de transferencia de archivos trivial) por lo regular se implementa mediante UDP. TCP es un protocolo confiable, orientado a conexiones. El programa `ftp` (protocolo de transferencia de archivos) casi siempre se implementa con TCP.

Es posible que más de un usuario a la vez estén utilizando TCP o UDP entre cierto par de máquinas. A fin de distinguir entre los diversos procesos que podrían estar comunicándose, TCP y UDP emplean enteros de 16 *bits* llamados puertos. Algunos de estos números de puerto se han asignado permanentemente a aplicaciones específicas y se denominan *direcciones bien conocidas*. Por ejemplo, `ftp` siempre utiliza el puerto 21, `telnet` emplea el 23, `tftp` usa el 69 y `finger` utiliza el 79. Es recomendable que los procesos de usuario elijan números de puerto por arriba del 7000 para no interferir los servicios de sistema y X.

La *capa de sesión* contiene interfaces con la capa de transporte. En este capítulo veremos dos interfaces estándar llamadas *sockets* y TLI (interfaz con la capa de transporte; en inglés, *transport layer interface*). La *capa de presentación* y la *capa de aplicación* constan de utilerías generales y programas de aplicación. La capa de presentación puede ocuparse de la compresión y el cifrado.

Una desventaja de utilizar el modelo OSI es que muchas redes ya estaban establecidas para cuando se aceptó el modelo de referencia OSI; por tanto, la organización de muchas redes no se ajusta con exactitud al modelo. De cualquier manera, el modelo es un valioso marco de referencia conceptual para comparar protocolos.

12.4 Implementación de UICI con *sockets*

La primera interfaz de *sockets* se originó con 4.1cBSD UNIX a principios de los años ochenta. La especificación 1170 incluye la versión 4.3BSD, y es probable que POSIX estandarice los *sockets* en un futuro no muy lejano. En la actualidad es común implementar *sockets* basándose en STREAMS en los sistemas System V.

Si desea utilizar las llamadas de *sockets*, compile los programas con las opciones de biblioteca `-lsocket` y `-lnsl`. Todas las llamadas de sistema de *sockets* devuelven –1 si no tienen éxito y establecen la variable externa `errno`.

En la tabla 12.2 de la página 444 se resumen las llamadas de *sockets* requeridas para establecer la comunicación. El servidor crea una asa (`socket`), lo asocia con una posición física en la red (`bind`) y establece el tamaño de la cola para las solicitudes pendientes (`listen`). A continuación, el servidor escucha para detectar solicitudes de los clientes (`accept`).

El cliente también crea una asa (`socket`), la cual luego asocia con la posición del servidor en la red (`connect`). Las asas del servidor y del cliente, también llamados *extremos de transmisión*, son descriptores de archivo. Una vez que el cliente y el servidor han establecido una conexión, pueden comunicarse mediante llamadas `read` y `write` ordinarias.

La llamada `socket` crea un extremo para la comunicación y devuelve un descriptor de archivo.

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Spec 1170

El parámetro `domain` (dominio) selecciona la familia de protocolos que se utilizará. La especificación 1170 exige que se manejen los dominios `AF_UNIX` y `AF_INET`. El primero sólo puede utilizarse entre procesos en un mismo sistema UNIX, mientras que el segundo es para la Internet y permite la comunicación entre nodos remotos.

Si se asigna a `type` (tipo) el valor `SOCK_STREAM`, se podrán usar flujos de bytes ordenados, confiables, bidireccionales y orientados a conexiones, los cuales por lo regular se implementan con TCP. Si el valor de `type` es `SOCK_DGRAM`, se contará con una comunicación sin conexiones empleando mensajes no confiables de longitud fija; esta comunicación por lo regular se implementa con UDP.

El parámetro `protocol` especifica el protocolo que se utilizará. Sin embargo, lo normal es que sólo haya un protocolo disponible para cada valor de `type` (por ejemplo, TCP con `SOCK_STREAM` y UDP con `SOCK_DGRAM`), así que lo usual es utilizar 0 como valor de `protocol`.

Ejemplo 12.5

El siguiente segmento de código establece un extremo de transmisión de socket *para la comunicación por Internet empleando un protocolo orientado a conexiones.*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
int sock;

if ((sock = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    perror("Could not create socket");
```

La llamada `bind` asocia un extremo de transmisión de *socket* o una asa con una conexión física de red en particular. Los protocolos del dominio de Internet especifican la conexión física mediante un número de puerto, mientras que los protocolos del dominio de UNIX lo hacen mediante un nombre de trayectoria.

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(int s, const struct sockaddr *address,
         size_t address_len);
```

Spec 1170

Aquí, `s` es el descriptor de archivo devuelto por la llamada `socket` y `address_len` es el número de bytes que tiene la estructura `*address`. Dicha estructura contiene un nombre de familia e información específica para el protocolo. El dominio de Internet utiliza `struct`

`sockaddr_in` en lugar de `struct sockaddr`. El tipo `struct sockaddr_in` está definido por

```
struct sockaddr_in {
     short   sin_family;
     u_short sin_port;
     struct  in_addr sin_addr;
     char    sin_zero[8];
};
```

El miembro `sin_family` es `AF_INET` y `sin_port` es el número de puerto. Podemos asignar a `sin_addr` el valor `INADDR_ANY` para permitir la comunicación desde cualquier nodo. El arreglo `sin_zero` rellena la estructura para que tenga el mismo tamaño que `struct sockaddr`.

Un servidor puede procesar varias solicitudes de comunicación simultáneas dirigidas al mismo número de puerto bien conocido bifurcando un hijo o iniciando un hilo nuevo para manejar cada comunicación. Mientras el servidor está procesando una solicitud de comunicación para un cliente, no puede responder a otras solicitudes de cliente. La llamada `listen` especifica cuántas solicitudes de cliente pendientes pueden acumularse antes de que el servidor rechace una conexión. El cliente recibirá un error `ECONNREFUSED` si el servidor rechaza su solicitud de conexión.

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int listen(int s, int backlog);
```

Spec 1170

El valor `s` es el descriptor devuelto por `socket` y `backlog` es el número de solicitudes de cliente pendientes permitidas.

La combinación de `open`, `bind` y `listen` establece una asa para que el servidor vigile las solicitudes de comunicación de un puerto bien conocido. El programa 12.5 es una implementación de `u_open` en términos de las llamadas de *sockets*.

Si se intenta escribir en un entubamiento o *socket* que ningún proceso abrió para lectura, la llamada `write` generará una señal `SIGPIPE` además de devolver un error y asignar `EPIPE` a `errno`. Puesto que la acción por omisión (predeterminada o dada por *default*) de `SIGPIPE` es terminar el proceso, esta señal impedirá una terminación ordenada cuando el nodo remoto cierre la conexión (a menos que se utilice un controlador de señales definido por el usuario). Por esta razón, la implementación de UICI con *sockets* ignora la señal si está activo el controlador por omisión.

El miembro `sin_port` del segundo argumento de `bind` representa el número de puerto empleando el ordenamiento de bytes de la red. Las máquinas que utilicen un ordenamiento de bytes distinto deberán efectuar una conversión. Se puede usar la macro `htons` (network short o nodo a red corto) para realizar la conversión de números de puerto; es recomendable utilizarla incluso cuando no es necesaria, a fin de mantener la transportabilidad.

Programa 12.5: Implementación de la función `u_open` de UICI con *sockets*.

```
int u_open(u_port_t port)
{
   int sock;
   struct sockaddr_in server;

   if ( (u_ignore_sigpipe() != 0) ||
        ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) )
      return -1;

   server.sin_family = AF_INET;
   server.sin_addr.s_addr = INADDR_ANY;
   server.sin_port = htons((short)port);

   if ( (bind(sock, (struct sockaddr *)&server, sizeof(server)) < 0) ||
        (listen(sock, MAXBACKLOG) < 0) )
      return -1;
   return sock;
}
```

Programa 12.5

Una vez que el servidor asocia un manejador a la conexión física, espera que un cliente inicie la conexión con la llamada `accept`.

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int accept(int s, struct sockaddr *address, int *address_len);
```

Spec 1170

Los parámetros son similares a los de `bind`, excepto que `accept` llena `*address` con información acerca del cliente que establece la conexión. En particular, el miembro `sin_addr` de la estructura `struct sockaddr_in` contiene un miembro, `s_addr`, para guardar en él la dirección de Internet del cliente.

El valor del parámetro `*address_len` de `accept` especifica el tamaño del *buffer* al que `address` apunta. Antes de emitir la llamada, llene este parámetro con el tamaño de la estructura `address`. Después de la llamada, `*address_len` contiene el número de bytes que la llamada `accept` realmente colocó en el *buffer*. La llamada devuelve un descriptor de archivo para comunicarse con el cliente. Ahora, el servidor puede bifurcar un hijo que se encargue de la comunicación y seguir vigilando el descriptor de archivo original.

Convierta esta dirección en un nombre invocando a `gethostbyaddr`.

SINOPSIS

```
#include <netdb.h>

struct hostent *gethostbyaddr(const void *addr, size_t len,
        int type);
```

Spec 1170

La estructura `struct hostent` incluye un miembro, `h_name`, que es un apuntador al nombre oficial del nodo. Si hay un error, `gethostbyaddr` devuelve `NULL` y establece el entero externo `h_errno`. La función `gethostbyaddr` no es segura respecto de los hilos, por lo que es preferible utilizar `gethostbyaddr_r`. (Véase la sección 12.8.)

El programa 12.6 es una implementación de `u_listen`. La llamada de *socket* `accept` espera una solicitud de conexión y devuelve un descriptor de archivo para comunicación. Si `accept` es interrumpida por una señal, devuelve un –1 y asigna el valor `EINTR` a `errno`. En este caso, la función `u_listen` reiniciará la llamada `accept`.

Programa 12.6: Implementación de la función `u_listen` de UICI con *sockets*.

```
int u_listen(int fd, char *hostn)
{
   struct sockaddr_in net_client;
   int len = sizeof(struct sockaddr);
   int retval;
   struct hostent *hostptr;

   while ( ((retval =
           accept(fd, (struct sockaddr *)(&net_client), &len)) == -1) &&
          (errno == EINTR) )
      ;
   if (retval == -1)
      return retval;
   hostptr =
      gethostbyaddr((char *)&(net_client.sin_addr.s_addr), 4, AF_INET);
   if (hostptr == NULL)
      strcpy(hostn, "unknown");
   else
      strcpy(hostn, (*hostptr).h_name);
   return retval;
}
```

Programa 12.6

El código del cliente para establecer una conexión con *sockets* es más sencillo. El cliente invoca a `socket` para establecer un extremo de transmisión y luego utiliza `connect` para establecer un enlace con el puerto bien conocido del servidor remoto.

SINOPSIS

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int s, struct sockaddr *address,
            size_t address_len);
```

Spec 1170

La estructura `struct sockaddr_in` se llena igual que con `bind`.

El programa 12.7 es una implementación de `u_connect` con *sockets*. Dada la posibilidad de que una solicitud de conexión sea para un nodo distinto, la especificación de la dirección es un tanto complicada. Convierta la cadena del nombre de nodo en una dirección de Internet apropiada invocando a `gethostbyname`.

SINOPSIS

```
#include <netdb.h>

struct hostent *gethostbyname(const char *name);
```

Spec 1170

La estructura `struct hostent` incluye dos miembros de interés que se llenan con `gethostbyname`. El miembro `h_addr_list` es un arreglo de apuntadores a direcciones de red utilizadas por este nodo. Utilice la primera, `h_addr_list[0]`. El miembro entero, `h_length`, se llena con el número de bytes de la dirección. Si hay un error, `gethostbyname` devuelve `NULL` y establece el entero externo `h_error`. Se dispone de macros para determinar el error. `gethostbyname` no es segura respecto de los hilos, por lo que se recomienda utilizar `gethostbyname_r` en aplicaciones con hilos. (Véase la sección 12.8).

El cliente crea un *socket* y emite la solicitud de conexión. La llamada `connect` puede ser interrumpida por una señal, en cuyo caso el ciclo reiniciará la llamada. El programa no utiliza `strncpy` para copiar la dirección del servidor en `server.sin_addr`, pues el origen puede tener un byte cero incluido.

Programa 12.7: Implementación de la función `u_connect` de UICI con *sockets*.

```
int u_connect(u_port_t port, char *hostn)
{
   struct sockaddr_in server;
   struct hostent *hp;
   int sock;
   int retval;

   if ( (u_ignore_sigpipe() != 0) ||
        !(hp = gethostbyname(hostn)) ||
        ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) )
```

```
        return -1;

    memcpy((char *)&server.sin_addr, hp->h_addr_list[0], hp->h_length);

    server.sin_port = htons((short)port);
    server.sin_family = AF_INET;

    while ( ((retval =
        connect(sock, (struct sockaddr *)&server, sizeof(server))) == -1)
         && (errno == EINTR) )
        ;
    if (retval == -1) {
         close(sock);
         return -1;
    }
    return sock;
}
```

Programa 12.7

Una vez que el cliente y el servidor establecen una conexión, pueden intercambiar información empleando `u_read` y `u_write` con sus descriptores de archivo para comunicación. Los *sockets* también apoyan la comunicación sin conexiones, la transferencia de descriptores de archivo y los datos fuera de banda, pero estas características no forman parte de UICI. En el apéndice B se proporciona una implementación completa de UICI con *sockets*.

12.5 Interfaz de la capa de transporte (TLI)

TLI es una alternativa a los *sockets* para las comunicaciones cliente-servidor en una red. TLI, siglas de *Transport Layer Interface* (interfaz de la capa de transporte), se introdujo en la versión 3 de System V en 1986. En System V, TLI se implementa con STREAMS como base. Compile los programas con `-lnsl` para tener acceso a la biblioteca de TLI.

Los programas con TLI son más complicados que los que utilizan *sockets* porque deben reservar espacio para las estructuras de datos de comunicación mediante la llamada de propósito general `t_alloc`. A fin de evitar fugas de memoria en los programas de ejecución larga, asegúrese de liberar estas estructuras mediante `t_free`. Otra complicación con TLI es que las estructuras de datos de TLI están en el área del usuario y no quedan accesibles después de una llamada `exec`. Utilice la función `t_sync` para copiar la información necesaria del área de usuario.

El manejo de errores también es más complicado en TLI. TLI no establece directamente la variable externa `errno` cuando se topa con un error. En vez de ello, TLI cuenta con su propia variable externa `t_errno` para especificar errores y una función `t_look` para determinar la situación del canal de comunicaciones. Si el problema se debió a un error de sistema, TLI asigna `TSYSERR` a `t_errno`; en este caso, `errno` contiene información importante.

La función `t_open` es el primer paso para establecer la comunicación, ya sea que la inicie el cliente o el servidor. Esta función establece una asa o extremo de transmisión para la comunicación.

SINOPSIS

```
#include <xti.h>
#include <fcntl.h>

int t_open(const char *path, int oflag, struct t_info *info);
```

Spec 1170

El parámetro `path` indica el protocolo de comunicación subyacente, por ejemplo, `/dev/tcp` (orientado a conexiones) o `/dev/udp` (sin conexiones). El parámetro `oflag` es similar a la bandera que se utiliza con la llamada de sistema `open` usual y por lo regular es `O_RDWR`. Si la función se ejecuta con éxito, el parámetro `*info` contiene información acerca del protocolo subyacente. Si este parámetro es el apuntador `NULL`, `t_open` no devuelve información sobre el protocolo subyacente. Si hay éxito, `t_open` devuelve un descriptor de archivo para comunicación; si no, devuelve –1 y establece `t_errno`.

La función `t_bind` es similar a la función `bind` para *sockets*; asocia a una conexión física el descriptor de archivo creado por `t_open`.

SINOPSIS

```
#include <xti.h>

int t_bind(int fildes, const struct t_bind *req,
           struct t_bind *ret);
```

Spec 1170

El parámetro `fildes` es la asa devuelto por `t_open`. Llene la estructura `req` con información acerca del puerto de red del servidor antes de invocar a `t_bind`. Si `ret` no es un apuntador `NULL`, la función `t_bind` llena `ret` con información referente al puerto siempre que tiene éxito. Esta función devuelve 0 si tiene éxito y devuelve `-1` y establece `t_errno` si ocurre un error.

La estructura `struct t_bind` contiene los siguientes miembros:

```
struct netbuf addr;
unsigned qlen;
```

Aquí, `addr` es del tipo `struct netbuf`, que en el caso del dominio de Internet es igual a la estructura `struct sockaddr_in` que explicamos para los *sockets* en la sección 12.4. El miembro `qlen` es un entero sin signo que especifica el número de conexiones pendientes y es similar al valor establecido para `listen` en el caso de los *sockets*. El manejo de las solicitudes de conexión acumuladas que todavía no se han atendido es complicado en TLI, así que la implementación de UICI no permite la acumulación de solicitudes.

El programa 12.8 es una implementación TLI de `u_open` en términos de `t_open` y `t_bind`. La mayor parte del código de `u_open` se ocupa de establecer las estructuras de dirección apropiadas para la conexión de Internet. La función `t_open` crea el descriptor de archivo para la conexión TLI. La función `t_bind` asocia ese descriptor de archivo al puerto de red apropiado para comunicarse con el mundo exterior.

Programa 12.8: Implementación TLI de la función `u_open` de UICI.

```
int u_open(u_port_t port)
{
   int fd;
   struct t_info info;
   struct t_bind req;
   struct sockaddr_in server;

   /* Crear un extremo TLI */
   fd = t_open("/dev/tcp", O_RDWR, &info);
   if (fd < 0)
      return -1;

   /* Especificar información del servidor con comodines */
   memset(&server, 0, (size_t)sizeof(server));
   server.sin_family = AF_INET;
   server.sin_addr.s_addr = INADDR_ANY;
   server.sin_port = htons((short)port);

   req.addr.maxlen = sizeof(server);
   req.addr.len = sizeof(server);
   req.addr.buf = (char *)&server;
   req.qlen = 1;

   /* Vincular el extremo TLI con el puerto */
   if (t_bind(fd, &req, NULL) < 0)
      return -1;
   return fd;
}
```

Programa 12.8

Un servidor TLI invoca a `t_listen` para esperar una solicitud de conexión de un cliente; esta función es más parecida a la función de *sockets* `accept` que a la función de *sockets* `listen`. Recuérdese que, con *sockets*, `listen` sólo establece el tamaño del *buffer* para las conexiones pendientes. En el caso de TLI, `t_bind` utiliza el miembro `qlen` de `req` para fijar el tamaño del *buffer*.

SINOPSIS

```
#include <xti.h>

int t_listen(int fildes, struct t_call *call);
```

Spec 1170

El parámetro `fildes` es la asa devuelta por `t_open`. Si `t_listen` tiene éxito, devuelve 0 y llena la estructura `*call` con información referente a la conexión; si no tiene éxito, devuelve –1 y establece `t_errno`.

Por el hecho de que TLI se diseñó con la intención de que fuera muy general, el empleo de la estructura `struct t_call` es complicado. TLI utiliza muchos tipos de estructuras de datos cuyo tamaño depende del protocolo de transporte; por tanto, no es posible utilizar variables automáticas para estas estructuras. En vez de ello, TLI proporciona un método para asignar memoria a estructuras mediante la función `t_alloc`.

SINOPSIS

```
#include <xti.h>

char *t_alloc(int fildes, int struct_type, int fields);
```

Spec 1170

La función `t_alloc` utiliza el descriptor de archivo `fildes` devuelto por `t_open` para determinar el protocolo de transporte y el tamaño de la estructura requerida. Si quiere asignar memoria a una estructura `struct t_call` para utilizarla con `t_listen`, use `t_alloc` con `struct_type` (tipo de estructura) igual a `T_CALL` y `fields` (campos) igual a `T_ADDR`. No es necesario iniciar la estructura `t_call` antes de ejecutar `t_listen`.

Una vez que `t_listen` regresa, el servidor invoca a `t_accept` para aceptar la solicitud de comunicación. El servidor crea un descriptor de archivo nuevo para la comunicación real, así que el descriptor original devuelto por `t_open` puede utilizarse para escuchar en espera de otra solicitud de conexión. Cuando se usan *sockets*, `accept` devuelve automáticamente un nuevo descriptor de archivo para comunicación.

SINOPSIS

```
#include <xti.h>

int t_accept(int fildes, int resfd, struct t_call *call);
```

Spec 1170

El parámetro `fildes` es la asa devuelta por la función `t_open` original y `resfd` es la asa devuelta por la segunda `t_open`. El último parámetro es la estructura `struct t_call` devuelta por `t_listen`. No es necesario modificar este parámetro antes de invocar a `t_accept`.

El programa 12.9 es una implementación TLI de la función `u_listen`. La función `t_listen` hace que el invocador se bloquee hasta que llegue una solicitud de conexión o hasta que ocurra un error o un suceso asíncrono en el canal de comunicación. En caso de un

suceso asíncrono, se asigna TLOOK a la variable externa t_errno. Invoque la función t_look para investigar más a fondo.

Programa 12.9: Implementación TLI de la función u_listen de UICI.

```
int u_listen(int fd, char *hostn)
{
   struct t_call *callptr;
   struct sockaddr_in *client;
   struct hostent *hostptr;
   int newfd;
   int tret;

   if ( (callptr =
         (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
      return -1;
   }

   while( ((tret = t_listen(fd, callptr)) < 0) &&
          (t_errno == TSYSERR) && (errno == EINTR) )
      ;                          /* se interrumpe y reinicia t_listen */
   if ( (tret < 0) ||
        ((newfd = t_open("/dev/tcp", O_RDWR, NULL)) < 0 ) ||
        (t_bind(newfd, NULL, NULL) < 0 ) ||
        (t_accept(fd, newfd, callptr) < 0) ) {
      t_free((char *)callptr, T_CALL);
      return -1;
   }
   client = (struct sockaddr_in *)(callptr->addr).buf;
   hostptr =
      gethostbyaddr((char *)&(client->sin_addr.s_addr), 4, AF_INET);
   if (hostptr == NULL)
      strcpy(hostn, "unknown");
   else
      strcpy(hostn, hostptr->h_name);
   t_free((char *)callptr, T_CALL);
   return newfd;
}
```

Programa 12.9

Si una llamada TLI falla porque fue interrumpida por una señal, la llamada devuelve –1, asigna TSYSERR a t_errno y asigna EINTR a errno. La llamada u_listen reinicia t_listen automáticamente cuando detecta que la llamada fue interrumpida por una señal.

Si ocurre un error, se debe liberar la memoria asignada mediante t_alloc. Aunque esto puede ser innecesario en el caso de programas que terminan cuando ocurre un error, cabe la posibilidad de que un servidor vuelva a escuchar después de una solicitud fallida. Los progra-

mas de ejecución larga pueden experimentar fugas de memoria graves si no se tiene cuidado de liberar toda la memoria asignada.

El cliente TLI invoca a `t_open`, `t_bind` y `t_connect`.

SINOPSIS

```
#include <xti.h>

int t_connect(int fildes, const struct t_call *sndcall,
              struct t_call *rcvcall);
```

Spec 1170

El cliente asigna memoria a `*sndcall` (enviar llamada) con `t_alloc` igual que antes y la llena con información acerca del servidor. Si la conexión se establece con éxito, la función `t_connect` llena la estructura `*rcvcall` (recibir llamada) con información referente a la conexión. Si no se necesita esta información, el invocador puede pasar un apuntador `NULL` para `rcvcall`.

El programa 12.10 es la función `u_connect` para establecer una conexión TLI. El canal de comunicación se crea con `t_open` y `t_bind`. Se requieren varias tareas de preparación para direccionar el nodo remoto y asignar memoria a las estructuras de datos de usuario.

Programa 12.10: Una implementación TLI de la función `u_connect` de UICI.

```
int u_connect(u_port_t port, char *inetp)
{
    struct t_info info;
    struct t_call  *callptr;
    struct sockaddr_in server;
    struct hostent *hp;
    int fd;
    int trynum;
    unsigned int slptime;

    fd = t_open("/dev/tcp", O_RDWR, &info);
    if (fd < 0)
        return -1;

    /* Crear direcciones TCP para comunicarse con el puerto */
    memset(&server, 0, (size_t)sizeof(server));
    server.sin_family = AF_INET;    /* familia de direcciones de Internet    */
    hp = gethostbyname(inetp); /* convertir nombre en dirección de Internet */
                               /* copiar dirección de Internet en buffer    */
    if (hp == NULL) {
        fprintf(stderr, "gethostbyname returned NULL\n");
        return -1;
    }
    memcpy(&(server.sin_addr.s_addr), hp->h_addr_list[0],
```

```
            (size_t)hp->h_length);
    server.sin_port = htons((short)port);

                            /* establecer buffer con información del destino */
    if (t_bind(fd, NULL, NULL) < 0)
        return -1;

                            /* asignar memoria a la estructura para connect  */
    if ( (callptr =
            (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL)
        return -1;

    callptr -> addr.maxlen = sizeof(server);
    callptr -> addr.len = sizeof(server);
    callptr -> addr.buf = (char *)&server;
    callptr -> opt.len = 0;
    callptr -> udata.len = 0;

                        /* Seguir intentando hasta que el servidor esté listo */
    for (trynum = 0, slptime = 1;
         (trynum < RETRIES) && (t_connect(fd, callptr, NULL) < 0);
         trynum++, slptime = 2*slptime) {
        t_rcvdis(fd, NULL);
        sleep(slptime);
    }
    callptr -> addr.buf = NULL;
    t_free((char *)callptr, T_CALL);
    if (trynum >= RETRIES)
        return -1;
    return fd;
}
```

Programa 12.10

La forma en que el servidor TLI maneja las solicitudes puestas en fila da pie a una complicación. Si la cola de solicitudes pendientes tiene una longitud de 1, las solicitudes de conexión se rechazan mientras se atiende una solicitud previa. Este problema también se presenta con *sockets*, pero en este caso el problema puede evitarse utilizando una cola de mayor longitud. Por desgracia, cuando la longitud de la cola TLI es mayor que 1, el servidor se vuelve muy complejo. Una función `t_accept` falla si hay otra solicitud de conexión en la cola, así que el servidor debe escuchar para detectar todas las conexiones pendientes y ponerlas en una cola antes de que `t_accept` pueda tener éxito.

A fin de evitar la complicación de la cola de solicitudes pendientes, la versión TLI de UICI utiliza una cola de longitud 1. Esto implica que una solicitud de conexión puede fallar simplemente porque el servidor todavía no ha aceptado una solicitud anterior. El programa 12.10 es una implementación de `u_connect`. Esta función intenta de nuevo una conexión cuando ésta falla. Después de un intento de conexión fallido, `u_connect` invoca a `t_rcvdis` para recibir

una notificación del rechazo. Es necesario hacer esta llamada a `t_rcvdis` antes del siguiente intento de conexión, aunque `u_connect` no utilice la información devuelta por `t_rcvdis`. Hay varias estrategias para determinar cuánto tiempo debe esperarse entre solicitudes y cuántas veces debe intentarse establecer una conexión antes de darse por vencido. En la implementación que se presenta aquí, el cliente duplica el tiempo que espera después de cada intento infructuoso hasta un número máximo de intentos.

Al igual que el servidor, el cliente utiliza `t_free` para liberar la memoria asignada dinámicamente si ocurre un error. La función `t_free` intenta liberar toda la memoria a la que hace referencia la estructura que le es pasada como parámetro. La función no sólo libera `callptr`, sino también la memoria a la que apunta `callptr->addr.buf`. Puesto que el *buffer* forma parte de una variable automática, ocurriría un error. La implementación de `u_connect` resuelve el problema asignando `NULL` a `callptr->addr.buf` antes de invocar a `t_free`.

La versión TLI de UICI utiliza `t_rcv` y `t_snd` para implementar `u_read` y `u_write`, respectivamente.

SINOPSIS

```
#include <xti.h>

int t_rcv(int fildes, char *buf, unsigned nbytes, int *flags);
int t_snd(int fildes, char *buf, unsigned nbytes, int flags);
```

Spec 1170

Estas funciones son parecidas a `read(2)` y `write(2)` con excepción del parámetro adicional `flags`, que distingue la comunicación de alta prioridad.

Nota: La especificación 1170 indica que se debe incluir el archivo de encabezado `xti.h` en los programas que hacen referencia a las funciones TLI. Puesto que ese archivo de encabezado no está disponible en muchos sistemas, los ejemplos utilizan el archivo de encabezado `tiuser.h`, más antiguo.

12.6 STREAMS

STREAMS es una interfaz general estandarizada de System V entre el usuario y los controladores de dispositivos orientados a caracteres. Si un programa se comunica con un dispositivo orientado a caracteres en System V, lo *más probable* es que lo haga a través de un *flujo* (*stream*). (Averigüe la razón invocando a `isastream` para el descriptor de archivo en cuestión.) STREAMS ofrece diversos servicios que van desde conjuntos de protocolos de red hasta controladores de dispositivo para módems e impresoras. La interfaz STREAMS promueve un diseño modular transportable.

Dennis Ritchie, de AT&T, creó el mecanismo de flujos original en 1982. Las primeras versiones comerciales de la interfaz STREAMS aparecieron en la versión 3.0 de System V en 1986 y su nombre se puso en mayúsculas para distinguirlo de la versión anterior, no comercial. Para la mayoría de los usuarios, STREAMS resulta un tanto incomprensible, pero su uso es

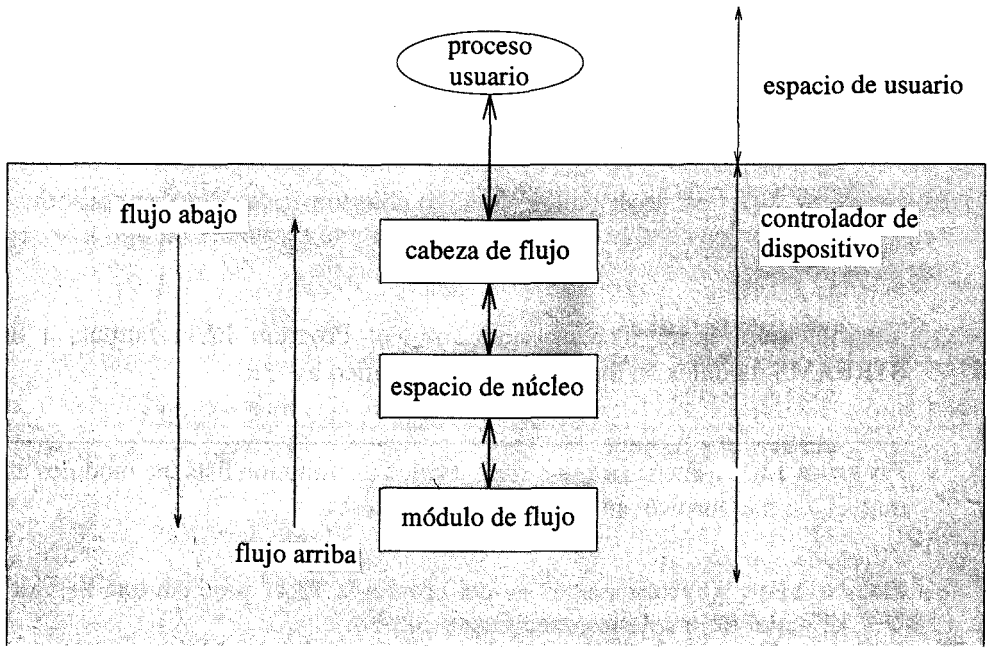


Figura 12.7: Un flujo es una interfaz de System V entre un proceso usuario y un controlador de dispositivo orientado a caracteres.

inevitable en programas que tienen cualquier interacción significativa con controladores de dispositivos. De hecho, las interfaces de *sockets* y TLI de System V por lo regular se implementan como módulos STREAMS.

En la figura 12.7 se muestra una representación esquemática de un flujo. El punto de ingreso en el núcleo se denomina *cabeza de flujo (stream head)*. El punto de ingreso es un canal de comunicación dúplex al que se accede mediante un descriptor de archivo a través de las llamadas de sistema `open`, `close`, `read`, `write`, `ioctl`, `putmsg` y `getmsg`. Cuando un programa lee de un flujo, la información viaja *flujo arriba*; cuando un programa escribe en un flujo, la información viaja *flujo abajo*.

La información en forma de mensajes se pasa desde la cabeza de flujo, a través de un número arbitrario de módulos de procesamiento, hasta el controlador de dispositivo, el cual tiene un punto de ingreso que aparece en el directorio `/dev`. Algunos de los controladores de dispositivo corresponden a dispositivos físicos reales, como el altavoz y el micrófono de una estación de trabajo. Otros controladores de dispositivo corresponden a *dispositivos simulados* o *pseudodispositivos*. Los controladores de este último tipo se denominan *controladores de pseudodispositivos*.

Los módulos de procesamiento comunes de STREAMS se encuentran en `/kernel/strmod` e incluyen `bufmod`, `connld`, `ldterm`, `pipemod`, `ptem`, `rpcmod`, `sockmod`, `timod`, `tirdwr` y `ttcompat`. En teoría, los programadores pueden crear sus propios módulos STREAMS, pero dado que los módulos residen en el núcleo después de meterse en el flujo, se requiere privilegios de superusuario para instalar los módulos en los directorios de dispositivos del sistema.

La opción `I_LIST` de `ioctl` produce una lista de los módulos de un flujo. La sinopsis de esta variación de `ioctl` es

```
#include <unistd.h>
int ioctl(int fd, I_LIST, struct str_list *mlist)
```

El parámetro `fd` es un descriptor de archivo abierto para el flujo en cuestión. El parámetro `*mlist` contiene los nombres de los módulos en una estructura de tipo `struct str_list` que se define así:

```
struct str_mlist {
        char l_name[FMNAMESZ+1];
};

struct str_list {
        int sl_nmods;
        struct str_mlist *sl_modlist;
};
```

La función `list_stream_modules` del programa 12.11 produce una lista de los módulos STREAMS en el flujo abierto especificado con `fd`.

Programa 12.11: La función `list_stream_modules` produce una lista de los módulos y controladores que están en el flujo abierto `fd`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/conf.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stropts.h>

/* Envía los nombres de los módulos del flujo fd a stdout.
 * Devuelve -1 si hay error o 0 si tiene éxito
 */
int list_stream_modules(int fd)
{
    struct str_list mlist;
    int num_mods, i;

    if (!isastream(fd)) {
       printf("The descriptor %d is not a stream\n", fd);
       return -1;
    }
    if ((num_mods = ioctl(fd, I_LIST, NULL)) == -1)
       return -1;
```

```
    mlist.sl_nmods = num_mods;
    mlist.sl_modlist =
       (void *) calloc(num_mods, sizeof(struct str_mlist));
    if (mlist.sl_modlist == NULL)
       return -1;
    if (ioctl(fd, I_LIST, &mlist) == -1)
       return -1;

    printf("Module(s) on the stream:\n");
    for (i = 0; i < num_mods; i++) {
        printf("%s\n", mlist.sl_modlist->l_name);
        mlist.sl_modlist++;
    }
    return 0;
}
```

Programa 12.11

Ejemplo 12.6

La siguiente ejecución de `who` *muestra que el usuario* `robbins` *ingresó en la máquina dos veces a través de dos controladores de dispositivo de pseudoterminal* `pts/1` *y* `pts/2` *desde la máquina remota* `vip`.

```
% who
robbins       pts/1        Oct  9 09:26        (vip)
robbins       pts/2        Oct  9 09:26        (vip)
```

El siguiente segmento de código invoca a `list_stream_modules` *para producir una lista de módulos de flujo en* `/dev/pts/2` *para el usuario* `robbins`.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

if ((fd = open("/dev/pts/2", O_RDONLY)) == -1)
   perror("Could not open /dev/pts/2");
else if (list_stream_modules(fd) == -1)
   fprintf(stderr, "Could not list command modules\n");
}
```

Una pseudoterminal representativa podría tener los siguientes módulos en su flujo.

```
Módulo(s) en el flujo:
    ttcompat
    ldterm
    ptem
    pts
```

Los dos módulos superiores del flujo de pseudoterminal del ejemplo 12.6, `ttcompat` y `ldterm`, establecen la disciplina de línea de terminal; es decir, estos módulos hacen que parezca que el flujo proviene de una terminal. El módulo `ldterm` maneja el carácter de eliminación (*delete*), el almacenamiento temporal de líneas y otras funciones. El módulo `ttcompat` hace que funcionen versiones antiguas de `ioctl`. El módulo `ptem` proporciona un control tipo terminal, por ejemplo, siguiendo la pista al tamaño de la ventana y haciendo caso omiso de cambios en la tasa de *bauds* o la paridad. El controlador de dispositivo de pseudoterminal subyacente es `pts`.

System V implementa los entubamientos conectando dos cabezas de flujo como se muestra en la figura 12.8. Aquí, `fd[0]` y `fd[1]` son flujos independientes, y el extremo de lectura de `fd[0]` se conecta al extremo de escritura de `fd[1]` y viceversa.

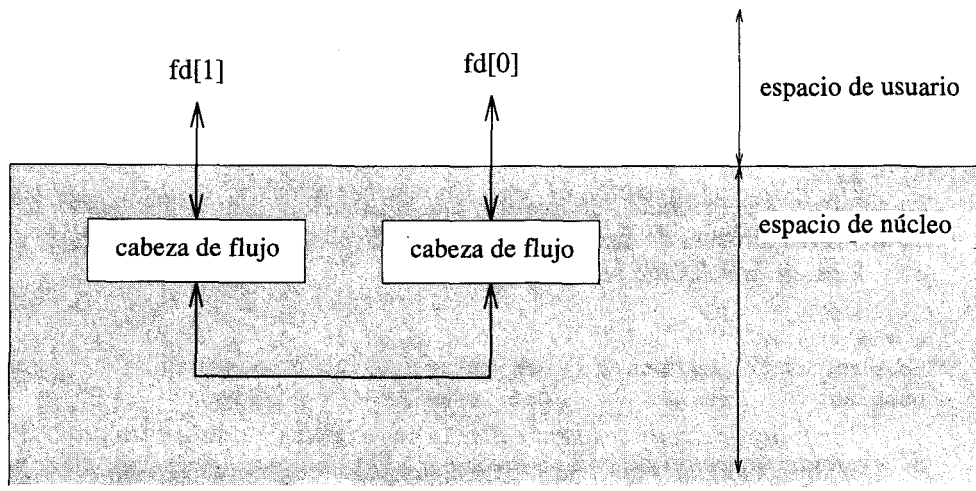


Figura 12.8: System V implementa los entubamientos conectando dos cabezas de flujo.

Ejercicio 12.2

¿Qué salida producirá el siguiente segmento de código?

```
#include <stdio.h>
#include <unistd.h>
int fd[2];

if (pipe(fd) == -1)
   perror("Bad pipe");
else if (list_stream_modules(fd[1]) == -1)
   fprintf(stderr, "Could not list modules\n");
```

Respuesta:

El código deberá indicar que no hay módulos ni controladores de dispositivo en el flujo.

La interfaz STREAMS es atractiva porque un programador puede meter módulos de procesamiento en el flujo y alterar el aspecto fundamental del dispositivo sin tener que modificar código del núcleo.

Ejemplo 12.7

El siguiente segmento de código hace que un entubamiento semeje una terminal metiendo los módulos apropiados en el flujo.

```
#include <unistd.h>
#include <stropts.h>
int fd[2];

if (pipe(fd) == -1)
   return -1;
if (ioctl(fd[0], I_PUSH, "ldterm") == -1)
   return -1;
if (ioctl(fd[0], I_PUSH, "ttcompat") == -1)
   return -1;
```

El extremo `fd[0]` del entubamiento del ejemplo 12.7 debe manejar el carácter de eliminación y contar con almacenamiento temporal de líneas.

Los módulos de flujos pasan información flujo arriba y flujo abajo en forma de mensajes del tipo `struct strbuf` definido por

```
struct strbuf {
   int    maxlen;    /* número de bytes en el buffer */
   int    len;       /* número de bytes devueltos */
   char   *buf;      /* apuntador a los datos */
};
```

Cada módulo STREAMS consiste en una cola de lectura, una cola de escritura y puntos de ingreso de procesamiento como se muestra en la figura 12.9. La cola de lectura contiene mensajes que se mueven flujo arriba, y la cola de escritura mensajes que se mueven flujo abajo. La llamada `open` realiza la iniciación y `close` se encarga de las actividades de cierre antes de que el módulo se saque del flujo. Un módulo transfiere un mensaje a la cola de otro módulo invocando el procedimiento `put` del otro módulo. Este procedimiento puede procesar el mensaje de inmediato o bien colocarlo en la cola para que `service` lo procese posteriormente. El punto de ingreso `service` se encarga del procesamiento diferido de mensajes.

12.7 Implementación de UICI con STREAMS

STREAMS utiliza `connld` para la comunicación orientada a conexiones. Un servidor mete `connld` por el extremo de un entubamiento y asigna un nombre al entubamiento. Después, un cliente puede abrir ese entubamiento especificándolo con su nombre y recibir una conexión única con el servidor.

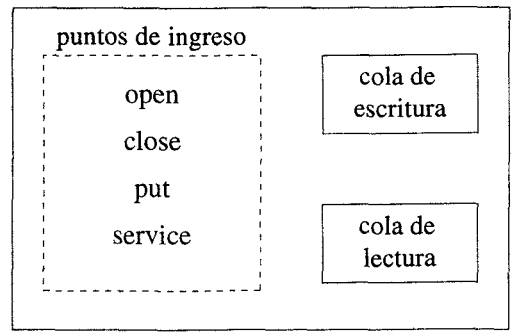


Figura 12.9: Un módulo STREAMS consta de un par de colas y varios puntos de ingreso.

Ejemplo 12.8

El siguiente segmento de código utiliza la llamada de sistema `ioctl` *con la función* `I_PUSH` *para meter el módulo de procesamiento* `connld` *por un extremo de un entubamiento STREAMS, como se muestra en la figura 12.10.*

```
#include <unistd.h>
#include <stropts.h>
int fd[2];
if (pipe(fd) == -1)
   perror("Bad pipe");
else if (ioctl(fd[1], I_PUSH, "connld") == -1)
   perror("Could not push connld");
```

Después de meter el módulo de procesamiento `connld` en el flujo, asigne un nombre al tubo `fd[1]` con `fattach`.

SINOPSIS

```
#include <stropts.h>

int fattach(int fildes, const char *path);
```

Spec 1170

La función `fattach` asigna un descriptor de archivo de STREAMS a un nombre de trayectoria, el cual corresponde a un archivo existente con sus privilegios establecidos de modo que tanto el cliente como el servidor puedan acceder a él. El nombre de trayectoria hace las veces del puerto bien conocido al cual hace referencia el cliente. Cuando el servidor ejecuta un `I_RECVFD ioctl` con el descriptor de archivo original, se bloquea hasta que otro proceso ejecuta un `open` con el nombre de archivo correspondiente. La llamada `open` devuelve al servidor un descriptor de archivo nuevo que puede usar para comunicarse con el cliente.

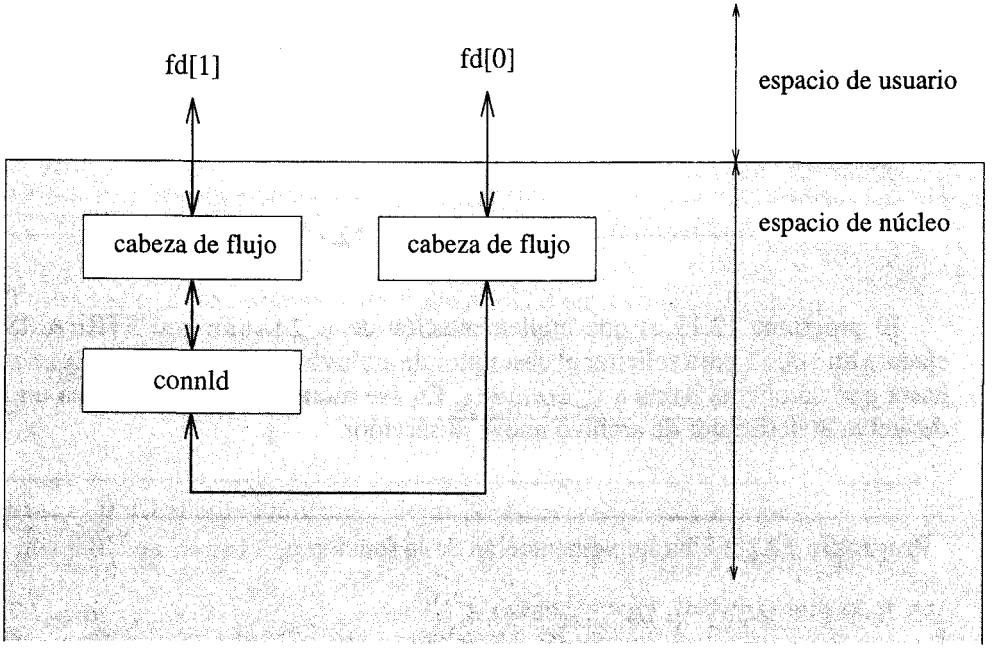


Figura 12.10: Conexiones después de meter `connld` en `fd[1]`.

El programa 12.12 es una implementación de la función `u_open` basada en entubamientos STREAMS. El nombre de trayectoria que se emplea para el puerto bien conocido no es más que un archivo del directorio `/tmp`, cuyo nombre se construye anexando un número convenido a `streams_uici_`. Si ocurre un error en cualquiera de las llamadas intermedias, `u_open` devuelve –1. Utilice `u_error` para exhibir un mensaje.

Programa 12.12: Una implementación de la función `u_open` de UICI con STREAMS.

```
int u_open(u_port_t port)
{
    char buffer[MAX_CANON];
    int mtpoint;
    int fd[2];

    sprintf(buffer, "/tmp/streams_uici_%d", (int)port);
    unlink(buffer);
    if ( ((mtpoint = creat(buffer, MTMODE)) != -1) &&
         (close(mtpoint) != -1) &&
         (pipe(fd) != -1) &&
```

```
        (ioctl(fd[1], I_PUSH, "connld") != -1) &&
        (fattach(fd[1], buffer) != -1) )
        return fd[0];
    close (fd[0]);
    close (fd[1]);
    return -1;
}
```

Programa 12.12

El programa 12.13 es una implementación de `u_listen` con STREAMS. El servidor ejecuta un `ioctl` para solicitar el descriptor de archivo del flujo privado. El `ioctl` se bloquea hasta que un cliente llama a `u_connect`. En ese momento, el `ioctl` crea un flujo nuevo y devuelve el descriptor de archivo nuevo al servidor.

Programa 12.13: Una implementación de la función `u_listen` de UICI con STREAMS.

```
int u_listen(int fd, char *hostn)
{
   mystrrecvfd conversation_info;
   int retval;

   while ( ((retval =
           ioctl(fd, I_RECVFD, &conversation_info)) == -1) &&
          (errno == EINTR) )
      ;
   if (retval == -1)
      return -1;
   *hostn = 0;
   return conversation_info.f.fd;
}
```

Programa 12.13

El programa 12.14 es una implementación de la llamada de cliente `u_connect` con STREAMS. El cliente abre el archivo que el servidor está esperando. La llamada `open` devuelve el descriptor de archivo para comunicación. Puesto que la implementación de UICI con STREAMS no apoya la comunicación en red, `u_listen` asigna al nombre de nodo la cadena vacía y `u_connect` no utiliza su segundo parámetro.

Programa 12.14: Implementación de la función `u_connect` de UICI con STREAMS.

```
int u_connect(u_port_t port, char *hostn)
{
    char buffer[MAX_CANON];
```

```
    sprintf(buffer, "/tmp/streams_uici_%d", (int)port);
    return open(buffer, O_RDWR);
}
```

Programa 12.14

12.8 UICI segura con respecto de los hilos

En esta sección examinaremos los problemas que implica implementar UICI de modo que sea segura con respecto de los hilos y proporcionaremos detalles de una implementación que se puede utilizar con hilos.

Las llamadas de sistema `read` y `write` de UNIX son inherentemente inseguras para usarse con hilos debido a la forma en que utilizan `errno` para devolver información sobre errores. Este problema se ha resuelto para la programación multihilo mediante el empleo de datos para hilos específicos con `errno`. De manera similar, la TLI segura respecto de los hilos utiliza datos para hilos específicos con `t_errno`. Esta representación es necesaria para preservar la sintaxis de las llamadas de sistema estándar de UNIX. Los programas de usuario no pueden utilizar con facilidad datos para hilos específicos, ni conviene que lo intenten. La implementación de UICI en términos de *sockets* casi es segura respecto de los hilos porque la implementación subyacente de los *sockets* es segura respecto de los hilos.

Las implementaciones con *sockets* y TLI de las secciones 12.4 y 12.5 no tienen en cuenta unos cuantos problemas menores. La implementación TLI utiliza `t_errno` para guardar información de errores. Esto funciona en tanto todos los errores sean generados por las llamadas de red, pero hay casos en que no sucede así. La función `u_connect` llama a `gethostbyname` y `u_listen` llama a `gethostbyaddr`. Estas funciones establecen `h_errno` en lugar de `errno` o `t_errno`, lo cual puede hacer que `u_error` produzca un mensaje engañoso. Además, estas llamadas utilizan memoria estática y por tanto no son seguras respecto de los hilos. La estructura `struct hostent` devuelta contiene apuntadores a dicha memoria estática. La versión de UICI segura respecto de los hilos emplea `gethostbyname_r` y `gethostbyaddr_r`.

SINOPSIS

```
#include <netdb.h>

struct hostent *gethostbyname_r(const char *name,
     struct hostent *result, char *buffer, int buflen,
     int *h_errnop);
struct hostent *gethostbyaddr_r(const char *addr,
     int length, int type, struct hostent *result,
     char *buffer,  int buflen, int *h_errnop);
```

Estas funciones realizan las mismas tareas que sus contrapartes inseguras pero no utilizan memoria estática. El parámetro `result` es un apuntador a una estructura `struct hostent` proporcionada por el usuario que contiene el resultado. Ciertos apuntadores de esta estructura

apuntan al *buffer* `buffer` proporcionado por el usuario, que tiene una longitud de `buflen`, la cual debe ser suficiente para contener los datos generados. La información de errores se coloca en `*h_errorp` si la función devuelve `NULL`.

Incluso con estas funciones, las implementaciones de UICI con *sockets* y TLI tendrían un problema debido a que `errno` y `t_error` no quedarían establecidos, y por tanto `u_error` no exhibiría el mensaje correcto.

A fin de que las versiones seguras respecto de los hilos de las funciones de UICI sigan siendo similares a las originales (inseguras), no se ha modificado la interpretación de los valores de retorno de estas funciones. En vez de ello, cada una lleva un parámetro adicional que representa el error. Todas estas funciones, con excepción de `u_error_r`, pasan un apuntador a una estructura `uerror_t`. Si no se necesita información sobre los errores, se puede pasar un apuntador `NULL`. Las versiones nuevas de todas las funciones tienen un sufijo `_r` que indica que son seguras respecto de los hilos.

La estructura `uerror_t` puede depender de la implementación. En la implementación con *sockets* contiene dos enteros. Uno contiene el valor de `errno` si es que éste se estableció y el otro contiene un valor que indica un error de `gethostbyname_r` o `gethostbyaddr_r`. En la implementación TLI se requieren tres enteros; el adicional es para `t_errno`. La versión TLI de `uerror_t` es

```
typedef struct {
   int tli_error;
   int syserr;
   int hosterr;
} uerror_t;
```

Los prototipos de las nuevas funciones de UICI son

```
int u_open_r(unsigned short port, uerror_t *errorp);
int u_listen_r(int fd, char *hostn, uerror_t *errorp);
int u_connect_r(unsigned short port, char *inetp,
                uerror_t *errorp);
int u_close_r(int fd, uerror_t *errorp);
ssize_t u_read_r(int fd, void *buf, size_t nbyte,
                 uerror_t *errorp);
ssize_t u_write_r(int fd, void *buf, size_t nbyte,
                  uerror_t *errorp);
void u_error_r(char *s, uerror_t error);
int u_sync_r(int fd, uerror_t *errorp);
```

El programa 12.15 es la implementación TLI de `u_error_r` y el programa 12.16 es una implementación de `u_connect_r`. Si la función `gethostbyname_r` devuelve un error, `u_connect_r` establece el miembro `hosterr` de la estructura `error`. Todos los demás errores establecen el miembro `tli_error` y posiblemente el miembro `syserror` de la estructura `error`. La función no establece `t_errno` directamente, ya que POSIX permite que ésta sea una macro. La función `u_set_error` establece la estructura de error. En el apéndice B se presenta una implementación completa de la UICI segura respecto a los hilos.

Programa 12.15: Implementación TLI de la función `u_error_r` de UICI segura respecto a los hilos.

```
#define GETHOSTNOERROR     0
#define GETHOSTBYNAMEERROR 1
#define GETHOSTBYADDRERROR 2

void u_error_r(char *s, uerror_t error)
{
   if (error.hosterr == GETHOSTBYNAMEERROR)
      fprintf(stderr,"%s: error in getting name of remote host\n", s);
   else if (error.hosterr == GETHOSTBYADDRERROR)
      fprintf(stderr,"%s: error converting host name to address\n", s);
   else if (error.tli_error == TSYSERR)
      fprintf(stderr,"%s: %s\n", s, strerror(error.syserr));
   else
      fprintf(stderr,"%s: %s\n", s, t_errlist[error.tli_error]);
}
```

Programa 12.15

Programa 12.16: Implementación TLI de la llamada de cliente `u_connect_r` de UICI segura respecto a los hilos.

```
static void u_set_error(uerror_t *errorp)
{
   if (errorp == NULL)
      return;
   errorp -> hosterr = GETHOSTNOERROR;
   errorp -> tli_error = t_errno;
   if (t_errno == TSYSERR)
      errorp ->syserr = errno;
}

int u_connect_r(unsigned short port, char *inetp, uerror_t *errorp)
{
   struct t_info info;
   struct t_call *callptr;
   struct sockaddr_in server;
   struct hostent *hp;
   struct hostent hostresult;
   int fd;
   int trynum;
   unsigned int slptime;
   int herror;
   char hostbuf[HOSTBUFFERLENGTH];

   fd = t_open("/dev/tcp", O_RDWR, &info);
   if (fd < 0) {
```

```
        u_set_error(errorp);
        return -1;
    }
                    /* Crear direcciones TCP para comunicarse con el puerto */
    memset(&server, 0, (size_t)sizeof(server));
    server.sin_family = AF_INET;        /* familia de direcciones de Internet */
                                /* convertir nombre en dirección de Internet */
    hp = gethostbyname_r(inetp, &hostresult, hostbuf, HOSTBUFFERLENGTH,
                         &herror);
    if (hp == NULL) {
        errorp -> hosterr = GETHOSTBYNAMEERROR;
        return -1;
    }
                                /* copiar dirección de Internet en el buffer */
    memcpy(&(server.sin_addr.s_addr), hostresult.h_addr_list[0],
       (size_t)hostresult.h_length);
    server.sin_port = htons(port);
                            /* establecer buffer con información del destino */
    if ( (t_bind(fd, NULL, NULL) < 0) ||
         ((callptr =
             (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) ) {
       u_set_error(errorp);
       return -1;
    }

    callptr -> addr.maxlen = sizeof(server);
    callptr -> addr.len = sizeof(server);
    callptr -> addr.buf = (char *)&server;
    callptr -> opt.len = 0;
    callptr -> udata.len = 0;
                                                    /* Reintentar si no hay éxito */
    for (trynum = 0, slptime = 1;
        (trynum < RETRIES) &&
           (t_connect(fd, callptr, NULL) < 0);
        trynum++, slptime = 2*slptime) {
        t_rcvdis(fd, NULL);
       sleep(slptime);
    }

    callptr -> addr.buf = NULL;
    t_free((char *)callptr, T_CALL);
    if (trynum >= RETRIES) {
       u_set_error(errorp);
       return -1;
    }
    return fd;
}
```

Programa 12.16

También puede obtenerse una versión de UICI segura respecto de los hilos utilizando `gethostbyname` y `gethostbyaddr` si estas funciones se protegen mediante candados mutex.

12.9 Ejercicio: Transmisión de audio

En esta sección extenderemos el servidor y el cliente de UICI de los programas 12.2 y 12.4 para enviar información de audio del cliente al servidor. Estos programas pueden servir para implementar un intercomunicador de red, un servicio telefónico de red o radiodifusión por red, como se describe en el capítulo 13. Comenzaremos por incorporar el audio en el servidor y el cliente de UICI:

- Implementar el copiado de archivos con el servidor y cliente de UICI que aparecen en los programas 12.2 y 12.4.
- Probar el programa con las tres implementaciones distintas de UICI. (Aplicar `diff` a los archivos de entrada y salida para verificar que todas las transferencias se lleven a cabo correctamente.)
- Modificar el servidor y el cliente de modo que invoquen las funciones de audio creadas en el capítulo 3 para transmitir audio del micrófono del cliente al altavoz del servidor.
- Probar las funciones de audio.

El programa transmite incluso cuando no se está hablando, porque una vez que el programa abre el dispositivo de audio, el controlador de dispositivo subyacente y la tarjeta de interfaz muestrean la entrada de audio a intervalos fijos hasta que el programa cierra el archivo. El muestreo continuo produce una cantidad de datos tan grande que su transmisión por la red resulta prohibitiva. Utilice un filtro para detectar los paquetes que contienen voz y los paquetes de audio desechables que no contienen voz. Un método de filtrado sencillo consiste en convertir los datos u-law a una escala lineal y rechazar los paquetes que queden abajo de un valor de umbral. El programa 12.17 es una implementación de este filtro para Solaris 2. La función `has_voice` devuelve 1 si el paquete contiene voz y 0 si debe desecharse. Incorpore `has_voice` u otro filtro de modo que el cliente no transmita silencio.

Programa 12.17: Función de umbral para filtrar datos que no contienen voz.

```
#include <stdlib.h>
#include "/usr/demo/SOUND/include/multimedia/audio_encode.h"

   /* amplitud del ruido ambiental, PCM lineal */
#define THRESHOLD 20

/* Devolver 1 si en audio_buffer hay valores que rebasan el umbral. */
```

```
int has_voice(char *audio_buffer, int length)
{
   int i;

   for (i = 0; i < length; i++)
      if (abs(audio_u2c(audio_buffer[i])) > THRESHOLD)
         return 1;
   return 0;
}
```

Programa 12.17

Escriba las siguientes mejoras al servicio de transmisión de audio básico:

- Cree una función de calibración que permita ajustar el umbral de detección de voz con base en el nivel actual del ruido ambiental del recinto.
- Utilice algoritmos de filtrado más elaborados que los simples umbrales.
- Lleve el conteo del total de paquetes y del número de paquetes que contienen datos de voz. Exhiba la información en el error estándar cuando el cliente reciba una señal `SIGUSR1`.
- Incorpore opciones de control de volumen tanto en el lado del cliente como en el del servidor.
- Diseñe una interfaz para aceptar o rechazar conexiones con base en información del transmisor.
- Invente protocolos análogos al ID de quien llama y a la espera de llamadas.
- Agregue una opción en el lado del servidor para grabar el audio recibido en un archivo y reproducirlo posteriormente. La grabación es sencilla si el cliente envía todos los paquetes. Sin embargo, como el cliente sólo envía paquetes con voz, una grabación directa no se oirá bien al reproducirse porque todos los silencios quedarán comprimidos. Guarde información de tiempos además de la información de audio en los datos grabados.

12.10 Ejercicio: Servidor de *ping*

Cuando un usuario utiliza el comando `ping` con un nodo de la red, la respuesta es un mensaje que indica si el nodo está activo.

Ejemplo 12.9

El siguiente comando consulta el nodo `vip`.

```
ping vip
```

El comando podría exhibir el siguiente mensaje para indicar que `vip` *está respondiendo a las comunicaciones en la red:*

```
vip is alive
```

En esta sección se describirá un ejercicio que utiliza `UICI` para implementar una versión un poco más elegante del servicio `ping` llamada `myping`. La función `myping` responde con

```
vip: 5:45am up 12:11, 2 users, load average: 0.14, 0.08, 0.07
```

Al igual que `ping`, el programa `myping` es una aplicación cliente-servidor. Un servidor `myping` que se ejecuta en el nodo escucha en un puerto bien conocido para detectar solicitudes de clientes. El servidor bifurca un hijo para que responda a la solicitud. El proceso servidor original sigue escuchando. Suponga que el número de puerto bien definido para `myping` está definido por la constante `MYPINGPORT`.

- Escriba el código para el cliente de `myping`. El cliente recibe el nombre del nodo como argumento de línea de comandos, establece una conexión con el puerto `MYPINGPORT`, lee lo que llega por la conexión y hace eco de esta respuesta en la salida estándar hasta que detecta el fin de archivo, después de lo cual cierra la conexión y termina. Suponga que si falla el intento de establecer una conexión con el nodo, el cliente duerme durante `SLEEPTIME` segundos y luego hace un reintento. Cuando el número de intentos de conexión infructuosos exceda `RETRIES`, el cliente exhibirá el mensaje que el nodo no está disponible y terminará.
- Escriba código para el servidor de `myping`. El servidor escucha en el puerto `MYPINGPORT` para detectar conexiones. Si se establece una conexión, el servidor bifurca un hijo para que atienda la solicitud y el proceso original sigue escuchando en `MYPINGPORT`. El hijo cierra el descriptor de archivo para escuchar, invoca la función `process_ping`, cierra el descriptor de archivo para comunicación y termina. En este paso, escriba una función `process_ping` que sólo exhiba un mensaje de error.
- Escriba una función `process_ping` con el siguiente prototipo:

```
int process_ping(int communfd);
```

La función `process_ping` escribe el mensaje de respuesta en el descriptor de archivo para comunicación. Un ejemplo de mensaje sería

```
vip: 5:45am up 12:11, 2 users, load average: 0.14, 0.08, 0.07
```

El mensaje consiste en el nombre del nodo y los resultados de ejecutar el comando `uptime`. Utilice `uname` para obtener el nombre del nodo.

SINOPSIS

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

POSIX.1, Spec 1170

Un miembro de `struct utsname` es `char nodename[Sys_NMLN]`, que especifica el nombre del nodo. Utilice la función `system` para ejecutar el comando `uptime`.

La llamada `system` bifurca un hijo para ejecutar con `execvp` el comando dado mediante `string`. Para la llamada, `string` es `"uptime"`. Puesto que `uptime` envía su resultado a la salida estándar, redirija esta salida al descriptor `communfd` antes de llamar a `system`.

SINOPSIS

```
#include <stdlib.h>

int system(const char *string);
```

Spec 1170

12.11 Lecturas adicionales

Computer Networks, 2a. ed., de Tanenbaum [90], es una referencia estándar sobre redes de computadoras y el modelo OSI. *UNIX Network Programming*, de Stevens [85], es un clásico en el área de las comunicaciones en redes, pero ha perdido algo de actualidad. *UNIX System V Network Programming*, de Rago [71], es un excelente libro de referencia actualizado que trata acerca de la programación de redes bajo System V. Los servicios de red todavía no se contemplan en POSIX, pero la especificación 1170 [99] estandariza tanto los *sockets* como TLI.

Capítulo 13

Proyecto: *Radio por Internet*

Las tecnologías de difusión, teléfonos y redes están convergiendo con rapidez. Las compañías de cable están ofreciendo servicio telefónico y conexiones con la Internet. Las compañías telefónicas desean ingresar en el negocio del entretenimiento con el video por pedido y los servicios de datos de alta velocidad. El *software* para videoconferencias y conferencias telefónicas por la Internet está ampliamente disponible. Es probable que el resultado final de los conflictos entre estas fuerzas en competencia dependa más de la política y las decisiones de regulación que de los méritos técnicos. Sea cual fuere el resultado, aumentará el número de computadoras que manejarán flujos de voz y audio además de datos. En este capítulo exploraremos la interacción de los hilos con la comunicación por redes a través de una aplicación de productor-consumidor llamada *buffer* multiplex. El proyecto utiliza el *buffer* para sincronizar una radiodifusión por una red, haciendo audibles los retardos en la transmisión por red y la sincronización.

La Internet Talk Radio es el resultado de la rápida expansión de los recursos de multimedios en la Internet. Programas de audio producidos por profesionales y que son de interés para los viajeros en la *carretera de la información* se codifican en el formato `.au` de Sun y se envían a servidores regionales. Una vez que el programa se ha distribuido a los sitios regionales, éste se encuentra disponible para transmisiones locales a través de un programa de multidifusión llamado `radio` por `ftp`.

En forma independiente, con respecto de Internet Talk Radio, se ha definido una espina dorsal de multidifusión llamada MBone para la Internet [63]. MBone es una combinación de multidifusión y enlaces punto por punto diseñada para facilitar la distribución de información difundida por la Internet. El término *difusión* indica que se distribuye un paquete a todos los nodos de una red. El término *multidifusión* indica que se distribuye un paquete a un subconjunto

de nodos de la red. Una conexión *punto por punto* implica que un nodo de origen específico envía un paquete a un destino específico.

Las redes locales se interconectan mediante *puentes* y *ruteadores* [69] para formar redes de área extensa que a su vez se interconectan para formar la Internet. El protocolo de capa de red subyacente de la Internet (llamado de manera apropiada IP, siglas en inglés de Protocolo de Internet) apoya la comunicación multidifundida para las redes de área local. Los puentes y ruteadores por lo regular no envían de nuevo los paquetes difundidos o multidifundidos para evitar que un nodo indisciplinado inunde la Internet. No obstante, hay situaciones en las que semejantes recursos de multidifusión resultan útiles, como en la distribución de un programa de radio a receptores en la Internet. En lugar de enviar una copia individual del mismo paquete a cada uno de los receptores, la multidifusión distribuye una copia del paquete por la red a todos los receptores especificados.

A fin de apoyar una multidifusión eficiente por los puentes y ruteadores, el proyecto MBone define una red virtual compuesta por subredes de la Internet conectadas mediante ruteadores especiales que apoyan la multidifusión. Los ruteadores especiales se llaman *mrouters*. Islas disconexas de la MBone se conectan mediante enlaces de comunicación punto por punto. Cuando un nodo con un enlace punto por punto detecta un paquete multidifundido, lo encapsula en paquetes de IP y transmite los paquetes encapsulados por el enlace. El *mrouter* del destino convierte los paquetes de vuelta en paquetes multidifundidos y los transmite por su espina dorsal local.

13.1 Panorama general del multiplexor

En este capítulo crearemos un sistema punto por punto para distribuir archivos de audio y datos a múltiples destinos. La implementación se basa en un servidor multiplexor como el que se muestra en la figura 13.1 y que no aprovecha las capacidades de multidifusión de IP. Las entradas que el servidor recibe se multiplexan o duplican en todas las líneas de salida, las cuales son conexiones de red con los clientes. Imagine una generalización en la que las líneas de salida están conectadas a ruteadores de multidifusión que se encargan de multidifundir la información a sus redes locales. El proyecto usa el servidor multiplexor para distribuir un programa de audio a un grupo de estaciones de trabajo receptoras, de ahí el término radio por Internet que da título al capítulo.

Este proyecto es una variación del problema de productores-consumidores. Hay un productor (el difusor) y un número arbitrario de consumidores (los receptores). El problema presenta ciertos bemoles adicionales. En el problema de productores-consumidores ordinario, cada paquete es consumido por un solo consumidor. En el multiplexor, cada consumidor debe consumir todos los paquetes, y los consumidores pueden participar en (sintonizar) o abandonar (desintonizar) la difusión en cualquier instante.

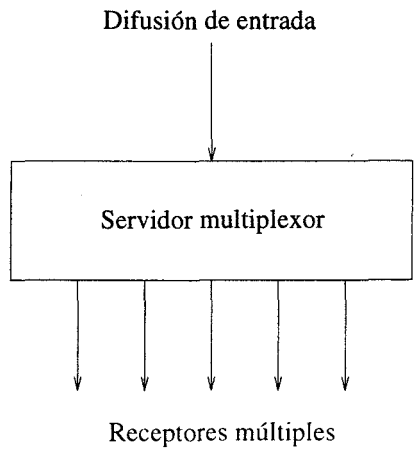


Figura 13.1: Un servidor multiplexor almacena temporalmente y copia sus entradas en todos sus puertos de salida.

13.2 Comunicación unidireccional

La primera etapa del proyecto establece una comunicación unidireccional por la red empleando la interfaz de comunicación UICI descrita en el capítulo 12. Puede utilizar la versión de UICI con *sockets*, o bien la versión TLI. La comunicación se ajusta a un modelo cliente-servidor en el que un programa servidor se ejecuta en el nodo remoto y un programa cliente en la máquina local desea establecer la comunicación, como se muestra en la figura 13.2.

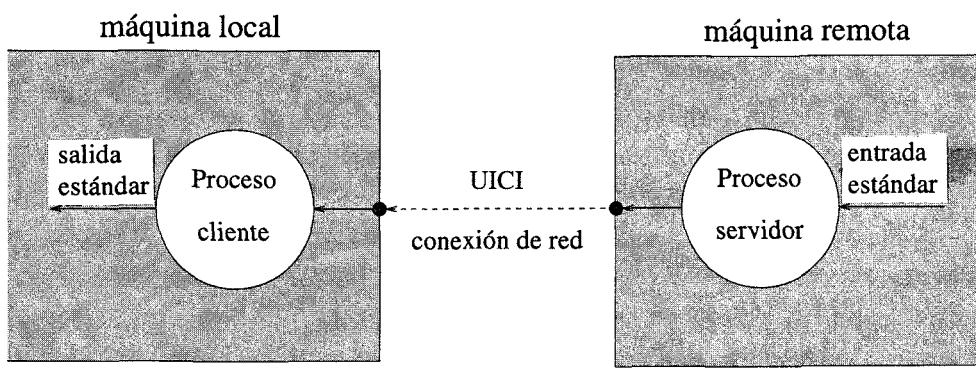


Figura 13.2: Comunicación unidireccional en una red.

En la siguiente especificación el programa servidor se llama `do_broadcast` (producir difusión) y el programa cliente se llama `get_broadcast` (recibir difusión).

- Escriba un programa servidor llamado `do_broadcast` para transmitir desde la entrada estándar hacia una red, empleando una de las versiones de UICI para red que estudiamos en el capítulo 12. `do_broadcast` escucha en un puerto bien conocido para detectar una conexión de un cliente invocando a `u_listen`. La función `u_listen` devuelve un *descriptor de archivo de conversación* que se utiliza en la comunicación real. Pase el número de puerto bien conocido al programa servidor como argumento de línea de comandos.
- Una vez que el programa `do_broadcast` ha establecido un enlace de comunicación, invoca una función `writer` (que se describe más adelante) para transmitir desde la entrada estándar hacia la red. Cuando la función `writer` regresa, el servidor vuelve a escuchar en el puerto bien conocido para detectar otro cliente de `do_broadcast`. Instrumente el código de modo que `do_broadcast` envíe mensajes informativos al error estándar.
- Escriba una función `writer` (escritor) que se encargue de la transmisión de datos de la entrada estándar a la red. El prototipo de la función `writer` es

  ```
  void *writer(void *fdp);
  ```

  La función `writer` lee de la entrada estándar invocando a `read` y envía la salida a la red invocando a `u_write` en el descriptor de archivo abierto especificado por `*fdp`. Realice las operaciones en un ciclo hasta encontrar un fin de archivo en la entrada estándar o hasta que ocurra un error. El parámetro de `writer` y el valor de retorno son apuntadores a `void`, de modo que esta función se puede invocar como hilo sin modificación.
- Escriba un programa cliente llamado `get_broadcast` que reciba comunicación unidireccional por la red mediante UICI. El cliente se conecta con el puerto bien conocido de una máquina remota llamando a `u_connect`. Pase el nombre de nodo y el número de puerto como argumentos de línea de comandos al cliente `get_broadcast`. Después de establecer la comunicación en la red, el cliente `get_broadcast` invoca a una función `reader` (que se describe más adelante) para transmitir desde la red a la salida estándar. Cuando `reader` regresa, el cliente termina.
- Escriba una función `reader` (lector) para manejar la transmisión de datos de la red a la salida estándar. El prototipo de la función `reader` es

  ```
  void *reader(void *fdp);
  ```

  La función `reader` lee información de la conexión de red invocando a `u_read` en el descriptor de archivo abierto especificado por `*fdp` y escribe en la salida estándar mediante `write`. Realice las operaciones en un ciclo hasta detectar un fin de archivo en `*fdp` o hasta que ocurra un error.

- Pruebe los programas ejecutando el servidor `do_broadcast` en una máquina y el cliente `get_broadcast` en otra. Ejecute el cliente varias veces. En cada ocasión, detenga al cliente con `ctrl-c`. Asegúrese de que sea posible iniciar otro cliente sin reiniciar el servidor. Usando el teclado, introduzca datos para la entrada estándar de `do_broadcast`.
- Pruebe el programa transmitiendo un archivo grande, digamos `my.file`. Suponga que el nombre de nodo de la máquina es `vip` y que está utilizando el puerto `8623` para la transmisión. Ejecute los siguientes comandos en el nodo `vip`:

```
do_broadcast 8623 < my.file
get_broadcast vip 8623 > my.file.out
diff my.file.out my.file
```

  Si la transmisión está funcionando correctamente y el programa está detectando el fin de archivo como es debido, los archivos `my.file` y `my.file.out` no deberán presentar diferencias. Asegúrese de probar un archivo de entrada de tamaño considerable, digamos de un megabyte o más. Utilice `diff` para comparar los archivos de entrada y de salida.
- Pruebe el programa con transmisión de audio. Suponga que el servidor `do_broadcast` se ejecuta en el nodo `vip` y el cliente `get_broadcast` se ejecuta en otra máquina. La máquina del servidor debe contar con micrófono y la máquina del cliente debe tener altavoz. La prueba es

```
do_broadcast 8623 < "/dev/audio"
get_broadcast vip 8623 > "/dev/audio"
```

  Hable al micrófono y escuche la transmisión. Utilice `ctrl-c` para detener los programas. Recuerde que se transmiten datos incluso si nadie habla. (`"/dev/audio"` es un nombre de dispositivo de Sun Microsystems para un altavoz y micrófono de estación de trabajo. Sustituya el nombre de dispositivo correcto según sea necesario.)

13.3 Comunicación bidireccional

En esta sección crearemos una versión cliente-servidor para la comunicación bidireccional [75]. Una vez que el cliente y el servidor establecen una conexión UICI con la red, se comunican de forma simétrica. Cada programa vigila la entrada estándar y un canal de entrada de su conexión remota. Un programa sin hilos podría utilizar `select` o `poll` para vigilar los dos descriptores de archivo (véase la sección 9.1). Esta implementación utiliza las funciones `reader` y `writer` de la sección 13.2 como hilos independientes para llevar a cabo la comunicación bidireccional.

Recuérdese que `writer` copia de la entrada estándar en un descriptor de archivo, mientras que `reader` copia de un descriptor de archivo en la salida estándar. Juntas, estas funciones proporcionan una comunicación bidireccional. En la figura 13.3 se muestra un diagrama de dicha comunicación. Los hilos `reader` y `writer` son independientes entre sí y no requieren sincronización.

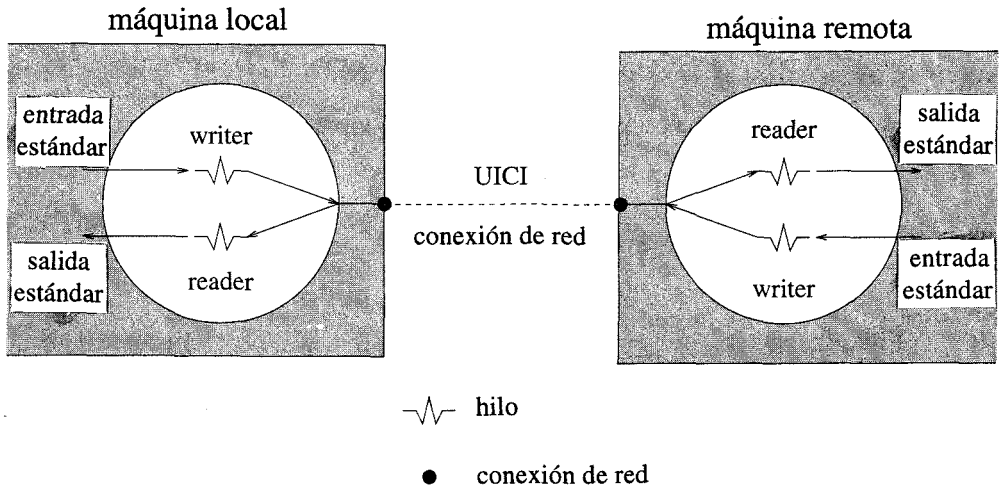


Figura 13.3: Diagrama de la implementación de comunicación bidireccional con hilos.

Modifique los programas cliente y servidor de la sección 13.2 como se indica a continuación, para convertir la comunicación unidireccional en bidireccional:

- Prepare una copia del código fuente para `get_broadcast` y modifíquela de la siguiente manera: Invoque el ejecutable modificado `call_up` (llamar). Después de establecer una conexión de red mediante `u_connect`, el cliente `call_up` crea dos hilos, uno para ejecutar `reader` y el otro para ejecutar `writer`. Pase un apuntador al descriptor de archivo de red devuelto por `u_connect` a ambos hilos.
- Prepare una copia del código fuente para `do_broadcast` y modifíquela como sigue: Invoque el ejecutable modificado `answer` (contestar llamada). Una vez que el servidor `answer_call` establece una conexión de red con el cliente mediante `u_listen`, crea dos hilos, uno para ejecutar `reader` y el otro para ejecutar `writer`. Pase un apuntador al descriptor de archivo de red devuelto por `u_listen` a ambos hilos para la comunicación en la red.
- Pruebe los programas transmitiendo un archivo grande en cada sentido y utilizando `diff` para verificar que la transmisión se realice sin errores.
- Pruebe los programas, como un intercomunicador de audio, redirigiendo tanto la entrada estándar como la salida estándar a `"/dev/audio"`. Para ello se requieren dos máquinas equipadas con micrófono y altavoces. Para probar el programa pida a dos usuarios que hablen al mismo tiempo.

13.4 El *buffer* de transmisión

El intercomunicador de audio de la sección 13.3 apoya la comunicación bidireccional entre procesos en una red. En una aplicación de radio, un proceso (el difusor) transmite a muchos procesos (los receptores). En esta sección presentaremos un *buffer* de transmisión entre el difusor y el receptor como preparación para el caso en que hay varios receptores. En la figura 13.4 el *buffer* de transmisión se representa como un *buffer* circular con elementos de tipo `buffer_t`.

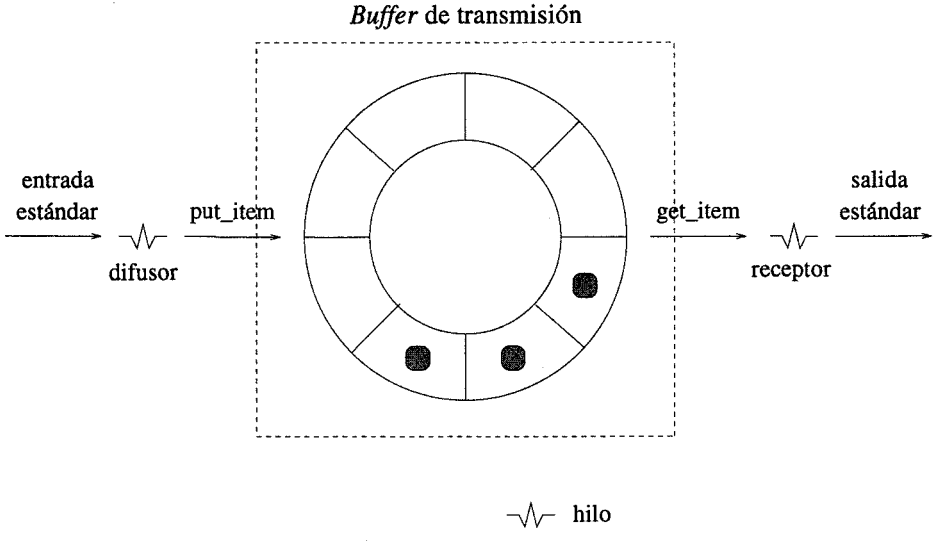


Figura 13.4: Diagrama del *buffer* de transmisión.

Defina `transmit_buffer` así:

```
#define MAXBUFSIZE 10
#define PACKETSIZE 1024
typedef struct buf {
    char buffer[PACKETSIZE];
    int packet_size;
    int number_unread;
    pthread_mutex_t element_lock;
} buffer_t

static int transmit_buffer_size;
static buffer_t transmit_buffer[MAXBUFSIZE];
static int current_start;
static int current_active_receivers;
```

Implemente `transmit_buffer` como un objeto con funciones de acceso para insertar elementos en el *buffer* y sacarlos de él. Un hilo `broadcaster` (difusor) transfiere información de un archivo de entrada al *buffer* invocando a `put_item` (poner elemento). Un hilo `receiver` (receptor) transfiere información del *buffer* a un archivo de salida invocando a `get_item` (obtener elemento).

Modifique el objeto de *buffer* circular del programa 10.1 de modo que en lugar de proteger todo el *buffer* con un candado mutex se cuente con un mutex para cada ranura del *buffer*. Un hilo sólo poseerá el mutex de un elemento mientras está modificando el miembro `number_unread` (cantidad sin leer) de dicho elemento. Al igual que en el capítulo 10, las funciones de acceso no proporcionan sincronización entre el difusor y el receptor.

La implementación del *buffer* circular del programa 10.1 incluyó las variables `bufin` y `bufout` para designar las ranuras en la que se inserta y de la que se toma. En la implementación de `transmit_buffer`, cada hilo mantiene sus propios apuntadores y los pasa como parte de las llamadas `get_item` y `put_item`. Otra modificación es que `put_item` no pasa explícitamente el elemento de datos que se va a insertar y `get_item` no devuelve el elemento de datos que se extrajo. En vez de ello, ambas funciones pasan descriptores de archivo abiertos. La función `get_item` copia el valor del *buffer* directamente al archivo representado por el descriptor `outfd`. De forma similar, `put_item` obtiene directamente los datos por insertar leyendo `infd`. La modificación elimina una operación de copiado por cada acceso a un paquete.

En esta sección se especifica la implementación del objeto de *buffer* de transmisión y se describe la forma de probar el objeto. Implemente las siguientes funciones de acceso para el objeto `transmit_buffer`:

- Escriba una función `initialize_transmit_buffer` para inicializar el *buffer* de transmisión. El prototipo de esta función es:

```
int initiali ze_transmit_buffer(int buffer_size);
```

  La función `initialize_transmit_buffer` inicializa los miembros `element_lock` (candado de elemento) y `number_unread` de cada elemento. También asigna `buffer_size` al miembro `transmit_buffer_size` (tamaño del *buffer* de transmisión). El valor `buffer_size` debe ser menor que `MAXBUFSIZE`. Si `initialize_transmit_buffer` tiene éxito, devuelve 0; en caso contrario, devuelve –1.

- Escriba una función `get_item` que copie el elemento presente en la ranura `*outp` de `transmit_buffer` en el archivo especificado por `outfd`. El prototipo de `get_item` es

```
int get_item(int outfd, int *outp);
```

  La función `get_item` incrementa `*outp` módulo `transmit_buffer_size` de modo que la variable siempre especifique la siguiente posición de la cual va a leer el invocador. La función `get_item` también decrementa `number_unread` y devuelve el nuevo valor de `number_unread`.

- Escriba una función `put_item` que lea el siguiente paquete del archivo de entrada especificado por el descriptor de archivo abierto `infd` y lo coloque en la ranura número `*inp` de `transmit_buffer`. El prototipo de `put_item` es

```
int put_item(int infd, int *inp);
```

  La función `put_item` asigna a `packet_size` de la ranura correspondiente el tamaño del paquete leído y asigna a `number_unread` el número de hilos receptores que están activos en ese momento, mismo que está especificado por `current_active_receivers`. La función `put_item` incrementa `*inp` módulo `transmit_buffer_size` de modo que la variable siempre especifique la siguiente posición en la que debe escribir el hilo invocador. La función `put_item` devuelve el número de bytes transferidos al *buffer* en caso de tener éxito, o –1 si fracasa.
- Escriba una función `get_current_start` que devuelva el valor actual de `current_start` (inicio actual). El prototipo de la función es

```
int get_current_start(void);
```

El objeto `transmit_buffer` incluye las variables estáticas `current_start` y `current_active_receivers` que se inicializan en 0 y 1, respectivamente. Estas variables estáticas tienen valores que serán constantes en esta parte del proyecto.

Para probar el objeto `transmit_buffer`, haga lo siguiente:

- Copie el programa 10.6 en un nuevo directorio. Cambie el nombre del hilo `producer` a `broadcaster` y el del hilo `consumer` a `receiver`. Invoque el ejecutable `test_buffer`.
- Modifique `broadcaster` de modo que en lugar de calcular un número fijo de elementos, continúe hasta que `put_item` devuelva un –1. El hilo `broadcaster` mantiene una posición de *buffer* local, `in`, a la cual asigna `current_start` como valor inicial. El hilo `broadcaster` pasa `&in` como segundo argumento de `put_item` y `STDIN_FILENO` como primer argumento.
- Modifique `receiver` de modo que mantenga una posición de *buffer* local, `out`, a la cual asigna valor inicial invocando a `get_current_start`. El hilo `receiver` pasa `&out` como segundo argumento de `get_item` y `STDOUT_FILENO` como primer argumento.
- Modifique el programa principal de modo que acepte el tamaño del *buffer* de transmisión como argumento de línea de comandos. Haga las modificaciones adicionales que sean necesarias para probar la biblioteca del *buffer* de transmisión.
- Utilice `test_buffer` para transmitir un archivo grande. Por ejemplo, para copiar `my.file` mediante un *buffer* de transmisión de tamaño 10 ejecute

```
test_buffer 10 < my.file > my.file.out
diff my.file my.file.out
```

13.5 Multiplexión del *buffer* de transmisión

En esta sección desarrollaremos la sincronización por multiplexión del *buffer* de transmisión que se muestra en la figura 13.5. Suponga que hay un número fijo de receptores que están activos durante toda la difusión. El programa principal recibe el tamaño del *buffer* de transmisión y el número de receptores como argumentos de línea de comandos.

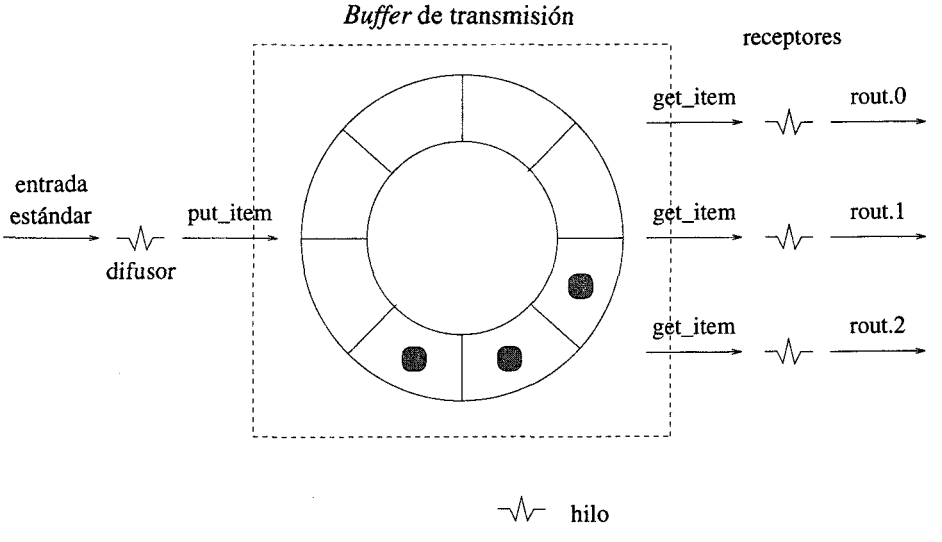


Figura 13.5: Diagrama del *buffer* de transmisión multiplexado.

- Reescriba la función `receiver` de la sección 13.4 de modo que en lugar de escribir datos en la salida estándar acepte un argumento que es un descriptor abierto del archivo en el que debe escribir.
- Reescriba la función `broadcaster` de la sección 13.4 de modo que en lugar de leer datos de la entrada estándar acepte un argumento que es un descriptor abierto del archivo del cual debe leer.
- Modifique el programa `test_buffer` de la sección 13.4 de la manera siguiente:
  * Pase el tamaño del *buffer* de transmisión y el número de receptores activos a través de la línea de comandos.
  * Invoque la función `initialize_transmit_buffer` para iniciar `transmit_buffer`.
  * Inicie todas las variables de sincronización que sean necesarias.
  * Abra `current_active_receivers` archivos con el nombre `rout.n`, donde `n` es el número del hilo `receiver` que se va a crear (comenzando con el 0). Los descriptores de archivo resultantes se pasan a los hilos `receiver` correspondientes creados por el programa principal.

* Cree `current_active_receivers` hilos `receiver` y pase a cada hilo el descriptor de archivo `rout.n` correspondiente, como parámetro, mediante `pthread_create`.
* Cree un hilo `broadcaster` como hizo en la sección 13.4 y pásele el descriptor de archivo `STDIN_FILENO`, como parámetro, mediante `pthread_create`.
* Espere para unirse con todos los demás hilos.

Pruebe el programa con archivos de diverso tamaño. Asegúrese de que los archivos se transmitan correctamente (es decir, que todos los archivos `rout.n` tengan el mismo contenido que el archivo de entrada). Utilice `diff` para comprobar que los archivos coincidan.

13.6 Receptores de red

En esta sección desarrollaremos la versión para red de los hilos `receiver`. Cree un subdirectorio y copie en él todo el código que escribió en la sección 13.5. Cambie el nombre del programa principal a `test_receivers`. El programa `test_receivers` requiere ahora tres argumentos de línea de comandos: el tamaño del *buffer* de transmisión, el número de receptores y el número de puerto bien conocido con el que se conectarán los receptores. En la figura 13.6 se muestra un diagrama del programa `test_receivers` cuando el número de hilos receptores es tres.

Implemente el programa `test_receivers` de la siguiente manera:

- Escuche en el puerto bien conocido del receptor hasta detectar el número especificado de solicitudes de receptor y cree un hilo `receiver` para cada solicitud. Pase el descriptor de archivo para conversación devuelto por `u_listen` como parámetro del hilo en la llamada a `pthread_create`.
- Cree el hilo `broadcast` y pase `STDIN_FILENO` como parámetro del hilo en la llamada a `pthread_create`.
- Únase con el hilo `broadcaster`.
- Únase con todos los hilos `receiver`.
- Haga limpieza.

Utilice el programa `get_broadcast` de la sección 13.2 para el cliente receptor. Pruebe el programa transmitiendo primero un archivo grande y luego audio como en la sección 13.2. Al transmitir audio a más de un cliente, asegúrese de ejecutar los clientes en máquinas diferentes, cada una de las cuales debe estar equipada con un altavoz. Experimente con lo siguiente:

- Diferentes valores de `transmit_buffer_size`.
- Diferentes prioridades para los hilos `broadcaster` y `receiver`.
- Hilos `broadcaster` y `receiver` con alcance `SYSTEM`.

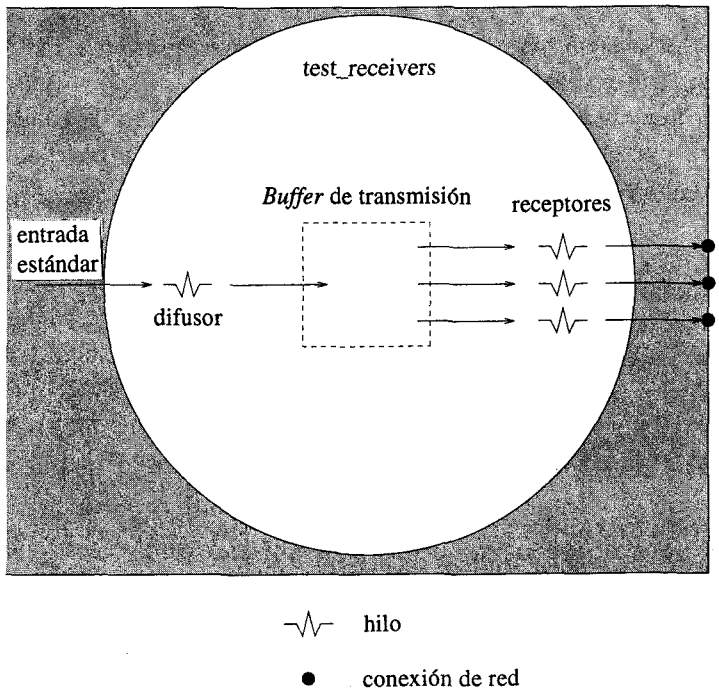


Figura 13.6: El programa `test_receivers` presenta receptores remotos que se conectan al *buffer* de transmisión a través de un puerto bien conocido.

13.7 Sintonización y desintonización

El programa que se creará en esta sección permite a los receptores unirse a una difusión ya iniciada y abandonar la difusión antes de que termine.

- Cree un directorio y copie en él los programas de la sección 13.6. Cambie el nombre del programa principal a `test_tuning`.
- El programa `test_tuning` recibe tres argumentos de la línea de comandos: el tamaño del *buffer* de transmisión, el número máximo de receptores (`max_receivers`) y el número de puerto bien conocido con el cual se conectan los clientes receptores externos.
- El programa `test_tuning` crea un hilo `get_receivers` que escucha en el puerto bien conocido del receptor para detectar solicitudes de conexión.
- A continuación, el hilo principal invoca a `pthread_create` para crear un hilo `broadcaster` con `STDIN_FILENO` como parámetro.
- Escriba una función `get_receivers` con el siguiente prototipo:

```
void *get_receivers(void *arg);
```

La función `get_receivers` deberá efectuar lo siguiente en un ciclo:

* Escuchar en el puerto bien conocido del receptor para detectar una solicitud de conexión invocando a `u_listen`.
* Cuando se recibe una solicitud, crear un hilo `receiver` nuevo y pasarle el descriptor de archivo devuelto por `u_listen`.

El hilo `get_receivers` crea hasta `max_receivers` hilos antes de regresar o de ser muerto.

- Los valores de `current_active_receivers` y `current_start` ya no son constantes. Defina un mutex para proteger estas variables compartidas. Actualice sus valores en los puntos apropiados.
- Si se mata un receptor remoto, el hilo `receiver` correspondiente experimentará un error al escribir salidas. El hilo `receiver` decrementará `current_active_receivers`, así como las cuentas apropiadas en el `transmit_buffer`.
- Desarrolle una estrategia para terminar de forma ordenada.

13.8 Difusor de red

En esta sección convertiremos el programa de Internet Radio de modo que acepte entradas de la red en lugar de la entrada estándar.

- Modifique el programa `test_tuning` de modo que el hilo principal cree el hilo `get_receivers` y luego ejecute `u_listen` en el puerto de entrada bien conocido. Invoque el ejecutable del programa modificado `test_network`.
- Cuando una fuente de difusión establece la conexión, el hilo principal crea el hilo `broadcaster` utilizando el descriptor devuelto por `u_listen` como argumento de `pthread_create`. El programa `test_network` ahora recibe cuatro argumentos de línea de comandos: el tamaño del *buffer* de transmisión, el número máximo de receptores, el puerto bien conocido al que se conectan los clientes receptores externos y el puerto bien conocido en el que el difusor escucha.
- Escriba un programa cliente `broadcast_source` que sea la fuente de difusión para la red. En la figura 13.7 se muestra un diagrama de la conexión.

13.9 Manejo de señales

Incorpore el manejo de señales en el programa `test_network` de la sección 13.8. Si `test_network` recibe la señal `SIGUSR1`, deberá matar el hilo `broadcaster` y permitir que los hilos `receiver` terminen de transmitir lo que queda en `transmit_buffer`. Investigue el mecanismo de puntos de cancelación que ofrecen los hilos POSIX.1c. Utilice un hilo dedicado con `sigwait` para manejar la señal `SIGUSR1`.

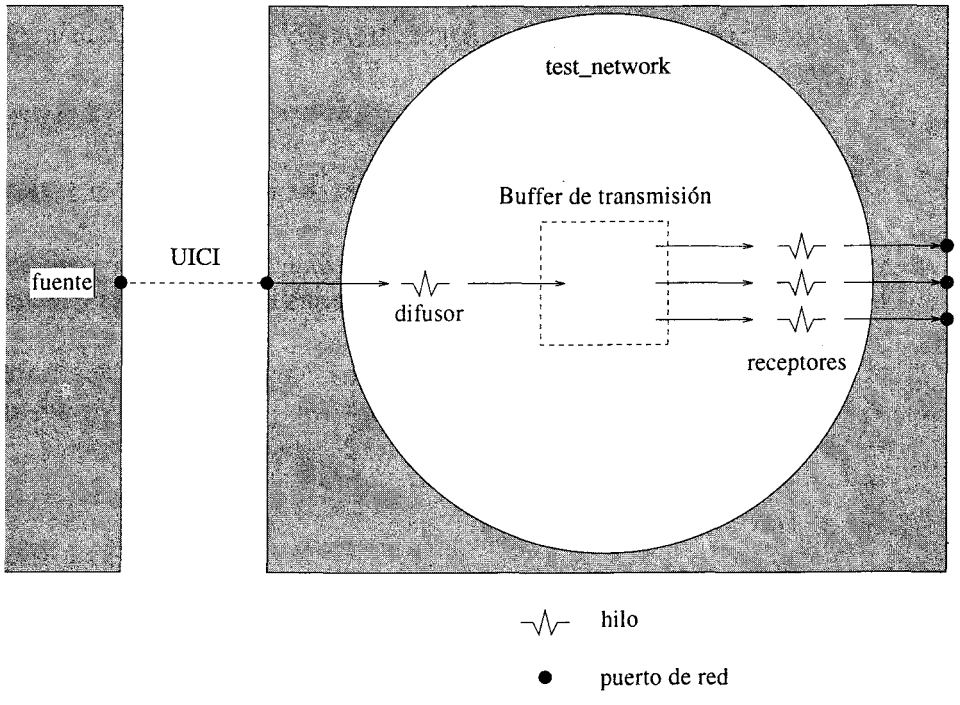


Figura 13.7: Una versión de la red del difusor.

13.10 Lecturas adicionales

Hay muchos archivos de audio disponibles en la Internet. Si desea recibir FAQ (respuestas a preguntas frecuentes), envíe un mensaje de correo a `inforadio.com`. Si desea una lista de los sitios que conservan archivos de radio, envíe correo a `sitesradio.com`. No es necesario incluir tema ni cuerpo del mensaje. La página de Internet Talk Radio en la WWW es `http://town.hall.org/radio`. El artículo "MBone provides audio and video across the Internet" analiza la disponibilidad de MBone y sugiere varios sitios de la Internet [63]. "Handling audio and video streams in a distributed environment" trata aspectos generales de las transmisiones de audio y video [43]. El número correspondiente a mayo de 1995 de *IEEE Computer* se dedicó a los sistemas multimedios [30, 34, 49, 73]. Algunos de los primeros experimentos con audio que constituyeron la base de este proyecto se informaron aquí [75].

Capítulo 14

Llamadas a procedimientos remotos

No existe consenso sobre las llamadas a procedimientos remotos (RPC, *remote procedure calls*): ¿Es esto lo mejor o lo peor que le ha podido suceder a la computación distribuida? Por desgracia, la interfaz de usuario con las RPC es un estándar en desarrollo que no es tan sencillo como debiera ser. Sea como sea, en este capítulo estudiaremos los fundamentos de las RPC y dejaremos que el lector se forme su propia opinión sobre su futuro. En el presente capítulo ilustraremos, mediante un ejemplo, la mecánica de la conversión de una función local en un servicio remoto. Después, abordaremos cuestiones tan importantes como el estado del servidor, las llamadas equipotentes y la semántica en caso de fracasos. Sun NFS (*Network File System*, sistema de archivos en red) es una aplicación comercial importante cuyos protocolos ilustran muchas de estas ideas. El capítulo concluye con una explicación de cómo crear un servidor con hilos para RPC.

En la programación estructurada tradicional y el diseño descendente, los programadores organizan los programas grandes en unidades funcionales más pequeñas a fin de producir diseños modulares en los que las funciones (o procedimientos) representan operaciones de alto nivel. Un programa invoca estas operaciones para diferentes datos pasando argumentos a las funciones a través de sus parámetros.

Parece natural generalizar la idea de las llamadas de función a un entorno distribuido permitiendo que un programa invoque una función que no se encuentre en el espacio de direcciones del proceso. Si se contara con semejante *llamada a función remota* o *llamada a procedimiento remoto*, un programador podría distribuir un programa ordinario en nodos conectados con una red y aprovechar servicios que no están disponibles localmente sin perder su diseño modular. La llamada a procedimiento remoto independiente del transporte (TI-RPC, *Transport Independent Remote Procedure Call*) de Open Network Computing (ONC, computación de red abierta) de Sun es un estándar naciente, y en este capítulo desarrollaremos la llamada a

procedimiento remoto dentro de ese sistema. En la primera sección se explica la razón de ser de las llamadas a procedimientos remotos. En secciones posteriores se presentan ejemplos que convierten llamadas locales en llamadas remotas y exploran cuestiones generales de vinculación, asignación de nombres y fracasos. En la sección 14.9 se explica cómo convertir un servidor de RPC ordinario en uno con hilos.

14.1 Funcionamiento básico

Cuando un programa invoca una función, la dirección de retorno y demás información de estado se mete en una pila de tiempo de ejecución y el control se transfiere al principio de la función. La memoria asignada en la pila se denomina *registro de activación* de la función, y también contiene valores iniciales de parámetros y direcciones para las variables automáticas declaradas dentro de la función. La llamada a una función forma parte de un solo hilo de ejecución, como se aprecia en la figura 14.1. La llamada de función da lugar a un cambio en la dirección de ejecución que representa al hilo de ejecución.

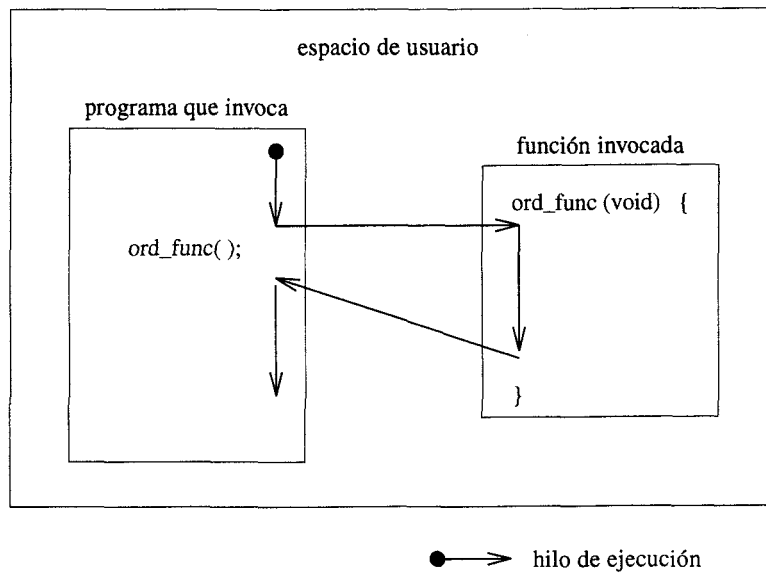


Figura 14.1: Hilo de ejecución de una llamada de función ordinaria.

El regreso de la función saca de la pila el registro de activación y asigna la dirección de retorno al contador de programa. (El regreso también podría restaurar registros y efectuar otras operaciones de aseo.) El regreso hace que el hilo continúe la ejecución del programa invocador en el enunciado que sigue a la llamada.

Ejemplo 14.1

La siguiente secuencia representa un hilo de ejecución para el proceso uno.

$$2998_1, 2999_1, 3000_1, 4000_1, 4001_1, 4002_1, 4003_1, 3001_1, 3002_1$$

Los subíndices en el ejemplo 14.1 indican el hilo de ejecución en cuestión. En el enunciado 3000, el hilo uno invoca una función. Los enunciados de la función son 4000, 4001, 4002 y 4003. A continuación, el hilo de ejecución regresa al enunciado 3001_1 del programa invocador.

Los programas solicitan servicios de sistema ejecutando una *llamada de sistema*, que a primera vista funciona igual que una función ordinaria excepto que la llamada hace referencia a código del sistema operativo, no en el programa. Sin embargo, desde el punto de vista del hilo de ejecución hay diferencias importantes entre una llamada de sistema y una llamada de función ordinaria. En la figura 14.2 se muestra el hilo de ejecución de una llamada de sistema.

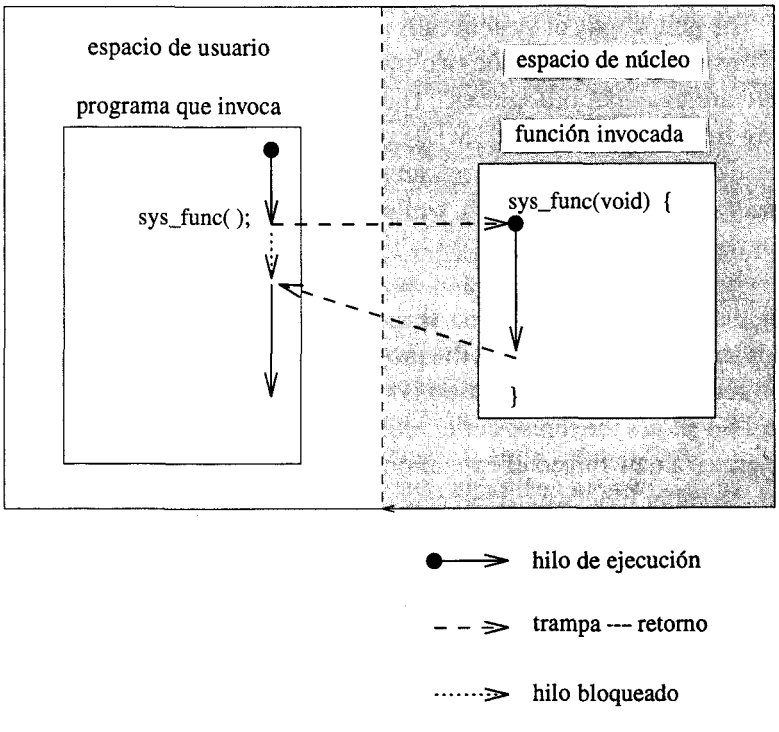


Figura 14.2: Hilo de ejecución de una llamada de sistema.

Una llamada de sistema es un señalamiento que conduce a un punto de ingreso al núcleo y que hace que el hilo de ejecución del programa invocador se bloquee. Un hilo de ejecución

aparte, que cuenta con una pila en el núcleo, ejecuta la llamada de sistema. Cuando el señalamiento regresa, el hilo de ejecución original se desbloquea. Un programa puede hacer una llamada de sistema directamente o a través de funciones de la biblioteca de C. Las funciones de biblioteca constituyen *envolturas* para los servicios subyacentes. Una envoltura podría ajustar los parámetros y realizar otras actividades de contabilización antes de abrir realmente el señalamiento de la llamada de sistema, a fin de proporcionar un servicio de usuario más transparente.

En la figura 14.3 se muestra el mecanismo subyacente de las llamadas de sistema. El programa invocador ejecuta una instrucción que causa una interrupción de *software* o señalamiento. El controlador de señalamiento del núcleo se adueña del control, examina la solicitud, ejecuta el servicio solicitado y devuelve el resultado. Las llamadas de sistema se manejan de este modo para evitar que el código de usuario acceda directamente a las estructuras de datos del núcleo, pues podría dañarlas.

El señalamiento hace que el *hardware* cambie a un modo de operación distinto, llamado modo supervisor, con objeto de acceder a recursos del sistema que no están a disposición del usuario común. Desde el punto de vista del usuario, una llamada de sistema no es más que una llamada de función opaca. El programa se bloquea hasta que se ejecuta la llamada y reanuda su ejecución en el enunciado que sigue a la llamada.

Lo que se pretende con las llamadas de sistema es hacer que se parezcan lo más posible a las llamadas de función ordinarias. Si bien para el programa invocador los dos mecanismos se ven iguales, hay diferencias fundamentales. La llamada de sistema es ejecutada por un hilo distinto que opera con una pila diferente, mientras que la llamada de función es ejecutada por el hilo del programa invocador y utiliza la pila de éste. Dicho de otro modo, la llamada de sistema se *ejecuta a nombre del* programa invocador, mientras que una llamada de función ordinaria es *ejecutada por* el programa invocador.

Supongamos ahora que, en lugar de invocar una función que está en el núcleo, un programa invoca una función situada en el espacio de direcciones de otro proceso, que tal vez reside en otra máquina. Lógicamente, el programa debería efectuar la llamada y bloquearse hasta que la función regrese. Una operación así es una *llamada a un procedimiento remoto (RPC)*. En la figura 14.4 se muestran los hilos de ejecución para una RPC. Al igual que las llamadas de sistema, la llamada remota genera un nuevo hilo de ejecución (esta vez en el espacio de direcciones de un servidor remoto) y el invocador se bloquea hasta que el nuevo hilo termina.

En la figura 14.5 se presenta la preparación básica para la llamada a un procedimiento remoto. Un cliente realiza una llamada de función tal como efectuaría una llamada ordinaria y espera que el control regrese (esto se indica con *llamada lógica* y *regreso lógico* en la figura).

La llamada real es menos directa de lo que parece a primera vista. El programa cliente se compila con código adicional llamado *talón de cliente (client stube)*, formando un solo proceso. El talón de cliente (que desempeña un papel análogo a la envoltura de una llamada de sistema) se encarga de convertir los argumentos y formar con ellos un mensaje apropiado para transmitirse por la red. La conversión en un mensaje de red se conoce como *organización de los argumento (marshaling the arguments)*. Lo que hace en realidad el programa es invocar una función del talón de cliente, la cual básicamente es una envoltura que sirve para empacar los argumentos en la solicitud subyacente. El talón de cliente realiza la organización convirtiendo los argumentos a un formato independiente de la máquina a fin de que puedan participar

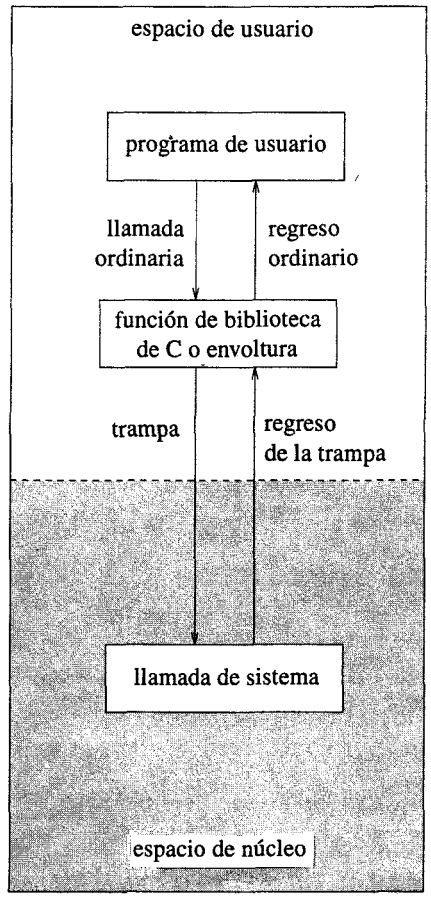


Figura 14.3: Una llamada de sistema genera un señalamiento que conduce al núcleo. El núcleo devuelve valores al programa invocador.

máquinas con diferentes arquitecturas. El formato estándar para la representación de datos independiente de la máquina se denomina formato XDR o de *representación externa de datos*. A continuación, el talón de cliente abre un señalamiento hacia las funciones de red del núcleo que se encargan de enviar un mensaje al servidor remoto, después de lo cual el talón de cliente se dedica a esperar un mensaje de respuesta.

Las funciones que se pueden invocar remotamente también se compilan con código adicional denominado *talón de servidor (server stub)*, el cual actúa como envoltura de las funciones del servidor. Cuando llega una solicitud de un cliente por la red, el núcleo del nodo remoto la pasa al talón de servidor en espera, el cual desorganiza los argumentos e invoca el servicio solicitado con una llamada de función ordinaria.

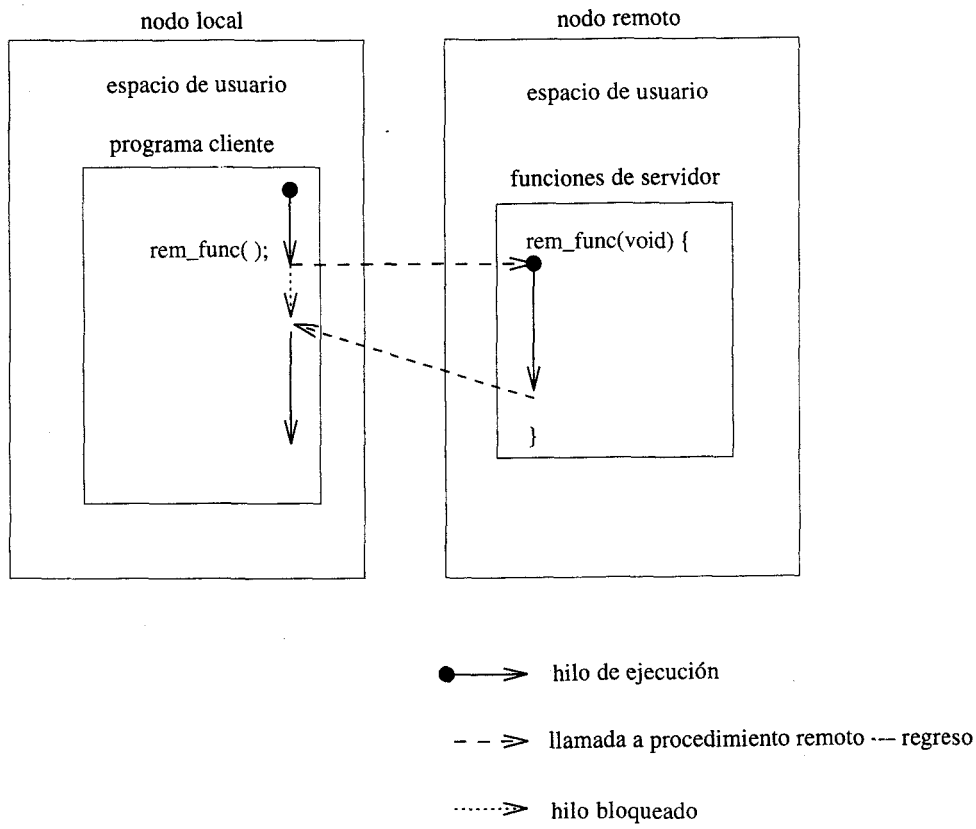


Figura 14.4: Hilo de ejecución de una llamada a un procedimiento remoto.

Cuando la llamada de función de servicio regresa, el talón de servidor organiza los valores devueltos para formar un mensaje de red apropiado y realiza una llamada de sistema al núcleo del servidor para solicitar la transmisión de la respuesta a través de la red hasta el nodo del cliente. El núcleo pasa el mensaje al talón de cliente en espera, el cual lo desorganiza y pasa al cliente como valor devuelto ordinario.

El mecanismo de RPC es transparente para el invocador. Lo único que ve el programa cliente es una llamada de función ordinaria al talón de cliente; la comunicación subyacente por la red queda oculta. En el lado del servidor, las funciones de servidor se invocan como funciones ordinarias porque el talón de servidor forma parte del proceso servidor. Los mecanismos subyacentes para transportar solicitudes y valores devueltos por la red se denominan *protocolo de transporte*. Las llamadas a procedimientos remotos se diseñan de modo que sean independientes del protocolo de transporte que se utiliza para prestar el servicio.

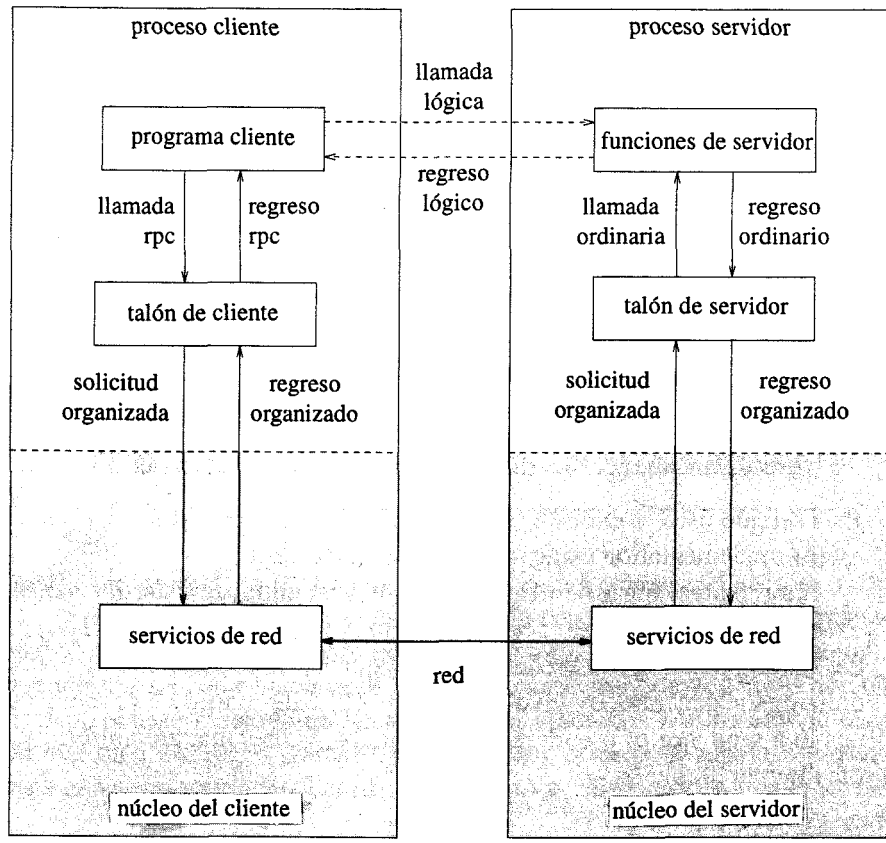


Figura 14.5: Protocolo para las llamadas a procedimientos remotos.

14.2 Conversión de una llamada local sencilla en una RPC

El enfoque más natural para desarrollar las llamadas remotas es depurar primero las funciones como funciones locales y luego analizar las alteraciones que sufren por la ejecución remota. Los dos aspectos principales de la conversión de una función local en una de invocación remota son establecer una asa para la función remota de modo que se invoque el servicio correcto y pasar los parámetros en una forma que pueda ser reconocida por diferentes tipos de nodos servidores.

En un mundo ideal, un programador podría crear un programa con llamadas locales, indicar cuáles llamadas deben ser remotas y hacer que el sistema genere el código de cliente y de servidor. Por desgracia, la generación de alto nivel de código para cliente y servidor todavía no es una realidad. Sun ofrece un comando llamado `rpcgen` que genera la versión remota a partir de un archivo de especificación cuyo nombre termina con `.x`. El programa `rpcgen` utiliza

información contenida en los prototipos de las funciones (tipo del valor devuelto y tipos de parámetros) y crea esqueletos de funciones en los que el programador inserta código. Los esqueletos indican cómo llamar a las funciones remotas y en qué forma dichas funciones devuelven sus valores.

En esta sección ilustraremos el proceso convirtiendo un servicio local sencillo que genera números pseudoaleatorios en un servicio remoto. El servicio se basa en la familia `drand48` de generadores de números pseudoaleatorios de la biblioteca estándar de UNIX.

SINOPSIS

```
#include <stdlib.h>

double drand48(void);
double erand48(unsigned short xsubi[3]);
void srand48(long seedval);
unsigned short *seed48(unsigned short seed16v[3]);
```

Spec 1170

En esta sección usaremos `srand48` y `drand48`, y en la siguiente presentaremos una implementación más robusta que utiliza `erand48` y `seed48`.

Antes de invocar a `drand48`, el programa debe inicializar un valor de partida llamando a la función `srand48` con un parámetro `long` llamado *semilla*. La semilla determina la posición de partida en una secuencia predeterminada de números pseudoaleatorios. Después de iniciar el generador invocando a `srand48`, se llama a `drand48` para que devuelva valores sucesivos en una secuencia de valores `double` pseudoaleatorios que están distribuidos uniformemente en el intervalo [0, 1).

Ejemplo 14.2

El siguiente segmento de código produce diez números pseudoaleatorios. La semilla del generador es 3243.

```
#include <stdio.h>
#include <stdlib.h>
int myseed;
int iters;
int i;

myseed = 3243;
iters = 10;
srand48(myseed);
for (i = 0; i < iters; i++)
    printf("%d : %f\n", i, drand48());
```

El diseño de un servicio local es el primer paso en el desarrollo de una RPC. El programa 14.1 presta tal servicio local de generación de números pseudoaleatorios. El servicio cuenta con dos funciones, `initialize_random` y `get_next_random`, que encapsulan las funciones `srand48` y `drand48`, respectivamente.

Programa 14.1: Servicio local para generar números pseudoaleatorios.

```
#include "rand.h"
void initialize_random(long seed)
{
    srand48(seed);
}

double get_next_random(void)
{
    return drand48();
}
```

Programa 14.1

El programa 14.2 invoca las funciones del programa 14.1. El valor de semilla y el número de iteraciones son argumentos de línea de comandos. El programa invoca el servicio `initialize_random` para inicializar el generador de números pseudoaleatorios subyacente y luego produce la cantidad especificada de números pseudoaleatorios llamando a `get_next_random`.

Programa 14.2: Programa que invoca un servicio local para generar números pseudoaleatorios.

```
#include <unistd.h>
#include <stdio.h>
#include "rand.h"

void main(int argc, char *argv[])
{
    int iters;
    int i;
    long myseed;

    if (argc != 3) {
       fprintf(stderr, "Usage: %s seed iterations\n", argv[0]);
       exit(1);
    }
    myseed = (long)atoi(argv[1]);
    iters = atoi(argv[2]);
    initialize_random(myseed);
    for (i = 0; i < iters; i++)
        printf("%d : %f\n", i, get_next_random());
    exit(0);
}
```

Programa 14.2

El programa 14.3 es el archivo del encabezado `rand.h` para el servicio. En el resto de esta sección explicaremos los pasos de la conversión de `initialize_random` y `get_next_random` en funciones remotas. El primer paso para convertir una función local en una remota es producir un archivo de especificación para el servicio remoto. Este archivo tiene la extensión `.x` y se escribe en el Lenguaje RPC de Sun, un lenguaje de especificación similar a C. Por fortuna, una plantilla estándar es suficiente para los servicios sencillos.

Programa 14.3: El archivo del encabezado `rand.h`.

```
#include <stdlib.h>
void initialize_random(long seed);
double get_next_random(void);
```

Programa 14.3

Una especificación de llamada de procedimiento remoto contiene tres números sin signo que identifican el programa, la versión y el procedimiento o función dentro del programa. (La terminología RPC utiliza procedimiento en lugar de función, pero el significado es el mismo.) Una autoridad central administra los números de programa. Algunos números de programa, incluidos los que se utilizan en los ejemplos de este capítulo, están disponibles para experimentar (pero véase la sección 14.6). El número de versión puede ser cualquier valor, pero por lo regular comienza con `1`. Cada vez que el servicio se actualice, escoja un nuevo número de versión para que las copias antiguas del servidor puedan estar activas mientras las versiones más nuevas se están probando. Las versiones antiguas de los clientes siguen comunicándose con los servidores antiguos mientras que los clientes nuevos utilizan el servidor nuevo. Un servidor puede encapsular varias funciones de invocación remota. La correspondencia entre las funciones y los números de procedimiento forman parte de la especificación del servicio.

El programa 14.4 es un ejemplo de archivo de especificación `rand.x` para el servidor remoto de números pseudoaleatorios `RAND_PROG` que se identifica con el número `0x31111111`. El número de versión del servidor, al que se hace referencia simbólicamente con `RAND_VERS`, es `1`. El servidor `RAND_PROG` exporta los servicios `initialize_random` y `get_next_random`.

Programa 14.4: El archivo de especificación `rand.x`.

```
/*      rand.x        */
program RAND_PROG {
   version RAND_VERS{
       void INITIALIZE_RANDOM(long) = 1;
       double GET_NEXT_RANDOM(void) = 2;
   } = 1;
} = 0x31111111;
```

Programa 14.4

Los nombres de función del programa 14.4 son los mismos que los de las funciones locales del programa 14.1, excepto que están en mayúsculas. La utilería `rpcgen` pasa estos nombres a minúsculas cuando forma los esqueletos de las funciones. Las funciones están numeradas, de modo que `initialize_random` es el servicio número `1` dentro del servidor `RAND_PROG` y la función `get_next_random` es el servicio número `2` dentro del servidor.

El programa `rcpgen` genera los archivos necesarios para crear el servicio remoto.

SINOPSIS

```
rpcgen infile
rpcgen [ -a ] [ -A ] [ -b ] [ -C ] [ -D name [ = value ] ]
      [ -i size ] [ -I [ -K seconds ] ] [ -L ]
      [ -M ] [ -N ] [ -T ] [ -Y pathname ] infile
rpcgen [ -c | -h | -l | -m | -t | -Sc | -Ss | -Sm ]
      [ -o outfile ] [ infile ]
rpcgen [ -s nettype ] [ -o outfile ] [ infile ]
rpcgen [ -n netid ] [ -o outfile ] [ infile ]
```

La opción `-C` indica que se utiliza ANSI C y la opción `-a` le indica a `rpcgen` que genere todos los archivos de apoyo. El archivo `infile` debe llevar una extensión `.x` para indicar que es un archivo de especificación.

Ejemplo 14.3

La figura 14.6 muestra los archivos que se generan cuando se ejecuta el siguiente comando. `proto.x` *es un archivo de especificación construido por el usuario.*

```
rpcgen -C -a proto.x
```

La opción `-a` de `rpcgen` solicita que se generen todos los archivos que se muestran en la figura 14.6. Si se omite esta opción, `rpcgen` sólo generará los archivos que están en los cuadros no sombreados. El programa `rpcgen` incorpora el nombre que precede a `.x` en el nombre de archivo de especificación como prefijo o sufijo de los nombres de los diversos archivos que genera. Los archivos de la figura 14.6 contienen la siguiente información para el servicio `proto.x`:

`makefile.proto`	Éste es el archivo *makefile* para compilar todo el código del cliente y el servidor.
`proto_clnt.c`	Este archivo contiene el talón de cliente, que normalmente no se modifica.
`proto_svc.c`	Este archivo contiene el talón de servidor, que por lo regular no se modifica.
`proto.h`	Este archivo de encabezado contiene todos los tipos XDR generados a partir de la especificación. Aquí puede verse cómo `rpcgen` convirtió los tipos definidos en el archivo `.x`.
`proto_client.c`	Este archivo contiene un esqueleto de programa principal del cliente con llamadas ficticias al servicio remoto. Inserte código

que prepare los valores de los argumentos para el servicio remoto antes de la llamada ficticia en el programa cliente.

`proto_server.c` Este archivo contiene los talones de los servicios remotos. Inserte el código para la versión local de los servicios en estos talones. Puede ser necesario modificar la forma en que estas funciones utilizan los parámetros.

`proto_xdr.c` Si se genera este archivo, contiene filtros XDR que el cliente y los talones de servidor necesitan. Este archivo normalmente no se modifica.

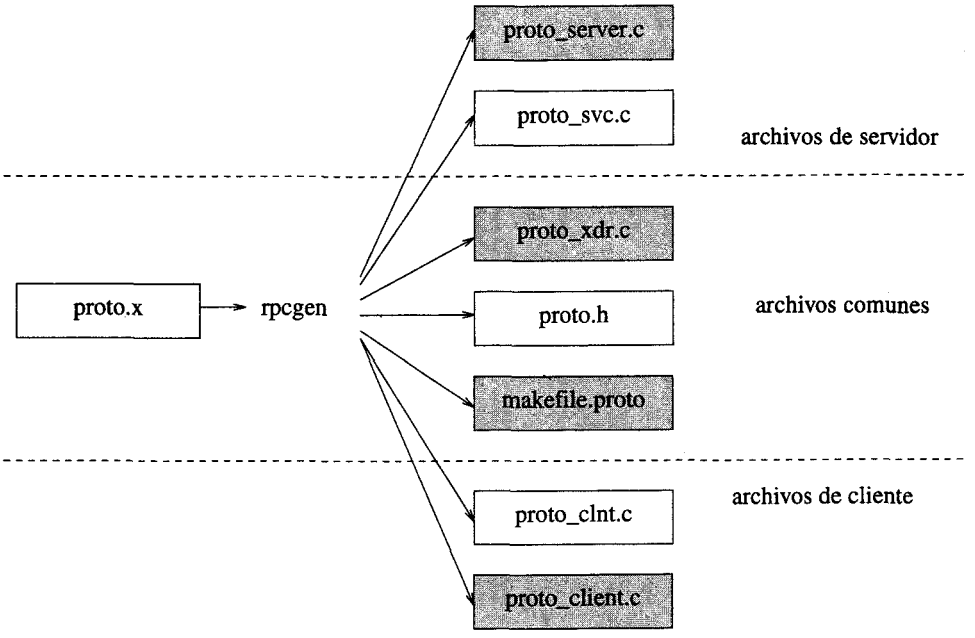


Figura 14.6: Archivos generados por `rpcgen` a partir del archivo `proto.x`. Los archivos sombreados se generan opcionalmente con la opción `-a` de `rpcgen`.

Ejemplo 14.4

El siguiente comando genera los programas de cliente y servidor para el servicio de números pseudoaleatorios a partir del archivo de especificación `rand.x` *del programa 14.4.*

```
rpcgen -C -a rand.x
```

Si desea convertir el servicio local de números pseudoaleatorios en un servicio remoto, siga estos pasos:

- Ejecute `rpcgen` para generar los archivos necesarios a partir del archivo de especificación `rand.x`.
- Modifique el archivo `rand_client.c` de modo que contenga el código del cliente.
- Modifique el archivo `rand_svc.c` de modo que contenga las funciones que se invocarán remotamente.

El programa 14.5 es el programa `rand_client.c` generado por `rpcgen`. El programa principal ficticio es sólo un vehículo para obtener el nombre del nodo de la línea de comandos. En nombre del nodo es la dirección de Internet en forma ASCII (por ejemplo, `vip.cs.utsa.edu`).

Programa 14.5: Programa `rand_client.c` generado por `rpcgen`.

```
/*
* Código de muestra generado por rpcgen. Estas sólo son
* plantillas que se pueden utilizar como pauta para
* crear funciones propias.
*/

#include "rand.h"


void
rand_prog_1(char *host)
{
CLIENT *clnt;
void  *result_1;
long  initialize_random_1_arg;
double  *result_2;
char *  get_next_random_1_arg;

#ifndef DEBUG
clnt = clnt_create(host, RAND_PROG, RAND_VERS, "netpath");
if (clnt == (CLIENT *) NULL) {
clnt_pcreateerror(host);
exit(1);
}
#endif /* DEBUG */

result_1 = initialize_random_1(&initialize_random_1_arg, clnt);
if (result_1 == (void *) NULL) {
clnt_perror(clnt, "call failed");
}
result_2 = get_next_random_1((void *)&get_next_random_1_arg, clnt);
if (result_2 == (double *) NULL) {
clnt_perror(clnt, "call failed");
```

```
}
#ifndef DEBUG
clnt_destroy(clnt);
#endif /* DEBUG */
}


main(int argc, char *argv[])
{
char *host;

if (argc < 2) {
printf("usage:  %s server_host\n", argv[0]);
exit(1);
}
host = argv[1];
rand_prog_1(host);
}
```

Programa 14.5

La parte más interesante del código del programa 14.5 está en la función de envoltura `rand_prog_1`. La llamada `clnt_create` genera una asa para el servicio remoto. Los parámetros `RAND_PROG` y `RAND_VERS` son los nombres de programa y de versión especificados en `rand.x`. El parámetro "`netpath`" indica que el programa debe buscar un mecanismo de transporte de red disponible según se especifica con la variable de entorno `NETPATH`. (Véase la explicación en la sección 14.6.) Si `clnt_create` falla, devuelve un apuntador `NULL`.

Las llamadas remotas convertidas que invocan a `initialize_random` y `get_next_random` llevan el número de versión como sufijo de los nombres de función, de modo que `initialize_random` se invoca con `initialize_random_1`. Otra diferencia entre las llamadas locales y las remotas es que los parámetros y valores de retorno se designan mediante apuntadores. De hecho, estos apuntadores se refieren a estructuras de datos definidas en el talón de cliente. El apuntador `clnt` devuelto por `clnt_create` es la asa para el servicio remoto y se utiliza como parámetro adicional en cada una de las llamadas a los procedimientos remotos. La memoria asignada a la asa `clnt` debe liberarse con `clnt_destroy` cuando el programa ya no necesite efectuar llamadas remotas.

El programa 14.6 es una modificación del programa `rand_client.c` que combina el programa principal del programa 14.2 con el programa 14.5. Comience con el programa original de servicio local del programa 14.2 e inserte la llamada a `create_client` cerca del principio del programa y la llamada a `clnt_destroy` al final. Ahora, el nombre del nodo se pasa como primer argumento de línea de comandos. El programa principal invoca directamente a las funciones remotas, así que no se necesita `rand_prog_1`.

El siguiente cambio implica la conversión de `initialize_random` y de llamadas `get_next_random` locales a llamadas remotas. Los nombres remotos tienen el sufijo `_1`

porque el número de versión es 1. Las funciones remotas pasan sus parámetros por apuntador y devuelven un apuntador al valor de retorno. La asa `clnt` se pasa como parámetro adicional en las llamadas. La conversión del lado del cliente ya está completa.

Programa 14.6: El programa `rand_client.c`.

```
#include <stdlib.h>
#include <stdio.h>
#include "rand.h"

void main(int argc, char *argv[])
{
   int iters, i;
   long myseed;
   CLIENT *clnt;
   void *result_1;
   double *result_2;
   char *arg;

   if (argc != 4) {
       fprintf(stderr, "Usage: %s host seed iterations\n", argv[0]);
       exit(1);
   }
   clnt = clnt_create(argv[1], RAND_PROG, RAND_VERS, "netpath");
   if (clnt == (CLIENT *) NULL) {
       clnt_pcreateerror(argv[1]);
       exit(1);
   }
   myseed = (long)atoi(argv[2]);
   iters = atoi(argv[3]);
   result_1 = initialize_random_1(&myseed, clnt);
   if (result_1 == (void *) NULL) {
      clnt_perror(clnt, "call failed");
   }
   for (i = 0; i < iters; i++) {
      result_2 = get_next_random_1((void *)&arg, clnt);
      if (result_2 == (double *) NULL) {
         clnt_perror(clnt, "call failed");
      }
      else
         printf("%d : %f\n", i, *result_2);
   }
   clnt_destroy(clnt);
   exit(0);
}
```

Programa 14.6

El código generado por `rpcgen` para `rand_server.c` se muestra en el programa 14.7. Los nombres de las funciones del servidor tienen como sufijos el número de versión y `_svc`. Los valores devueltos son estáticos porque se pasan mediante un apuntador al talón de servidor. Si estos valores no fueran estáticos, la memoria que ocupan se liberaría al regresar la llamada y el talón de servidor no tendría oportunidad de organizarlos.

Programa 14.7: El código de esqueleto de `rand_server.c` generado por el `rpcgen` del ejemplo 14.4.

```
/*
* Código de muestra generado por rpcgen. Éstas sólo son
* plantillas que se pueden utilizar como pauta para
* crear funciones propias.
*/

#include "rand.h"

void *
initialize_random_1_svc(long *argp, struct svc_req *rqstp)
{
static char * result;

/*
 * inserte aquí el código del servidor
 */

return((void *) &result);
}

double *
get_next_random_1_svc(void *argp, struct svc_req *rqstp)
{
static double  result;

/*
 * inserte aquí el código del servidor
 */

return (&result);
}
```

Programa 14.7

El programa 14.8 es el `rand_server.c` final, una vez que se ha insertado el código de las funciones `initialize_random` y `get_next_random` del programa 14.1 en los talones de

servidor ficticios de `rand_server.c`. La inclusión `#include <stdlib.h>` provee los prototipos de `srand48` y `drand48`.

Programa 14.8: Versión final del programa servidor para el servicio de números pseudoaleatorios.

```
#include <stdlib.h>
#include "rand.h"

void *
initialize_random_1_svc(long *argp, struct svc_req *rqstp)
{
   static char *result;
   srand48(*argp);
   result = (void *)NULL;
   return (void *) &result;
}

double *
get_next_random_1_svc(void *argp, struct svc_req *rqstp)
{
   static double  result;
   result = drand48();
   return &result;
}
```

Programa 14.8

Utilice el archivo creador generado por `rpcgen` para producir ejecutables para el cliente y el servidor.

Ejemplo 14.5

El siguiente comando produce los dos ejecutables `rand_client` *y* `rand_server` *para el cliente y el servidor, respectivamente.*

```
make -f makefile.rand
```

Ejemplo 14.6

El siguiente comando registra al servidor de números pseudoaleatorios.

```
rand_server
```

Una vez que se ha ejecutado el comando del ejemplo 14.6, el servicio `rand_server` queda registrado en el nodo actual y está listo para recibir solicitudes remotas. (En la sección 14.6 se explica cómo eliminar el servicio.)

Ejemplo 14.7

Suponga que el servidor del ejemplo 14.6 se ejecuta en el nodo con nombre de Internet `vip.cs.utsa.edu`. *El siguiente comando ejecuta el cliente con la semilla inicial* `4323` *y produce diez números pseudoaleatorios invocando el servicio de números pseudoaleatorios remoto.*

```
rand_client vip.cs.utsa.edu 4323 10
```

En síntesis, los pasos a seguir para convertir llamadas locales en remotas son:

- Logre que el programa funcione utilizando funciones locales.
- Reestructure las funciones de manera que cada una tenga un solo parámetro que se pasa por valor, y asegúrese de que funcionen cuando se invocan localmente.
- Cree un archivo de especificación con extensión `.x`.
- Invoque a `rpcgen` con las opciones `-a` y `-C` para generar un conjunto completo de archivos según se muestra en la figura 14.6.
- Antes de modificar los archivos generados, utilice el archivo *makefile* generado para compilarlos. En muchos casos es posible detectar errores en las definiciones de tipos del archivo de especificación durante este paso.
- Inserte el programa invocador en el archivo `_client.c` generado por `rpcgen`. (Aquí el nombre del archivo de especificación aparece antes de `_client.c`.)
- Inserte el código de la función local en el archivo `_server.c` generado por `rpcgen`.
- Trate de compilar los programas con el archivo *makefile* generado.
- Afine los programas fuente `_server.c` y `_client.c` hasta que funcionen. Dicha afinación no será necesaria si sólo se utilizan tipos de datos sencillos, pero las cosas no siempre son tan fáciles.

Como el servicio de números pseudoaleatorios contiene funciones cuando mucho con un parámetro, el cual no se modifica dentro de la función, no hay necesidad de reestructurar las funciones de modo que sólo reciban un parámetro. En la siguiente sección describiremos una versión más complicada del servicio de números pseudoaleatorios.

14.3 Un servicio remoto de números pseudoaleatorios mejorado

Un problema que presenta la implementación de la sección anterior es que otro cliente puede solicitar una reinicialización de la semilla en cualquier momento. Una estrategia para evitar la reinicialización arbitraria es agregar una bandera estática a `initialize_random_1` para indicar que el generador ya está inicializado. Entonces, la función `initialize_random_1` sólo invocará la función `srand48` si la bandera está despejada. Este enfoque evita que se llame a la función de semilla más de una vez, pero no aísla a los clientes individuales ni permite que un cliente reinicie con una semilla distinta. Todos los clientes dependen del valor inicial elegi-

do por el primer invocador de `initialize_random_1`. En esta sección desarrollaremos una estrategia más satisfactoria: una versión del servicio de números pseudoaleatorios capaz de proporcionar series independientes de números pseudoaleatorios. Los parámetros de invocación son más complicados, por lo que será necesario aplicar pasos de conversión adicionales al código generado por `rpcgen`.

La función `drand48` y similares generan números pseudoaleatorios utilizando el algoritmo de congruencia lineal y aritmética entera de `48` bits según la fórmula $X_{n+1} = (aX_n + c) \bmod m$, donde $n \geq 0$. Los valores por omisión (predeterminados o dados por *default*) de los demás parámetros son a = 0x5DEECE66D, c = 0xB y $m = 2^{48}$. Un *buffer* interno guarda el valor actual del entero de 48 bits X_n, de modo que el siguiente valor de X_{n+1} se puede calcular directamente a partir del anterior. La función `drand48` extrae los bits apropiados de X_{n+1} y devuelve un número pseudoaleatorio convertido. Lo normal es que el invocador proporcione un valor de semilla `long` con el cual `srand48` produce el X_0 inicial. La página del manual indica que el método actual para hacer esto es copiar el valor de semilla en los 32 bits superiores de X_0 e insertar el valor arbitrario 0x330E16 en los 16 bits más bajos para producir un valor de 48 bits.

En una forma alternativa del generador de números pseudoaleatorios, `erand48`, el invocador proporciona el valor de X_n en un arreglo de tres valores short sin signo. La función `erand48` llena el arreglo con el nuevo valor de X_n de 48 bits y devuelve el número pseudoaleatorio. De este modo, `erand48` no necesita utilizar variables estáticas y puede servir para generar múltiples series independientes de números pseudoaleatorios. Para obtener un valor de X_n inicial a partir del valor de semilla, invoque a `srand48` con la semilla deseada para establecer el valor X_n estático interno del generador de números pseudoaleatorios. La función `seed48` obtiene una semilla a partir de un valor de X_n y devuelve un apuntador al valor anterior de X_n. Invoque a `seed48` con cualquier argumento, pero guarde el valor devuelto a fin de encontrar el valor de X_n asociado a la primera llamada de `srand48`.

Ejemplo 14.8

El siguiente segmento de código muestra cómo generar el valor X_0 interno sin tener conocimiento del mecanismo exacto empleado para el cálculo.

```
#include <stdlib.h>

void initialize_random(long seed, unsigned short xsubi[3])
{
    unsigned short *xp;
    srand48(seed);
    xp = seed48(xsubi);
    xsubi[0] = *xp;
    xsubi[1] = *(xp + 1);
    xsubi[2] = *(xp + 2);
}

double get_next_random(unsigned short xsubi[3])
{
    return erand48(xsubi);
}
```

Los valores X_n internos se guardan en un arreglo de tres valores `unsigned short`. La llamada inicial a `srand48` en el ejemplo 14.8 establece los valores del arreglo de semilla interno. La llamada a `seed48` restablece la semilla a otro valor, pero devuelve el arreglo de semilla previamente establecido. Este valor devuelto es el punto de partida para las llamadas a `erand48`.

La versión de `get_next_random` del ejemplo 14.8 devuelve el número pseudoaleatorio y establece el siguiente valor de X_n. El mecanismo de invocación estándar a la usanza antigua de Sun requiere que toda función remota tenga un solo parámetro y que éste no se utilice para devolver información al programa invocador. (En fechas recientes se agregó a `rpcgen` de Sun Solaris una opción para pasar múltiples parámetros.) Toda la información se devuelve a través del valor de retorno, así que en esta sección ilustraremos la reestructuración de `initialize_random` y `get_next_random` para ajustarlas al formato de un solo parámetro como paso intermedio. Para facilitar las cosas, definiremos una estructura de tipo `struct randpack` para contener tanto el entero interno de 48 bits como el `double` que es el valor pseudoaleatorio que nos interesa. El programa 14.9 es el `rand.h` modificado.

Programa 14.9: El archivo de encabezado `rand.h` para servicio local.

```
#include <stdlib.h>
struct randpack {
    double pseudo;
    unsigned short xi[3];
};
struct randpack initialize_random(long seed);
struct randpack get_next_random(struct randpack p);
```

Programa 14.9

El programa 14.10 mustra las funciones locales después de la conversión al formato de un solo parámetro. La función `initialize_random` modificada acepta la semilla `long` original como parámetro y devuelve una estructura de tipo `struct randpack` que contiene el valor de 48 bits que se pasará a `erand48`. La función `get_next_random` acepta una estructura de tipo `struct randpack` que contiene el valor de 48 bits, y devuelve el número pseudoaleatorio y el nuevo valor de 48 bits en una `struct randpack`.

Programa 14.10: Servicio local para generar series independientes de números pseudoaleatorios.

```
#include "rand.h"

struct randpack initialize_random(long seed)
{
```

```
    struct randpack result;
    unsigned short *xp;
    srand48(seed);
    xp = seed48(result.xi);
    result.xi[0] = *xp;
    result.xi[1] = *(xp + 1);
    result.xi[2] = *(xp + 2);
    return result;
}

struct randpack get_next_random(struct randpack p)
{
    struct randpack result;
    double rpseudo;
    rpseudo = erand48(p.xi);
    result = p;
    result.pseudo = rpseudo;
    return result;

}
```

Programa 14.10

El programa 14.11 es el programa principal para el servicio local de números pseudoaleatorios. Cada llamada a `get_next_random` restablece la semilla interna del generador `drand48`, de modo que el resultado no depende de información de estado conservada de llamadas anteriores.

Programa 14.11: Programa principal que ilustra el empleo del nuevo servicio local de números pseudoaleatorios.

```
#include <stdio.h>
#include "rand.h"
void main(int argc, char *argv[])
{
    int iters, i;
    long myseed;
    unsigned short xi[3];
    struct randpack next_seed;

    if (argc != 3) {
       fprintf(stderr, "Usage: %s seed iterations\n", argv[0]);
       exit(1);
    }

    myseed = (long)atoi(argv[1]);
    iters = atoi(argv[2]);
    next_seed = initialize_random(myseed);
```

```
    for (i = 0; i < iters; i++)  {
       next_seed = get_next_random(next_seed);
       printf("%d : %f\n", i, next_seed.pseudo);
    }
    exit(0);
}
```

Programa 14.11

Ejemplo 14.9

El archivo de especificación del programa 14.12 describe un servicio remoto para generar series independientes de números pseudoaleatorios. El siguiente comando genera los archivos de esqueleto para el servicio.

```
rpcgen -C -a rand.x
```

Programa 14.12: Archivo de especificación para un servicio de números pseudoaleatorios independientes.

```
/*      rand.x        */
struct randpack {
    double pseudo;
    unsigned short xi[3];
};

program RAND_PROG {
    version RAND_VERS{
        randpack INITIALIZE_RANDOM(long) = 1;
         randpack GET_NEXT_RANDOM(randpack) = 2;
    } = 2;
} = 0x31111111;
```

Programa 14.12

Ejemplo 14.10

La salida de `rpcgen` *para el ejemplo 14.9 contiene la siguiente traducción de la* `struct randpack` *del archivo* `rand.h`.

```
struct randpack {
       double pseudo;
       u_short xi[3];
};
typedef struct randpack randpack;
```

La traducción de `struct randpack` que hace `rpcgen` es `typedef struct randpack randpack`, así que debe evitarse el empleo de `typedef` en la definición original en `rand.x`.

El programa 14.13 es la versión final de `rand_client.c`. Los principales cambios respecto de la llamada local son el empleo de una asa de cliente `clnt` y el empleo de apuntadores

para los parámetros y los valores de retorno. El número de versión es 2, así que el programa cliente llama a los servicios con `initialize_random_2` y `get_next_random_2`, respectivamente.

Programa 14.13: Cliente final para un servicio remoto de series independientes de números pseudoaleatorios.

```
#include <stdlib.h>
#include <stdio.h>
#include "rand.h"

void main(int argc, char *argv[])
{
    int iters, i;
    long myseed;
    CLIENT *clnt;
    randpack next_seed;
    randpack  *result;
    char *host;

    if (argc != 4) {
       fprintf(stderr, "Usage: %s host seed iterations\n", argv[0]);
       exit(1);
    }
    host = argv[1];
    clnt = clnt_create(host, RAND_PROG, RAND_VERS, "netpath");
    if (clnt == (CLIENT *) NULL) {
        clnt_pcreateerror(host);
        exit(1);
    }
    myseed = (long)atoi(argv[2]);
    iters = atoi(argv[3]);
    result = initialize_random_2(&myseed, clnt);
    if (result == (randpack *) NULL) {
        clnt_perror(clnt, "cannot initialize generator");
        exit(1);
    }
    next_seed = *result;
    for (i = 0; i < iters; i++) {
        result = get_next_random_2(&next_seed, clnt);
        if (result == (randpack *) NULL)
           clnt_perror(clnt, "call for next number failed");
        printf("%d : %f\n", i, result->pseudo);
        next_seed = *result;
    }
    exit(0);
}
```

Programa 14.13

El programa 14.14 es el servidor remoto para el cliente del programa 14.13. Las llamadas a `initialize_random_2_svc` y `get_next_random_2_svc` no utilizan información estática y son independientes entre sí.

Programa 14.14: Versión final de las funciones de servidor para un servicio remoto que genera series independientes de números pseudoaleatorios.

```
#include <stdlib.h>
#include "rand.h"

randpack *
initialize_random_2_svc(long *argp, struct svc_req *rqstp)
{
    static randpack  result;
    unsigned short *xp;

    srand48(*argp);
    xp = seed48(result.xi);
    result.xi[0] = *xp;
    result.xi[1] = *(xp + 1);
    result.xi[2] = *(xp + 2);
    return &result;
}

randpack *
get_next_random_2_svc(randpack *argp, struct svc_req *rqstp)
{
    static randpack result;
    double rpseudo;

    rpseudo = erand48(argp->xi);
    result = *argp;
    result.pseudo = rpseudo;
    return &result;
}
```

Programa 14.14

Ejercicio 14.1

Experimente con el servicio remoto de números pseudoaleatorios. Examine el código de `rand_clnt.c` y `rand_svc.c`, que son los talones para el cliente y el servidor, respectivamente. Trate de entender el código generado por `rpcgen`. (Los aspectos complejos están principalmente en las funciones de conversión de los parámetros a XDR.)

14.4 Estado del servidor y solicitudes equipotentes

Las llamadas a procedimientos remotos están diseñadas de modo que se asemejen lo más posible a las llamadas locales, pero la ejecución de llamadas remotas presenta ciertas complicaciones inevitables. Las explicaciones de esta sección se basan en el ejemplo de escribir cierto número de bytes en un archivo. A fin de simplificar el análisis, suponga que las llamadas locales y remotas presentan exactamente el mismo aspecto y concéntrese en el efecto de una llamada local en contraposición al de una llamada remota.

Ejemplo 14.11

La siguiente función `write_file` *escribe* `nbytes` *bytes de* `buf` *en el archivo designado por* `fd`. *La función devuelve el número de bytes escritos si la operación tiene éxito o* `-1` *en caso contrario.*

```
int write_file(int fd, char *buf, int nbytes)
{
    return put_block(fd, nbytes, buf);
}
```

La función `put_block` del ejemplo 14.11 es una versión remota de la llamada `write` ordinaria; es decir, `put_block(fd, nbytes, buf)` equivale a `write(fd, buf, nbytes)`, excepto que se ejecuta en un nodo remoto. El orden de los parámetros en `put_block` es diferente del orden en `write_file` porque `write_file` es similar a la llamada `write` ordinaria, mientras que `put_block` es análoga a la llamada remota en NFS (*Network File System*, sistema de archivos de red).

Cuando un programa ejecuta una llamada `write` ordinaria, el descriptor de archivo hace referencia a una entrada de la tabla local de descriptores de archivo del proceso, y la llamada `write` hace que se actualice la posición en la tabla de archivos del sistema en el nodo del programa invocador.

Cuando `write_file` llama a `put_block`, el equivalente remoto de `write`, el descriptor de archivo hace referencia a una entrada en la tabla de descriptores de archivo del servidor. La entrada correspondiente en la tabla de descriptores de archivo remota apunta a una entrada en la tabla de archivos del sistema del nodo remoto, como se aprecia en la figura 14.7. La función `put_block` actualiza la posición en el archivo en la entrada de la tabla de archivos del sistema del nodo servidor.

El servicio remoto del ejemplo 14.11 funciona satisfactoriamente cuando la red es confiable y ninguno de los participantes se cae. Las llamadas remotas se complican por el hecho de que un cliente o un servidor pueden fallar de forma independiente. No siempre es posible saber si una falta de respuesta se debe a un fracaso de red o a un nodo caído. En contraste, las llamadas locales se entregan de forma confiable. Si se cae el sistema operativo del nodo, tanto el cliente como el servidor terminarán.

Consideremos la situación en la que un cliente emite una solicitud pero no recibe respuesta. Después de cierto lapso, el cliente reintenta la solicitud. ¿Qué sucede? La respuesta depende de la razón por la cual el cliente no recibió respuesta. Entre las posibilidades están:

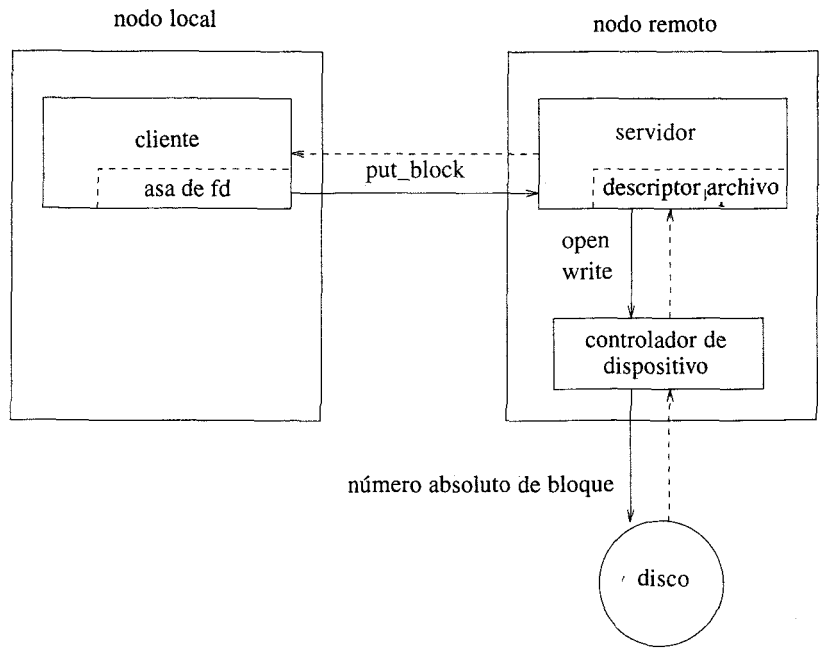


Figura 14.7: La función `put_block` remota hace referencia a un descriptor de archivo en el servidor.

- La solicitud inicial se perdió en la red.
- El servidor recibió la solicitud y la atendió, pero la respuesta se perdió.
- El servidor ya no está activo.

Si se perdió la solicitud inicial, el cliente puede volver a intentarla sin consecuencias adversas. Por otro lado, si el servidor atendió la solicitud original, pero la respuesta se perdió, el reintento producirá un resultado incorrecto porque la posición en el archivo del servidor se actualiza cada vez que se invoca `put_block`. Una posible solución sería identificar cada solicitud con un número de secuencia. El servidor mantiene un registro del número de secuencia de la última solicitud que atendió y del resultado de la última solicitud. Si llega una solicitud repetida, el servidor volverá a enviar la respuesta en lugar de volver a ejecutar el servicio solicitado. El empleo de números de secuencia no resuelve el problema de una caída del servidor, ya que los números de secuencia se pierden cuando hay una caída.

Una estrategia para resolver ambos problemas, el de respuesta perdida y el de servidor caído, consiste en disponer las cosas de modo que la misma solicitud pueda volverse a ejecutar sin alterar los resultados. Se dice que una solicitud de este tipo es *equipotente (idempotent)*. La función `put_block` del ejemplo 14.11 no es equipotente porque la escritura actualiza la posición en el archivo dentro de la tabla de archivos del sistema remota. Si un cliente repite la misma solicitud, el servidor utilizará el valor actualizado de la posición en el archivo. La segunda llamada hará que se escriba la misma información inmediatamente después del final

del bloque que se escribió con la solicitud anterior. Una forma de resolver este problema es hacer que el cliente mantenga un registro de su propia posición dentro del archivo.

Ejemplo 14.12

La siguiente versión equipotente de `put_block` *ajusta la posición dentro del archivo a un punto especificado antes de escribir la información.*

```
int put_block(int file, int offset, int count, char *data)
{
   int returncode = 0;
   if (lseek(file, offset, SEEK_SET) == -1)
       returncode = -1;
   else
       returncode = write(file, data, count);
   return returncode;
}
```

La función `put_block` del ejemplo 14.12 es equipotente en el sentido de que un cliente puede repetir la llamada y hacer que escriba los mismos datos en el mismo lugar del archivo. Desde luego, la repetición de la llamada cambia la hora de modificación del archivo, pero por ahora haremos caso omiso de esta complicación.

Para invocar la función `put_block` del ejemplo 14.12 como llamada remota, el cliente mantiene un registro de la posición en el archivo en la que desea escribir, y actualiza dicho registro cuando recibe la confirmación de que `put_block` tuvo éxito.

Ejemplo 14.13

La función `write_file` *es una envoltura para una llamada remota equipotente* `put_block` *del ejemplo 14.12.*

```
static int fd_offset = 0;
int write_file(int fd, char *buf, int nbytes)
{
    int bytes_written;
    if ((bytes_written =
        put_block(fd, fd_offset, nbytes, buf)) != -1)
        fd_offset += bytes_written;
    return bytes_written;
}
```

La variable de cliente `fd_offset` del ejemplo 14.13 guarda la posición en el archivo de la última solicitud ejecutada con éxito. Antes de invocar a `write_file`, el programa debe abrir remotamente el archivo para obtener un manejador que utilizará en las solicitudes subsecuentes. En el ejemplo 14.13 se utiliza el manejador entero `fd`, pero en una implementación remota real el manejador podría ser una estructura opaca.

La apertura remota crea una entrada en la tabla de archivos del sistema en el nodo remoto. Si el archivo no fue abierto por otros procesos, también se crea una nueva entrada en la tabla de

inodos en memoria. Las llamadas `lseek` y `write` de `put_block` actualizan la posición almacenada en la entrada de la tabla de archivos del sistema del nodo remoto creada por la apertura de archivo remota.

La función `put_block` del ejemplo 14.12 no resuelve por completo el problema del servidor caído. Consideremos la forma en que `put_block` se ejecuta en un servidor remoto. El descriptor de archivo `file` hace referencia a una entrada de la tabla de descriptores de archivo del servidor, y el cliente debe abrir el archivo y recibir el manejador antes de emitir una solicitud remota. Cada llamada `put_block` del cliente actualiza la posición de archivo local del servidor. El servidor conserva el descriptor de archivo y su correspondiente entrada en la tabla de archivos del sistema hasta que el cliente le solicita explícitamente que cierre el archivo. Si el servidor se cae y luego se reinicia, no tendrá forma de saber cuál archivo debe volver a abrir cuando reciba la siguiente solicitud `put_block` del cliente.

Las caídas del cliente también representan un problema. Los archivos remotos no se cierran automáticamente como parte del procedimiento de terminación `exit` del cliente. Si el cliente olvida cerrar el archivo remoto o se cae antes de cerrarlo, el servidor se quedará con una entrada pendiente en su tabla de descriptores de archivo. Durante periodos de operación largos, estos descriptores de archivo pendientes podrían hacer que el servidor falle al agotarse el espacio libre en su tabla de descriptores. En tal situación, el problema se debe a que la solicitud del cliente hace que el servidor conserve información de estado (un descriptor de archivo abierto). Incluso si las solicitudes son equipotentes, una caída del cliente puede dar pie a información de estado no válida o innecesaria en el servidor.

En la figura 14.8 se ilustra una estrategia equipotente y sin estados para la escritura remota. La función `put_block` abre el archivo, busca la posición apropiada, escribe el bloque y cierra el archivo. El cliente mantiene un registro local de la posición en el archivo para determinar la posición de búsqueda de las llamadas.

Ejemplo 14.14

La siguiente versión de `put_block` *es autosuficiente. Si se invoca desde un nodo remoto, no deja ningún estado en el servidor.*

```
int put_block(char *fname, int offset, int count, char *data)
{
    int returncode = 0;
    int file;
    if ((file = open(fname, O_WRONLY|O_CREAT, 0600)) == -1)
       returncode = -1;
    else if (lseek(file, offset, SEEK_SET) == -1)
       returncode = -1;
    else
       returncode = write(file, data, count);
    close(file);
    return returncode;
}
```

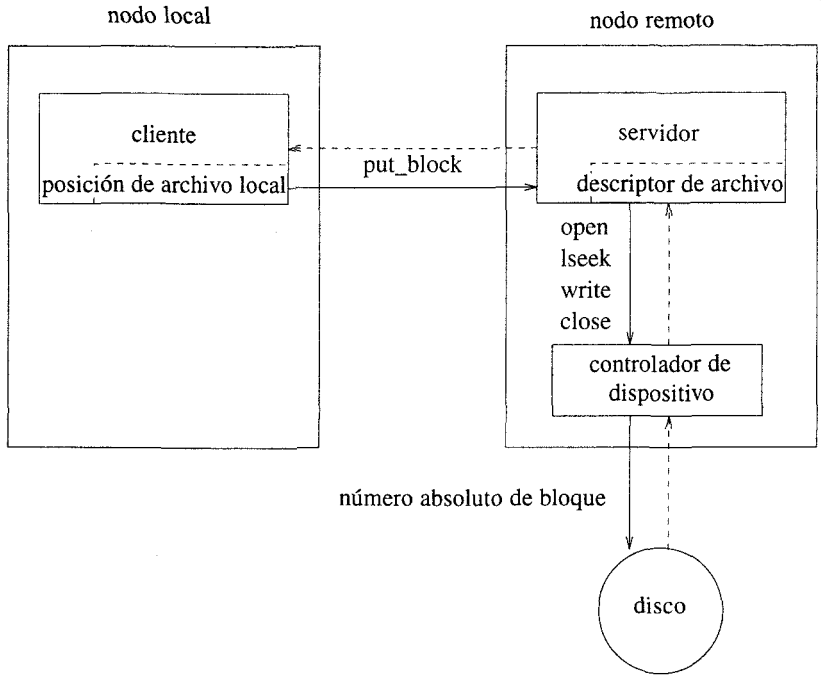


Figura 14.8: Versión equipotente y sin estados de `put_block`.

El cliente conserva la posición en el archivo localmente y la actualiza cuando recibe una confirmación de que el servidor atendió con éxito la solicitud. Al servidor no le importa si el cliente se cae, pues no conserva información de estado generada por las solicitudes de los clientes. El cliente no puede distinguir entre una respuesta lenta y un servidor caído. Como las solicitudes son equipotentes, el cliente puede seguir reintentando la solicitud hasta tener éxito. Si el sistema remoto vuelve a arrancar y el servidor se reinicia, la solicitud reintentada por el cliente llegará al nuevo servidor.

En un servidor *sin estados* las solicitudes de los clientes son autosuficientes y no dejan información de estado residual en el servidor. Los servidores sin estados resisten bien las caídas. No todos los problemas se pueden plantear en una forma sin estados, pero hay varios ejemplos bien conocidos de servidores sin estados. El sistema de archivos de red de Sun, NFS, se implementa como servidor sin estados RPC, como se verá en la sección 14.8.

14.5 Servicio de archivos equipotente remoto

En esta sección desarrollaremos una versión remota de la función `put_block` del ejemplo 14.14. También ilustraremos algunas de las complicaciones que pueden surgir en la conversión a XDR.

El programa 14.15 es la especificación de una versión sin estados del servicio remoto `put_block`. Recuérdese que las especificaciones no se escriben en C, sino en un lenguaje de especificación especial similar a C. La `string` del programa 14.15 es un tipo incluido en `rpcgen`. El uso de `< >` en lugar de `[ ]` indica cadenas o arreglos de longitud variable.

Programa 14.15: Especificación de un servicio de archivos remoto sin estados.

```
/*    rfile.x              */
const MAX_BUF = 1024;
const MAX_STR = 256;

struct packet {
    string fname<MAX_STR>;
    int count;
    int offset;
    char  data<MAX_BUF>;
};

program RFILEPROG {
    version RFILEVERS {
        int PUT_BLOCK(packet) = 1;
    } = 1;
} = 0x31111112;
```

Programa 14.15

Ejemplo 14.15

El siguiente comando crea los archivos de esqueleto.

```
rpcgen -C -a rfile.x
```

El programa `rpcgen` *traduce* `struct packet` *al siguiente código en C del archivo de encabezado* `rfile.h`.

```
#define MAX_BUF 1024
#define MAX_STR 256

struct packet {
    char *fname;
    int offset;
    int count;
    struct {
        u_int data_len;
        char *data_val;
    } data;
};
typedef struct packet packet;
```

Observe que `rpcgen` traduce una cadena, como el miembro `fname` en el programa 14.15, a un apuntador en lugar de un arreglo fijo. El arreglo de longitud variable `data` se convierte en una estructura que tiene un apuntador a `char` y un miembro que indica la longitud. El ejemplo subraya la importancia de estudiar el archivo de cabecera generado por `rpcgen` antes de insertar código en los archivos de esqueleto del cliente y el servidor.

El programa 14.16 es la versión final de `rfile_server.c`. Cada llamada al servicio es autosuficiente. El programa 14.17 es el cliente correspondiente.

Programa 14.16: Versión final del `rfile_server.c`.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include "rfile.h"
#define PERMS S_IRUSR | S_IWUSR

int *
put_block_1_svc(packet *pack, struct svc_req *rqstp)
{
   static int  result;
   int file;
   result = 0;
   if ((file = open(pack->fname, O_WRONLY|O_CREAT, PERMS)) == -1)
      result = -1;
   else if (lseek(file, pack->offset, SEEK_SET) == -1)
      result = -1;
   else
      result = write(file, pack->data.data_val, pack->count);
   close(file);
   return &result;
}
```

Programa 14.16

Programa 14.17: Código del cliente para el programa 14.16.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include "rfile.h"
static int fd_offset = 0;

int write_file(CLIENT *clnt, char *filename, char *buf, int nbytes)
{
    int  bytes_written;
```

```
    struct packet pack;
    int *result;
    pack.count = nbytes;
    pack.offset = fd_offset;
    pack.fname = filename;
    pack.data.data_val = buf;
    pack.data.data_len = nbytes;
    result = put_block_1(&pack, clnt);
    if (result == (void *)NULL)
       bytes_written = -1;
    else{
       bytes_written = *result;
       fd_offset += bytes_written;
    }
    return bytes_written;
}

void main(int argc, char *argv[])
{
    int i, blocks, block_size;
    char stuff[256];
    CLIENT *clnt;
    char *host;

    if (argc != 4) {
       fprintf(stderr,"Usage: %s host filename blocks\n", argv[0]);
       exit(1);
    }

    host = argv[1];
    clnt = clnt_create(host, RFILEPROG, RFILEVERS, "netpath");
    if (clnt == (CLIENT *) NULL) {
       clnt_pcreateerror(host);
       exit(1);
    }

    blocks = atoi(argv[3]);
    sprintf(stuff, "This is a test\n");
    block_size = strlen(stuff);
    for (i = 0; i < blocks; i++) {
        if (write_file(clnt, argv[2], stuff, block_size) <= 0) {
           fprintf(stderr, "Error in writing file\n");
           break;
        }
    }
    clnt_destroy(clnt);
    exit(0);
}
```

Programa 14.17

14.6 Vinculación y asignación de nombres a servicios

Cada mensaje RPC tiene tres números sin signo que identifican el programa, la versión y el número de procedimiento dentro del programa. La convención de Sun Microsystems para los números de programa es

`00000000 - 1fffffff`	Definidos y administrados por Sun
`20000000 - 3fffffff`	Definidos por el usuario (para servicios de depuración)
`40000000 - 5fffffff`	Transitorios (aplicaciones que generan números dinámicamente)
`60000000 - ffffffff`	Reservados para uso futuro

Se permiten múltiples versiones del mismo servicio para que los servicios puedan modernizarse y los clientes que utilizan versiones antiguas puedan seguir funcionando. Los servidores que están escuchando en puertos como servicios RPC deben registrarse con `rpcbind`. El código de registro se incluye automáticamente en el talón de servidor `_svc.c` generado por `rpcgen`. Utilice `rpcinfo` para averiguar cuáles servicios RPC están disponibles en un nodo dado.

SINOPSIS

```
rpcinfo [ -m ] [ -s ] [ host ]
rpcinfo -p [ host ]
rpcinfo -T transport host prognum [ versnum ]
rpcinfo -l [ -T transport ] host prognum [ versnum ]
rpcinfo [ -n portnum ] -u host prognum [ versnum ]
rpcinfo [ -n portnum ] -t host prognum [ versnum ]
rpcinfo -a serv_address -T transport prognum [ versnum ]
rpcinfo -b [ -T transport ] prognum versnum
rpcinfo -d [ -T transport ] prognum versnum
```

Ejemplo 14.16

El comando

```
rpcinfo -s
```

produjo la siguiente salida parcial en cierta máquina:

```
program version(s)       netid(s)                    service  owner
100000  2,3,4  udp,tcp,ticlts,ticotsord,ticots  rpcbind  superuser
100029  2,1    ticots,ticotsord,ticlts          keyserv  superuser
100078  4      ticots,ticotsord,ticlts          kerbd    superuser
100087  10     udp                              admind   superuser
100011  1      ticlts,udp                       rquotad  superuser
100002  3,2    ticlts,udp                       rusersd  superuser
100099  1      ticots,ticotsord,ticlts          -        superuser
100012  1      ticlts,udp                       sprayd   superuser
100008  1      ticlts,udp                       walld    superuser
```

```
100001  4,3,2  ticlts,udp                          rstatd   superuser
100024  1      ticots,ticotsord,ticlts,tcp,udp     status   superuser
100021  2,3,1  ticots,ticotsord,ticlts,tcp,udp     nlockmgr superuser
100068  4,3,2  udp                                 -        superuser
100020  2      ticots,ticotsord,ticlts,tcp,udp     llockmgr superuser
100083  1      tcp,udp                             -        superuser
1342177279  1,2    tcp                                -        5001
```

Cada servicio RPC tiene un número de programa y un número de versión que lo identifican. El campo `netid` (identificador de red) de la salida de `rpcinfo` indica el proveedor de transporte subyacente. Los dos mecanismos de transporte que suelen manejarse son UDP y TCP. UDP es un protocolo sin conexiones que no garantiza la entrega libre de errores. TCP es un protocolo orientado a conexiones que ofrece una entrega en orden libre de errores. Los identificadores `TICOTS`, `TICOTSORD` y `TICLTS` designan protocolos diseñados para utilizarse en la misma máquina y se caracterizan como *proveedores de transporte de retorno*. `TICLTS` es análogo a UDP, mientras que `TICOTS` es un proveedor de transporte orientado a conexiones análogo a TCP. `TICOTSORD` es un proveedor orientado a conexiones que tiene un mecanismo de liberación ordenado.

Ejemplo 14.17

El siguiente comando elimina la versión `1` *del servicio* `1342177279` *perteneciente al usuario* `5001` *del listado del ejemplo 14.16.*

```
rpcinfo -d 1342177279 1
```

El nuevo mecanismo TI-RPC promueve la escritura de código que sea independiente respecto del transporte. Un servicio dado puede contar con varios transportes diferentes. En las llamadas a procedimientos remotos independientes del transporte, el cliente no indica explícitamente cuál transporte se debe usar. La función `clnt_create` (crear cliente) prueba los transportes en el orden especificado por la variable de entorno `NETPATH`.

El servicio `rpcbind` sirve para registrar los servicios remotos provistos por un nodo específico. Los clientes invocan a `clnt_create` para ponerse en contacto con el servicio `rpcbind` durante la ejecución con el fin de obtener la dirección del servicio. Cada transporte que se maneje debe tener una noción especificada de una *dirección bien conocida* que se representa en un formato de *dirección universal* (*Universal Address*). La dirección universal de un transporte dado es una cadena en un formato previamente determinado. Si se consulta `rpcbind`, devolverá la dirección en el formato de dirección universal. Los talones de cliente y de servidor cuentan con funciones para traducir la cadena de dirección universal a una estructura de dirección TLI local específica para un transporte. Por tanto, todo transporte que se desee usar deberá tener una interfaz TLI en el nodo local. (En el capítulo 12 se describe el mecanismo de direccionamiento de TLI.)

`rpcbind` permite al cliente acceder a un servicio especificando el número de programa de 32 bits y devuelve la información de puerto local. En contraste, un cliente TLI ordinario debe conocer el número de puerto bien conocido de 16 bits del servidor. Puesto que el número de puerto puede ser cualquier número convenido entre `rpcbind` y el servidor, la posibilidad de conflictos de puerto es considerablemente menor que con una implementación que requiere la

especificación directa de puertos bien conocidos. Además de tener más bits, estos números de servicio de RPC son administrados por una autoridad central, a diferencia de los números de puerto para *sockets* o TLI.

Ejemplo 14.18

El siguiente valor de `NETPATH` *indica que el cliente debe probar primero UDP, luego TCP y por último TP4.*

```
udp:tcp:tp4
```

Si `NETPATH` no tiene valor, la aplicación utilizará por omisión los transportes visibles especificados en el archivo `netconfig` del sistema. Aunque podría parecer que la selección del transporte no es crucial, sí afecta la forma en que se manejan los fracasos, como veremos en la siguiente sección.

14.7 Fracasos

Los fracasos presentan un problema delicado para los diseñadores de llamadas remotas. A diferencia de las llamadas locales, el fracaso de un cliente y el fracaso de un servidor no siempre son simultáneos. Si un programa cliente intenta una llamada a un procedimiento remoto y no recibe respuesta, hay varias posibilidades:

- La red está lenta y el cliente no esperó lo suficiente.
- Se perdió el mensaje inicial.
- El servidor recibió el mensaje, atendió la solicitud y luego se cayó.
- El servidor prestó el servicio pero se perdió la confirmación.

En vista de la posibilidad de una caída remota, la mayor parte de las implementaciones de llamadas a procedimientos remotos tienen asociado un valor de plazo o tiempo límite (*timeout*). Si no se recibe una respuesta dentro del periodo del plazo, la llamada devuelve un fracaso al programa invocador. Dado el predominio de los medios de transmisión por red que no cuentan con un límite superior para el tiempo de entrega de los mensajes (por ejemplo, Ethernet), el programa invocador no puede distinguir entre un fracaso de la red y un fracaso del servidor. Para decidir que hacer, el programa invocador debe saber cómo se realizó la llamada subyacente, es decir, la *semántica de llamada en caso de fracasos*. Son varias las opciones comunes para la semántica de llamada:

- Con la *semántica de exactamente una vez*, se garantiza que la llamada se ejecutó una y sólo una vez. Es imposible garantizar la semántica de exactamente una vez si los servidores remotos se pueden caer.
- Con la *semántica quizá* no se ofrecen garantías. La llamada se realiza una vez, y si no se recibe respuesta en un lapso determinado, la llamada regresa con un error, en cuyo caso el cliente no tiene idea de si la solicitud se ejecutó y se perdió la confirmación o si se perdió la solicitud inicial.

- Con la *semántica de cuando mucho una vez*, el mecanismo invocador trata de efectuar la llamada; si no recibe respuesta, lo vuelve a intentar. El servidor elimina las solicitudes repetidas y envía respuestas sin volver a ejecutar la solicitud. Mientras el servidor no se caiga, la semántica de cuando más una vez garantiza que la solicitud se ejecutará exactamente una vez. Si el servidor se cae, la solicitud podría haberse ejecutado una vez o ninguna, dependiendo del momento de la caída. El servidor debe guardar estos números de solicitud en almacenamiento permanente para poder reiniciarse después de una caída.
- Con la *semántica de por lo menos una vez*, el mecanismo de invocación subyacente sigue reintentando la llamada hasta que recibe una respuesta.

En el capítulo 12 dijimos que UDP es un protocolo sin conexiones que no realiza verificación de errores. Se envía un solo paquete y no se verifica que haya llegado. El mecanismo de transporte UDP para las llamadas a procedimientos remotos TI_RPC de ONC ofrece semántica quizá porque no hay garantía de que los paquetes lleguen. Utilice UDP en aplicaciones que de todos modos implementen la verificación de errores en el nivel de usuario. El *software* del servidor de archivos NFS de Sun utiliza TI-RPC de ONC con un transporte UDP.

El transporte TCP ofrece transmisión de datos libre de errores orientada a conexiones. En tanto el servidor no se caiga, la solicitud llegará al servidor y el cliente recibirá la respuesta. El protocolo TCP logra ser tolerante respecto de fracasos acusando el recibo de cada paquete y retransmitiendo los paquetes para los cuales recibe una confirmación negativa. El transporte TPC proporciona semántica de, cuando mucho, una vez para TI-RPC de ONC.

14.8 NFS —Sistema de archivos de red

El sistema de archivos de red NFS (*Network File System*) de Sun es un sistema que se ha transportado a muchas máquinas y que se encarga del acceso a archivos en una red. NFS, que apareció en 1985, procura ofrecer un acceso eficiente y transparente a sistemas de archivos remotos en una red heterogénea. La versión 3 apareció en 1994. NFS se implementa como un servidor sin estados por RPC de forma similar al `put_block` sin estados del ejemplo 14.16.

La mayor parte de las llamadas UNIX estándar para archivos y directorios tienen contrapartes NFS para acceso remoto. El programa 14.18 es la especificación RPC de los servicios para la versión 3 de NFS. La mayor parte de los servicios no requieren explicación y son análogos a las llamadas de sistema de UNIX correspondientes para archivos locales. Hay procedimientos remotos para obtener o establecer atributos, leer, escribir, crear o eliminar directorios o enlaces simbólicos y obtener información de situación. Cada uno de los servicios remotos de NFS tiene tipos bien definidos para sus parámetros y un tipo para su valor de retorno.

La función NFS para escribir en un archivo, `NFSPROC3_WRITE`, tiene un parámetro y valores de retorno especificados por `WRITE3args` y `WRITE3res`, respectivamente. El programa 14.19 muestra las especificaciones para estos tipos; recuerde que están en un lenguaje de especificación, no en C.

Programa 14.18: Especificación de procedimientos remotos de NFS versión 3.

```
program NFS_PROGRAM {
   version NFS_V3  {
      void NFSPROC3_NULL(void)                                    = 0;
      GETATTR3res NFSPROC3_GETATTR(GETATTR3args)                  = 1;
      SETATTR3res NFSPROC3_SETATTR(SETATTR3args)                  = 2;
      LOOKUP3res NFSPROC3_LOOKUP(LOOKUP3args)                     = 3;
      ACCESS3res NFSPROC3_ACCESS(ACCESS3args)                     = 4;
      READLINK3res NFSPROC3_READLINK(READLINK3args)               = 5;
      READ3res NFSPROC3_READ(READ3args)                           = 6;
      WRITE3res NFSPROC3_WRITE(WRITE3args)                        = 7;
      CREATE3res NFSPROC3_CREATE(CREATE3args)                     = 8;
      MKDIR3res NFSPROC3_MKDIR(MKDIR3args)                        = 9;
      SYMLINK3res NFSPROC3_SYMLINK(SYMLINK3args)                  = 10;
      MKNOD3res NFSPROC3_MKNOD(MKNOD3args)                        = 11;
      REMOVE3res NFSPROC3_REMOVE(REMOVE3args)                     = 12;
      RMDIR3res NFSPROC3_RMDIR(RMDIR3args)                        = 13;
      RENAME3res NFSPROC3_RENAME(RENAME3args)                     = 14;
      LINK3res NFSPROC3_LINK(LINK3args)                           = 15;
      READDIR3res NFSPROC3_READDIR(READDIR3args)                  = 16;
      READDIRPLUS3res NFSPROC3_READDIRPLUS(READDIRPLUS3args)      = 17;
      FSSTAT3res NFSPROC3_FSSTAT(FSSTAT3args)                     = 18;
      FSINFO3res NFSPROC3_FSINFO(FSINFO3args)                     = 19;
      PATHCONF3res NFSPROC3_PATHCONF(PATHCONF3args)               = 20;
      COMMIT3res NFSPROC3_COMMIT(COMMIT3args)                     = 21;
   } = 3;
} = 100003;
```

Programa 14.18

Programa 14.19: Especificación de tipos de parámetros y valores de retorno para NFSPROC3_WRITE.

```
enum stable_how {
   UNSTABLE = 0,
   DATA_SYNC = 1,
   FILE_SYNC = 2
};

struct WRITE3args {
   nfs_fh3        file;
   offset3        offset;
   count3         count;
   stable_how     stable;
   opaque         data<>;
}
```

```
struct WRITE3resok {
   wcc_data     file_wcc;
   count3       count;
   stable_how   committed;
   writeverf3   verf;
};

struct WRITE3resfail {
   wcc_data     file_wcc;
};

union WRITE3res switch (nfsstat3 status) {
   case NFS3_OK;
      WRITE3resok  resok;
   default;
      WRITE3resfail resfail;
};
```

Programa 14.19

Todas las especificaciones de parámetros y retorno para las funciones NFS se ajustan al formato del programa 14.19. `file` es el manejador de archivo NFS para el archivo que se va a escribir; es un identificador opaco proporcionado por el servidor cuando se localiza inicialmente el archivo. Los miembros `offset` (posición), `count` (cuenta) y `data` (datos) tienen el mismo significado que los miembros de `struct packet` de la especificación de `put_block` del programa 14.15.

El valor usual de `stable` es `FILE_SYNC`, que indica que el servidor debe completar la escritura en disco antes de regresar de la llamada remota. Las otras posibilidades se explicarán más adelante en esta sección.

Los servicios NFS devuelven diferentes estructuras cuando tienen éxito y cuando fracasan. Los valores de retorno son manejados por la unión conmutada union `WRITE3res`. Si la llamada RPC tiene éxito, devuelve una estructura `struct WRITE3resok`, la cual incluye una cuenta del número de bytes que se escribieron realmente y una indicación de si los datos se escribieron en almacenamiento permanente o en una reserva de memoria (*cache*). Si `NFSPROC3_WRITE` fracasa, devuelve una estructura `struct WRITE3resfail`.

La implementación en el nivel de usuario de `put_block` del ejemplo 14.16 requiere una llamada `open` (abrir) explícita en cada bloque que se escribe porque tiene que acceder al disco a través de las llamadas del sistema de archivos. La implementación de `NFSPROC3_WRITE` del servidor NFS forma parte del núcleo; puede pasar por alto las llamadas del sistema de archivos en el nivel de usuario y obtener bloques directamente sin ejecutar una llamada `open` explícita. La obtención directa implica un gasto extra mucho menor que la secuencia `open`, `lseek`, `write` y `close`.

Puesto que los servidores NFS carecen básicamente de estados, no guardan información acerca de los archivos abiertos. El núcleo del cliente mantiene información referente a los

archivos que sus usuarios han abierto. En la función `put_block` del programa 14.15, el archivo se identifica mediante el nombre de trayectoria completa y el servidor recorre su árbol de directorios para localizar el archivo. En NFS, el cliente recorre el árbol de directorios del servidor mediante una serie de llamadas de consulta. Cada consulta produce una asa de archivo NFS para ese nodo del árbol de directorios y permite al cliente leer el directorio a fin de obtener la asa de archivo del siguiente nodo árbol abajo. Las asas de archivo son opacos para el cliente con objeto de que éste pueda acceder a servidores que emplean diferentes formatos de archivo. Una consecuencia de hacer que el cliente recorra el árbol de directorios es que cada sistema de archivos al que se accede remotamente mediante NFS debe estar montado en algún punto del sistema de archivos raíz del cliente. El procedimiento de montaje es independiente del protocolo NFS.

14.8.1 Almacenamiento en reservas de memoria (*caching*) y consistencia

La filosofía del NFS es hacer el servidor lo más sencillo que se pueda y dejar que el cliente realice la mayor parte del trabajo. Tenga presente que el programa que realiza las llamadas RPC no es un programa de usuario; más bien, es una parte del sistema operativo del cliente de NSF. La figura 14.9 es un diagrama de la operación. Un programa de usuario que se ejecuta en un cliente NFS realiza operaciones ordinarias con archivos, como `read`. Si la llamada `read` hace referencia a un archivo remoto, el núcleo del cliente iniciará una solicitud de NFS remota a nombre del usuario. En el protocolo NFS estándar, el núcleo del cliente se bloquea hasta que el servidor remoto responde con el bloque de datos. Esto tarda mucho.

Aunque no forma parte de la especificación NFS, las implementaciones prácticas de NFS utilizan almacenamiento temporal (*caching*) de memoria tanto en el cliente como en el servidor a fin de mejorar el rendimiento. El *uso de almacenamiento temporal de memoria* (*caching*) se refiere a la retención de datos en la memoria para poder usarlos sin tener que obtenerlos otra vez. El uso de reservas de memoria no afecta la forma en que el cliente y el servidor intercambian información, pero sí reduce considerablemente el tráfico entre el cliente y el servidor.

El núcleo convencional de UNIX utiliza reservas de memoria de varias maneras para hacer más eficiente el acceso a archivos. El núcleo mantiene en la memoria una copia de los inodos de todos los archivos abiertos; además, podría guardar inodos de los archivos y nodos de directorio a los que accedió recientemente. El núcleo puede conservar tablas de nombres de trayectoria y números de inodos para agilizar el recorrido de los árboles de directorios; además, conserva una reserva grande de bloques de datos a los que accedió recientemente. Cuando un proceso solicita un bloque de datos de un archivo, lo primero que hace el núcleo es verificar si está en la memoria. Casi todas las implementaciones utilizan lectura adelantada para preobtener bloques de archivo adicionales cuando se accede a un bloque en particular, a fin de anticipar solicitudes futuras. Cuando un proceso realiza una escritura, el bloque se escribe en el *buffer* de memoria, no directamente en el disco. El sistema operativo escribe periódicamente en el disco los bloques de datos modificados. Esta escritura periódica se conoce como operación `sync`, y por lo regular se efectúa cada 30 segundos. Si el sistema se cae entre dos operaciones `sync`, es posible que la copia que está en disco no refleje los cambios más recientes, pero

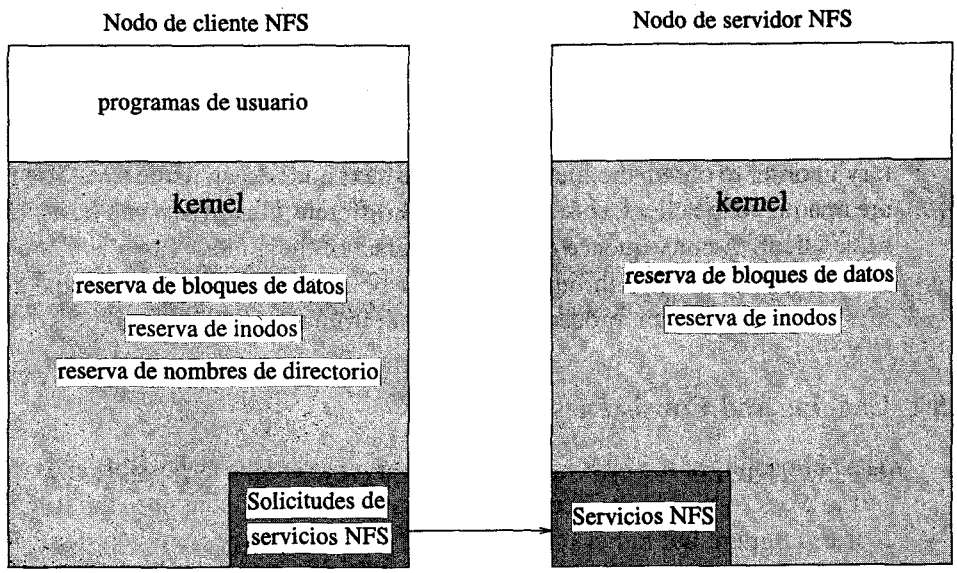


Figura 14.9: Diagrama esquemático de la relación entre clientes y servidores de NFS.

fuera de esto todo es consistente, porque las lecturas y escrituras en el sistema pasan por el mismo almacenamiento temporal del sistema (*system cache*).

El uso del mecanismo de reservas tanto en los clientes como en los servidores de NFS puede mejorar considerablemente el rendimiento de los servicios de archivos remotos, pero también causa graves problemas de consistencia. La mayor parte de las implementaciones utiliza reservas de memoria (*caching*) y cuentan con mecanismos adicionales para mejorar, pero no garantizar, la consistencia.

El uso de reservas en el lado del cliente se asemeja en muchos aspectos al almacenamiento convencional de archivos en memoria. Los programas de usuario leen bloques de la reserva de bloques de datos y los escriben en ella. El núcleo preobtiene bloques y periódicamente escribe los bloques modificados. Las obtenciones y las escrituras de bloques modificados son llamadas al servidor de NFS, no a los controladores de disco locales. Cuando el cliente emite una solicitud a NFS, se bloquea hasta que la llamada regresa.

El servidor NFS mantiene una reserva grande de los bloques que se solicitaron recientemente, y también realiza lecturas adelantadas para anticipar futuras solicitudes de lectura de los clientes. Las escrituras son más complicadas. En el protocolo de la versión 2 de NFS, el servidor tenía que completar la estructura en almacenamiento estable antes de responder al cliente. Si el miembro `stable` de la solicitud `NFSPROC3_WRITE` tiene el valor `FILE_SYNC`, el servidor escribe en almacenamiento estable antes de regresar al cliente. Si el miembro `stable` tiene el valor `UNSTABLE`, el servidor sólo está obligado a escribir el bloque en su propia reserva (*cache*); más tarde puede escribirlo en disco como parte de su operación `sync` normal o en respuesta a una solicitud `NFSPROC3_COMMIT` del cliente. Si `stable` tiene el valor `DATA_SYNC`, el servidor debe escribir los datos en almacenamiento estable, pero no tiene que asegurar los

metadatos. El término *metadatos del sistema (system metadata)* se refiere a información de estado y de atributos que usualmente se almacena en el inodo del archivo.

Un problema importante del uso de reservas de memoria *(caching)* es que los clientes y los servidores no comparten la misma reserva. Las copias de un bloque de datos pueden perder su consistencia si un usuario en un cliente escribe un bloque en su reserva local y un usuario en otro cliente accede a una copia local del mismo bloque antes de que reciba la actualización. NFS no especifica un mecanismo para asegurar que las copias de bloques de datos almacenadas en reservas sean consistentes. Cada implementación está en libertad de utilizar su propio método para determinar la consistencia. Una estrategia típica consiste en que el cliente pregunte periódicamente cuándo se efectuó la última modificación de un bloque dado invocando a `NFSPROC3_GETATTR`. Si el cliente detecta que la copia que está en el servidor fue modificada después de que el cliente la obtuvo, puede marcar su propia copia como no válida. Esta técnica permite lograr una consistencia "muy buena" *(consistencia de reserva débil weak cache unsistency)* pero no absoluta; se basa en la suposición de que sólo alrededor del 5% de los accesos remotos son operaciones de escritura y que normalmente se escribe en archivos no compartidos. A menudo se ofrece un servicio de candados independiente para aplicaciones que escriben en bloques compartidos.

Un problema de utilizar marcas de tiempo para aproximar la consistencia de las reservas es que el cliente y el servidor no están sincronizados con exactitud. La falta de sincronización entre el reloj del cliente y el del servidor se denomina *sesgo de tiempo*. Es posible que el cliente y el servidor tengan que intercambiar información periódicamente para determinar por cuánto están fuera de sincronía sus relojes.

Otro problema relativo al tiempo tiene que ver con los efectos secundarios sobre operaciones que por lo demás son equipotentes. Supongamos que un cliente emite una solicitud `NFSPROC3_WRITE` y no recibe respuesta del servidor. En la implementación usual, el cliente esperará cierto tiempo y luego reintentará la solicitud. Supongamos que el servidor atendió la primera solicitud pero se perdió la respuesta. Puesto que las operaciones tienen el efecto secundario de cambiar la hora de modificación, la segunda escritura cambia el estado del archivo. A fin de reducir los efectos secundarios y mejorar la eficiencia, la especificación de protocolo de la versión 3 de NFS recomienda que los servidores mantengan una reserva de solicitudes recientes llamada *reserva de solicitudes repetidas*. En esta reserva se registra el estado de terminación *(completion status)* de las solicitudes recientes. Si el servidor recibe una solicitud que es un duplicado de otra que está en la reserva, simplemente devuelve el estado de retorno original sin intentar ejecutar otra vez la operación.

14.9 Hilos y llamadas a procedimientos remotos

Las operaciones RPC normales son síncronas y en serie, lo que significa que el cliente se bloquea después de una llamada RPC hasta recibir una respuesta. Por tanto, un cliente sólo puede emitir una solicitud a la vez. En el lado del servidor, éste sólo procesa una solicitud a la vez. La estrategia tradicional para aumentar el paralelismo en el lado del servidor consiste en instalar múltiples servidores (como los demonios servidores de NSF) en un nodo.

Los hilos ofrecen nuevas oportunidades de paralelismo tanto para los clientes como para los servidores. Un cliente de RPC puede dedicar un hilo a la emisión de llamadas remotas y a la espera de respuestas mientras el resto del programa hace otras cosas. Varios hilos del cliente pueden efectuar llamadas remotas simultáneas siempre que cada hilo cree su propia asa con `clnt_create`. Si los hilos del cliente comparten una sola asa de RPC, las llamadas se procesarán secuencialmente, no en paralelo.

El manejo de servidores de RPC con hilos es más complicado porque cada servicio remoto se puede invocar como un hilo independiente. Es obvio que el programa debe proteger las variables compartidas y utilizar mecanismos apropiados para sincronizar la interacción de los servicios. Una buena estrategia consiste en crear una versión local con hilos en la que el programa principal crea un hilo nuevo para invocar el servicio y esperar el resultado. El programa principal puede iniciar muchos hilos simultáneos y probar las interacciones entre los servicios.

Los parámetros y valores de retorno presentan un problema sutil que surge a causa de la estructura tradicional de los talones de RPC. Un talón de servidor de un solo hilo utiliza variables estáticas para desorganizar información y pasar parámetros a los servicios. La función remota comunica su valor de retorno devolviendo un apuntador a una variable estática. Entonces, el talón utiliza la variable estática para organizar un valor de retorno. Un servidor con hilos debe apartar espacio para los parámetros y el valor de retorno antes de crear el hilo de servicio. Cuando el hilo termine, el talón de servidor deberá liberar el espacio apartado. Estos cambios requieren modificaciones del talón de servidor.

Por suerte, `rpcgen` cuenta ya con una opción `-A` para generar automáticamente un servidor multihilo. El servidor resultante se ejecuta en *modo MT automático*. La opción `-A` hace que `rpcgen` genere servicios remotos cuyos parámetros y valores de retorno se guardan en espacio de memoria apartado dinámicamente en lugar de asignarse a variables estáticas. El archivo de `plantilla _server.c` también incluye funciones que el talón servidor invoca para liberar almacenamiento apartado dinámicamente una vez que el servicio regresa.

Ejemplo 14.19

El programa 14.20 es el archivo de plantilla `rfile_server.c` *para un servidor con hilos producido por el siguiente comando.*

```
rpcgen -a -A -C rfile.x
```

En la función `put_block_1_svc` del programa 14.20, `*result` es el valor de retorno para el servicio `put_block` especificado en `rfile.x`. El valor de retorno real para `put_block_1_svc` es sólo una bandera booleana que indica que la función se ejecutó. Basta asignar 1 a `retval` para indicarle al cliente que la llamada se ejecutó.

Programa 14.20: Plantilla para la versión con hilos del servidor `put_block`.

```
#include "rfile.h"

bool_t
put_block_1_svc(packet *argp, int *result, struct svc_req *rqstp)
{
    bool_t retval;
    /*  inserte aquí el código del servidor */
    return (retval);
}

int
rfileprog_1_freeresult(SVCXPRT *transp, xdrproc_t xdr_result,
                       caddr_t result)
{
   (void) xdr_free(xdr_result, result);
   /* Inserte aquí el código de liberacion adicional, si es necesario */
}
```

Programa 14.20

Ejemplo 14.20

El programa 14.21 es un archivo de plantilla `rfile_server.c` *para un servidor sin hilos producido por el siguiente comando* `rpcgen`. *La función devuelve* `result` *al talón de servidor mediante un apuntador a una variable estática.*

```
rpcgen -a -C rfile.x
```

Programa 14.21: El archivo de plantilla `rfile_server.c` para un servidor sin hilos.

```
#include "rfile.h"

int *
put_block_1_svc(packet *argp, struct svc_req *rqstp)
{
   static int  result;
   /*  inserte aquí el código del servidor */
   return (&result);
}
```

Programa 14.21

El programa 14.22 es una versión con hilos del archivo `rfile_server.c` después de insertar el código en la plantilla. Puesto que `put_block_1_svc` no hace referencia a variables estáticas, las invocaciones simultáneas de la función no se interfieren.

Programa 14.22: El servicio `put_block` en un servidor con hilos.

```
#include "rfile.h"

bool_t
put_block_1_svc(packet *pack, int *result, struct svc_req *rqstp)
{
    bool_t retval;
    int fd;

    *result = 0;
    if ((fd = open(pack->fname, O_WRONLY|O_CREAT, 0600)) == -1)
        *result = -1;
    else if (lseek(fd, pack->offset, SEEK_SET) == -1)
        *result = -1;
    else
       *result = write(fd, pack->data.data_val, pack->count);
    close(fd);
    retval = 1;
    return retval;
}

int
rfileprog_1_freeresult(SVCXPRT *transp, xdrproc_t xdr_result,
    caddr_t result)
{
   (void) xdr_free(xdr_result, result);
   return 0;
}
```

Programa 14.22

En las llamadas a un servidor sin hilos, el cliente detecta un fracaso de RPC cuando la llamada remota devuelve un apuntador `NULL` al valor de retorno. En las llamadas a servidores con hilos, la llamada remota devuelve un código de error que especifica si la llamada tuvo o no éxito. El código de error es `RPC_SUCCESS` cuando la función tiene éxito. La constante `RPC_SUCCESS` tiene un valor numérico de 0, lo que puede dar pie a confusiones ya que un valor de retorno de 0 para el servicio en el lado del servidor indica fracaso. Observe que la función `put_block_1_svc` del programa 14.22 devuelve 1 para indicar éxito.

El resultado real de una llamada a un servicio con hilos se devuelve como un parámetro, así que una llamada de cliente a un servidor con hilos tiene un parámetro adicional que contiene un apuntador al valor de retorno. El programa 14.23 muestra la llamada del lado del cliente al servicio remoto en el servidor con hilos. La llamada devuelve un código de error en lugar de un apuntador a un resultado.

Programa 14.23: La función cliente `write_file` invoca un servicio en un servidor RPC con hilos.

```
#include "rfile.h"

int write_file(CLIENT *clnt, char *filename, char *buffer, int nbytes)
{
   int  bytes_written;
   enum clnt_stat retval_1;
   int result_1;
   packet  pack;

   pack.count = nbytes;
   pack.offset = fd_offset;
   pack.fname = filename;
   pack.data.data_val = buffer;
   pack.data.data_len = nbytes;
   retval_1 = put_block_1(&pack, &result_1, clnt);
   if (retval_1 != RPC_SUCCESS)
      bytes_written = -1;
   else {
      bytes_written = result_1;
      fd_offset += bytes_written;
    }
    return bytes_written;
}
```

Programa 14.23

En un servidor de un solo hilo creado por `rpcgen`, el talón establece la comunicación utilizando un servicio de transporte como TCP con *sockets*. A continuación, el talón invoca una función `svc_run` que vigile los descriptores de comunicación mediante `select` o `poll`. Cuando llega una solicitud, el servidor la desorganiza, invoca el servicio como una función ordinaria, organiza los resultados y contesta por el descriptor para comunicación. Después, el servidor reanuda la vigilancia de los descriptores para detectar solicitudes adicionales. La opción `-A` de `rcpgen` no sólo resuelve el problema del paso de parámetros, sino que genera automáticamente una función `svc_run` que crea un hilo nuevo para cada solicitud de servicio. Sun Solaris 2.4 permite por omisión un máximo de 16 hilos simultáneos, pero el talón de servidor puede cambiar este valor invocando a `rpc_control`.

El programa `rpcgen` utiliza hilos Solaris, no POSIX. La documentación de Solaris indica que un programa puede combinar los mecanismos de sincronización de POSIX y de Solaris en el mismo programa. No olvide agregar `-lpthreads` a `LDLIB` en el `makefile` y los archivos de cabecera `pthreads.h` apropiados en el programa fuente.

Al parecer, las llamadas remotas con hilos se encuentran en una etapa de transformación continua y todavía no surgen estándares en lo que respecta a su formato. Lea la documentación vigente antes de crear servidores con hilos en un sistema determinado. He aquí algunas pautas que puede seguir al escribir un servidor con hilos:

- Cree los servicios remotos como llamadas locales sencillas.
- Convierta las llamadas locales en llamadas locales con hilos.
- Para realizar pruebas, utilice un programa que cree múltiples hilos que invoquen las funciones locales. Cada hilo invoca la función local con una llamada ordinaria, espera un valor de retorno y termina. Los problemas de sincronización se detectan con mayor facilidad en un entorno local que en uno remoto. No continúe con el siguiente paso hasta que la versión local con hilos funcione.
- Cree un archivo de especificación `.x` (por ejemplo, `rfile.x` o `rand.x`). (Los archivos de especificación para las versiones con hilos y sin hilos son iguales.)
- Ejecute `rpcgen` con el comando

```
rpcgen -a -A -C rfile.x
```

  (Sustituya el nombre del archivo de especificación [por ejemplo, `rfile.x` o `rand.x`.] La opción `-A` designa el *modo MT automático*. Con esta opción, `rpcgen` genera un talón servidor que automáticamente crea hilos para atender cada solicitud.
- Si usa hilos POSIX en lugar de Solaris, no olvide agregar `-lpthreads` al archivo creador e incluir el archivo de cabecera `pthreads.h` en el programa fuente.

14.10 Ejercicio: Servidor de archivos sin estados

Este ejercicio explora diversos aspectos del recurso de llamadas a procedimientos remotos TI-RPC de ONC.

- Cree un directorio nuevo y copie en él la especificación `rfile.x` para la versión remota de `put_block`.
- Ejecute `rpcgen -C -a rfile.x` para generar los archivos necesarios.
- Utilice los archivos resultantes para implementar un cliente remoto similar al del programa 14.17.
- Modifique el archivo `rfile_server.c` para implementar un servidor remoto similar el del programa 14.16.
- Cree los archivos utilizando el archivo creador `makefile.rfile`.
- Ejecute `rfile_server` para registrar el servicio con `rcpbind`.
- En la misma máquina, ejecute `rfile_client` con los argumentos de línea de comandos apropiados. Verifique que los archivos se transmitan correctamente.

- Ejecute `rfile_server` en una máquina distinta de aquella en la que se ejecuta `rfile_client` para comprobar que transmita archivos correctamente.
- Experimente con diferentes valores de `NETPATH`. En particular, pruebe a transmitir un archivo grande con UDP y luego con TCP. Determine si la versión con UDP llegó bien. Compare los tiempos de transmisión de ambos transportes. Realice el mismo experimento con varias transferencias `ftp` de archivos grandes ejecutándose al mismo tiempo. ¿Hay diferencias entre UDP y TCP en cuanto a rendimiento y exactitud en estas circunstancias?
- Agregue funciones `get_block` (obtener bloque) y `read_file` (leer archivo) a la versión local del programa. Estas funciones son análogas a `put_block` y `write_file`; sus prototipos son

```
int get_block(char *fname, int offset, int count,
          char *data);
int write_file(int fd, char *buf, int nbytes);
```

  Depure la versión local y luego agregue la función `get_block` como servicio remoto al servidor de archivos.
- Agregue una función `myrddir` a la versión local del programa. Esta función es similar a la función de biblioteca `readdir` de C; su prototipo es

```
struct dirent myrddir(char *pathname, int cookie);
```

  La función devuelve una entrada del directorio `pathname` según se especifique con `cookie`. Un valor de `cookie` de 0 indica la primera entrada, un valor de `cookie` de 1 indica el segundo nombre de archivo, y así sucesivamente. Escriba la función de modo que sea equipotente. Después de depurar la versión local, agregue `myrddir` como servicio remoto al servidor de archivos.
- Añada una función `mystat` a la versión local del programa. La función `mystat` es similar a `stat` y su prototipo es

```
struct stat mystat(char *pathname)
```

  La función devuelve una estructura `struct stat` para el archivo indicado por la trayectoria `pathname`. Escriba la función de modo que sea equipotente. Después de depurar la versión local, agregue `mystat` como servicio remoto al servidor de archivos.
- Escriba una función `showdir` que exhiba el contenido de un directorio remoto. El prototipo es

```
int showdir(int fd, char *pathname);
```

  La función envía una lista de las entradas del directorio `pathname`, una por línea, al archivo `fd`. La función invoca a `myrddir` para leer el directorio. La función `showdir` forma parte del programa cliente y no tiene que ser equipotente.

- Escriba una función `showall` que exhiba una entrada de directorio seguida de su información de inodo (dueño, hora de creación, etc.). La función tiene el prototipo

```
int showall(int fd, char *pathname);
```

La función exhibe la entrada de directorio seguida de su información de inodo. Utilice `myrddir` y `mystat` para implementar la función. La función `showall` forma parte del programa cliente y no tiene que ser equipotente.

14.11 Lecturas adicionales

La llamada a un procedimiento remoto fue propuesta inicialmente por Birrell y Nelson [8] en un artículo clásico intitulado "Implementing remote procedure calls". El libro *UNIX System V Network Programming* de Stephen Rago incluye un tratamiento de XDR y la programación RPC de bajo nivel [71]. El análisis más extenso de las RPC se encuentra en *Power Programming with RPC*, de John Bloomer [11]. Russel Sandberg presenta un panorama general sobre la implementación original de NFS en un Sun White Paper [76] titulado "The Sun Network File System: Design, implementation and experience". La especificación completa de la versión 3 de NFS se puede obtener por la Internet en el sitio ftp anónimo `ftp.uu.net`.

Capítulo 15

Proyecto: *Espacio de tuplas*

Linda es un elegante lenguaje de programación diseñado para apoyar el procesamiento en paralelo en redes de computadoras. El lenguaje se basa en primitivas simples y un modelo de espacio de tuplas que utiliza datos compartidos para la comunicación y la sincronización, en contraste con los enfoques tradicionales de transferencia de mensajes o memoria compartida. David Gelernter, el inventor de Linda, expresa así el principio que sustenta este lenguaje: "La elegancia al escribir *software* consiste en lograr la máxima funcionalidad con el mínimo de complejidad... Es lo mismo que con la buena prosa: sacar el mayor jugo posible a cada palabra que se escribe." En este capítulo desarrollaremos la especificación de un modelo de espacio de tuplas simplificado e ilustraremos su uso en la construcción de una aplicación de búsqueda cooperativa distribuida. El proyecto integra la mayor parte de los conceptos cubiertos en los capítulos anteriores, culminando con un servidor de llamadas a procedimientos remotos con hilos para un subconjunto de Linda. Los mecanismos que crearemos en este capítulo tienen una potencia asombrosa y establecen los cimientos para un sistema operativo verdaderamente distribuido.

Un *espacio de tuplas* contiene objetos llamados *tuplas*, que son colecciones ordenadas de elementos de datos. Cuando un proceso tiene necesidad de comunicarse, genera una tupla y la inserta en el espacio de tuplas. Otros procesos pueden recuperar tuplas del espacio de tuplas. Los emisores y receptores están desacoplados en este modelo de programación y unos no conocen la identidad de los otros; solamente se comunican a través de información compartida. Un número sorprendente de problemas de programación paralela se ajusta bien a este paradigma de programación, y todas las primitivas paralelas se pueden simular en él [33].

El espacio de tuplas semeja una memoria asociativa compartida que contiene tuplas y maneja cuatro operaciones: `out`, `in`, `rd` y `eval`. La operación `out` coloca una tupla en el

espacio de tuplas. La operación `in` encuentra una tupla apropiada en el espacio de tuplas, la elimina y devuelve su valor. La operación `rd` es similar a `in`, pero `rd` elimina la tupla después de devolver su valor. Por último, `eval` crea una tupla activa (un proceso).

El modelo de espacio de tuplas constituye la base del sistema de comunicación en paralelo Linda creado en la Universidad de Yale [17] y actualmente comercializado por Scientific Computing Associates. Linda contempla dos tipos de tuplas: ordinarias y activas. Las tuplas ordinarias contienen datos, en tanto que las activas representan procesos. Además de las cuatro operaciones estándar del espacio de tuplas (`out`, `in`, `rd` y `eval`), la versión comercial de Linda cuenta con versiones no bloqueadoras de `in` y `rd` que designa como `inp` y `rdp`, respectivamente.

Una meta de la computación en paralelo es lograr que el desarrollo de aplicaciones paralelas y distribuidas sea completamente transparente para el programador. Una estrategia para lograr esto es hacer paralelos los compiladores; otra es la abstracción del espacio de tuplas. El programador medio no se va a conformar con escribir una aplicación en PVM (véase el capítulo 11). La creación de Linda es una contribución importante al campo de la computación en paralelo en virtud de la sencillez y potencia expresiva de este lenguaje. Linda permite a los usuarios crear aplicaciones distribuidas en un nivel muy alto en comparación con otros sistemas de coordinación distribuida como PVM. El usuario escribe programas empleando la abstracción de tuplas y no conoce ni controla la distribución subyacente del problema en la red.

En la práctica, la abstracción de Linda es un arma de dos filos. Si la implementación de Linda es eficiente, todo saldrá bien, pero si la implementación tiene un desempeño pobre, el resultado será un sistema elegante pero poco práctico o incluso inútil. Se asegura que las versiones comerciales de Linda son eficientes, pero no han demostrado de manera convincente que esto sea verdad. En cualquier caso, la mayor parte de los sistemas distribuidos tienen un gasto extra considerable en comparación con los sistemas operativos de monoprocesador, y el hecho de partir de un modelo abstracto permite esperar que el rendimiento mejorará conforme la implementación subyacente se perfeccione.

El proyecto de este capítulo es ambicioso: la implementación de un sistema de espacio de tuplas plenamente funcional llamado Richard. (El nombre Richard continúa la tradición de bautizar los lenguajes de programación en honor de miembros famosos de la familia Lovelace. Richard Lovelace fue un poeta inglés del siglo XVII que tendió un velo romántico sobre su vida en prisión y murió en la miseria.) En la siguiente sección describiremos brevemente a Linda, y en la que sigue presentaremos a Richard. En la sección 15.3 especificaremos una implementación de un espacio de tuplas Richard al que se accede mediante llamadas a procedimientos remotos. Esta implementación no pretende ser eficiente. En la sección 15.4 se analiza la optimización cooperativa como una aplicación del espacio de tuplas y se crea un algoritmo de retroceso para el problema de *n* reinas, a manera de ejemplo. En la sección 15.5 extenderemos Richard para incluir tuplas activas, mientras que en la sección 15.6 la extensión tendrá por objeto incluir tuplas que son por sí mismas espacios de tuplas. En la sección 15.7 se describirá la forma de manejar con hilos el servidor de espacio de tuplas Richard. Estas tres extensiones del sistema Richard básico son independientes entre sí, pero juntas establecen los cimientos para construir un sistema de computación distribuida.

15.1 Linda

El modelo de programación Linda se basa en cuatro operaciones de tuplas, `out`, `in`, `rd` y `eval`, que acrecientan un lenguaje de programación estándar como FORTRAN o C. Preprocesadores especiales de Linda y compiladores optimizadores traducen los programas para la arquitectura objetivo subyacente y procuran minimizar los gastos en tiempo de ejecución.

Ejemplo 15.1

Una tupla es una colección ordenada de elementos de datos. La siguiente operación `out` *inserta una tupla con tres elementos en el espacio de tuplas.*

```
out("elemento", 5, 3.0);
```

El orden de los valores en una tupla de Linda es importante: una tupla `("elemento", 3.0, 5)` no es idéntica a la tupla del ejemplo 15.1. En el espacio de tuplas pueden residir copias idénticas de los elementos, así que dos operaciones `out` con la misma tupla insertan dos copias en el espacio de tuplas. La operación `out` nunca se bloquea.

Hay dos tipos de tuplas en Linda: ordinarias y activas. Las tuplas activas se crean con `eval` y corresponden a la ejecución de una función. Cuando la función termina de ejecutarse, la tupla activa se convierte en ordinaria. Las operaciones `in` y `rd` recuperan tuplas ordinarias del espacio de tuplas. La diferencia principal entre las dos operaciones es que `in` retira la tupla del espacio, en tanto que `rd` se limita a devolver una copia. Estas operaciones se bloquean si no hay una tupla apropiada en el espacio de tuplas y regresan posteriormente cuando aparece tal tupla, sea debido a una operación `out` o a que una operación `eval` completa su ejecución.

La cuestión de qué constituye una tupla apropiada es compleja. La operación `in` o `rd` contiene una plantilla que especifica qué es una tupla apropiada. Linda escoge al azar entre las tuplas que coinciden con la plantilla. La plantilla puede incluir el comodín `?` para indicar que un elemento en particular puede coincidir con cualquier cosa que sea de cierto tipo. Las plantillas y las tuplas coinciden cuando tienen la misma longitud y cada campo tiene el mismo tipo.

Ejemplo 15.2

Cada uno de los siguientes enunciados presenta una plantilla que contiene cinco elementos.

```
rd("x", ?b, 3.0, "esto", ?c);
rd(?y, "uuu", 3.0, ?z, 5.0);
```

Ambas solicitudes del ejemplo 15.2 coinciden con la tupla `("x", "uuu", 3.0, "esto", 5.0)` a condición de que `y`, `b` y `z` sean `char *` y `c` sea `double`. La notación `?b` especifica un *formal* de Linda y dice que la operación `rd` debe sustituir la variable `b` por el valor correspondiente en la tupla devuelta por `rd`. El programa invocador no recibe realmente la tupla devuelta; más bien, se asignan valores de la tupla a las variables que aparecen como formales. Si `rd` fuera una función ordinaria de C, un parámetro `&b` designaría un apuntador a la variable `b` a la

cual la función asignará un valor. Sería difícil escribir una función `rd` general porque la función no tiene forma de saber cuáles de sus parámetros son formales y cuáles contienen valores que deben coincidir. La notación `?b` permite a un preprocesador sustituir la `b` por el enunciado de asignación apropiado para que el usuario no tenga que hacerlo. Sin apoyo por parte del compilador, esta sería una tarea laboriosa para un programador ordinario.

La elegancia de Linda surge de su poder de expresión: los mecanismos de sincronización comunes son fáciles de programar con Linda. Los candados mutex y las actualizaciones atómicas de variables compartidas se expresan de forma natural con pares *in-out*.

Ejemplo 15.3

Los siguientes enunciados de Linda incrementan atómicamente una variable compartida `x`.

```
in(x, ?i);
out(x, i+1);
```

La `x` del ejemplo 15.3 es una etiqueta que rotula la variable compartida. La `i` es una variable local de programa, del mismo tipo que la variable compartida (p. ej., `int`). La operación `in` busca una tupla que contenga dos elementos, el primero de los cuales es `x`. El segundo valor puede ser cualquier `int`. La operación `in` saca la tupla. La operación `out` escribe una tupla nueva cuyo primer valor es `x` y cuyo segundo elemento tiene un valor que es uno más que el elemento correspondiente en la tupla original. El par `in-out` es atómico si sólo hay una tupla en la que el primer valor sea `x`, ya que los demás procesos se bloquearán cuando intenten efectuar una operación `in` con una tupla que no existe.

Linda apoya la transferencia de mensajes con entrega en orden mediante una secuencia de tuplas de la forma

```
(commun_id, sequence_number, message)
```

El `id_commun` identifica la sesión de comunicación y es análogo a los puertos bien conocidos de *sockets*. El `num_secuencial` identifica la posición del mensaje en la secuencia y `mensaje` es uno de los mensajes individuales dentro de la secuencia.

Ejemplo 15.4

El emisor ejecuta el siguiente segmento de código para enviar `n` *enteros de un arreglo* `A`*, cada uno como un mensaje individual.*

```
int A[MAX_SIZE];
int s_seq;
int n;

for (s_seq = 0; s_seq < n; s_seq++)
     out("ID1", s_seq, A[i]);
```

De forma similar, el receptor ejecuta lo siguiente:

```
int B[MAX_SIZE];
int r_seq;
int n;

for (r_seq = 0; r_seq < n; r_seq++)
     in("ID1", r_seq,  ?B[r_seq]);
```

El ejemplo 15.4 garantiza la comunicación en orden, libre de errores, entre el emisor y el transmisor en tanto nadie más haga referencia a una tupla en la que el valor de la primera etiqueta sea `"ID1"`. El emisor y el receptor deben convenir con antelación en identificar su comunicación mediante la etiqueta `"ID1"`. `?B[r_seq]` no es más que una variable a la que se asignará el valor `int` correspondiente de la tupla. El `in` bloqueador controla el flujo de control entre el emisor y el receptor. Si el receptor intenta leer un mensaje que todavía no se envía, se bloqueará hasta que el emisor inserte la tupla necesaria.

En el problema del *buffer* acotado, el emisor se puede bloquear si no hay espacio en el *buffer* para contener el siguiente mensaje. Aquí, el emisor inserta tuplas en tanto haya lugar en el espacio de tuplas. Una operación `out` puede fallar si no hay lugar en el espacio de tuplas, pero tal posibilidad es remota si los programas funcionan correctamente. De hecho, todos los procesos Linda comparten un *buffer* global, con lo cual la flexibilidad es mucho mayor que si un proceso aparta un conjunto fijo de *buffers* para cada sesión de comunicación.

La operación `eval` de Linda crea un proceso nuevo que evalúa cada uno de sus parámetros. Esta operación normalmente se utiliza cuando uno o más de los parámetros son llamadas de función. La operación `eval` nunca se bloquea. Linda inserta una tupla activa en el espacio de tuplas y crea un proceso nuevo para evaluar cada una de las funciones de la tupla. Cuando un proceso termina, deposita el valor de retorno de la función en la entrada correspondiente de la tupla. Una vez que llegan todos los valores de retorno, la tupla se convierte en una tupla ordinaria.

Ejemplo 15.5

El siguiente enunciado en Linda crea un proceso que ejecuta la función `myfun`.

```
eval("MY", i, j, myfun(i, j));
```

El enunciado `eval` del ejemplo 15.5 inserta una tupla activa en el espacio de tuplas y crea un proceso para evaluar `myfun`. Cuando el proceso termina, sustituye la tupla activa por una tupla ordinaria de la forma `("MY", i, j, result)`, donde `result` contiene el valor devuelto por `myfun`. Las operaciones `in` y `read` nunca comparan plantillas con tuplas activas. Una `in` o `read` bloqueada puede desbloquearse si una tupla activa se convierte en ordinaria.

Una tabla de dispersión (*hash*) distribuida es un escenario natural para las implementaciones actuales de Linda, aunque a la larga podrían utilizarse técnicas más avanzadas como las memorias asociativas a gran escala o las reservas de memoria multinivel.

Dado que las tuplas tienen tipos, las implementaciones actuales emplean tablas de dispersión divididas en particiones de acuerdo con el número de componentes de las tuplas y los tipos de sus valores.

15.2 Richard, un Linda simplificado

Las implementaciones de Linda requieren considerable apoyo durante el preprocesamiento, la compilación y la ejecución. En este capítulo desarrollaremos una especificación para un servidor de espacio de tuplas llamado Richard con semántica simplificada y sin apoyo adicional por parte del compilador. La especificación de Richard limita las tuplas y plantillas permisibles con objeto de facilitar la implementación.

A diferencia de las tuplas de Linda que pueden tener cualquier longitud, una tupla de Richard contiene exactamente dos elementos: un identificador de tupla (ID) entero y un valor. En la implementación inicial, el valor sólo puede ser un entero o un arreglo de caracteres. En la sección 15.5 describiremos una extensión en la que los valores de tupla pueden ser funciones, y en la sección 15.6 se verá una extensión en la que los valores de tupla pueden ser espacios de tuplas. Estas extensiones confieren a Richard una funcionalidad similar a la de Kernel Linda [46], la base del sistema operativo QIX.

Richard también difiere de Linda en su semántica de búsqueda. Linda permite operaciones de búsqueda arbitrarias y devuelve una tupla al azar que coincide con la plantilla de búsqueda. Richard sólo permite dos tipos de búsqueda: buscar una tupla con cierto valor de ID y buscar cualquier tupla que contenga cierto tipo de campo de valor. Cuando se busca una tupla con un ID de tupla específico, Richard devuelve las tuplas coincidentes en el orden en que se insertaron en el espacio de tuplas. La semántica de Richard implica que las operaciones `out` ponen en una cola las tuplas que tienen el mismo ID de tupla.

Puesto que Richard se implementa sin apoyo del compilador y sus operaciones deben ajustarse al formato RPC, Richard prescinde de la notación de formales `?i` y sustituye el comodín `?` por las poco estéticas especificaciones `union` y `struct`. (Véanse en la sección 15.3 los horribles detalles.) En los ejemplos de esta sección utilizaremos un pseudocódigo de Richard similar a la sintaxis de Linda. En la sección 15.3 presentaremos la sintaxis de Richard.

Ejemplo 15.6

El siguiente pseudocódigo de Richard elimina la primera tupla que se insertó con `x` *como su primer valor. (Por ahora no se fije en el aspecto de las tuplas de Richard.)*

```
in(x, ?i);
```

En contraste con el ejemplo 15.6, el enunciado `in` de Linda equivalente elimina una tupla al azar que coincide con la plantilla. Otra diferencia importante es que Linda busca que coincidan los tipos de los formales, de modo que si `i` es un entero, las tuplas que satisfagan la operación `in` del ejemplo 15.6 deberán tener un entero como segundo valor. Richard considera que la tupla coincide si el formal es de cualquiera de los tipos permitidos. Esta flexibilidad es

necesaria si no se cuenta con apoyo considerable de preprocesador o compilador. También se aprovechará esta flexibilidad posteriormente al implementar las tuplas activas.

Ejemplo 15.7

El siguiente pseudocódigo Richard implementa la transferencia de mensajes, libre de errores del enunciado `in`*, para una sesión de comunicación identificada por* `x`*. El emisor escribe tuplas que contienen elementos consecutivos del arreglo* `A` *como sigue:*

```
int A[MAX_SIZE];
int i;
int n;

for (i = 0; i < n; i++)
   out(x, A[i]);
```

Una vez que el emisor termina, el receptor elimina las tuplas como sigue:

```
int B[MAX_SIZE];
int i;
int n;

for (i = 0; i < n; i++)
   in(x, ?B[i]);
```

El código del ejemplo 15.7 es pseudocódigo Richard. El receptor recibe los mensajes en el orden en que el emisor los envió porque Richard ponen en una cola las tuplas que tienen el mismo ID de tupla. En el ejemplo 15.4, cada tupla de Linda necesitó un número de secuencia adicional para asegurar la entrega en orden de los mensajes. Puesto que Richard se implementa con C como base sin apoyo adicional por parte del compilador, no podemos utilizar la sintaxis `?B[i]`. En la sección 15.3 se describirán los detalles de la sintaxis real de Richard.

En el ejemplo 15.7 el emisor debe insertar todos sus mensajes antes de que el receptor comience, pues Richard y Linda tienen diferentes semánticas de bloqueo. Las operaciones `in` y `rd` de Linda se bloquean indefinidamente si ninguna tupla del espacio coincide con la plantilla de la solicitud. Varios algoritmos importantes dependen de este mecanismo de bloqueo. La implementación de Richard de este capítulo tiene semántica no bloqueadora (es decir, `in` y `rd` devuelven un erro si ninguna tupla coincide) porque es difícil implementar correctamente las operaciones de bloqueo empleando llamadas a procedimientos remotos. Si el servidor de tuplas no tiene hilos, una llamada bloqueadora puede causar un bloqueo mortal. Además, el plazo de las llamadas a procedimientos remotos se vence si el servidor no responde en un lapso preestablecido.

El bloqueo de llamadas a procedimientos remotos se puede implementar con *llamadas posteriores al cliente*. De lo que se trata es que el servidor devuelva de inmediato una respuesta nula cuando comienza a procesar una llamada. Posteriormente, cuando el servidor tiene listo el resultado, efectúa una llamada de procedimiento remoto al cliente. Los parámetros de esta llamada del servidor al cliente son los valores de retorno de la llamada original. El cliente

tiene que registrar su propio servidor cuando inicia su ejecución. Por desgracia, el empleo de llamadas posteriores obliga a añadir procesamiento de ciclo de sucesos del servidor tanto en el lado del cliente como en el del servidor, y la interacción de las llamadas posteriores con servidores y clientes con hilos es muy complicada.

Ejemplo 15.8

El ejemplo 15.7 no funciona en Richard si el receptor trata de eliminar mensajes antes de que el emisor termine de insertarlos. El siguiente pseudocódigo de Richard implementa la transferencia de mensajes sin esta restricción, pero en una forma poco eficiente. El emisor escribe tuplas que contienen elementos sucesivos del arreglo `A` *como sigue:*

```
int A[MAX_SIZE], i, n;

for (i = 0; i < n;  i++)
    out(x, A[i]);
```

El receptor elimina las tuplas como sigue:

```
int B[MAX_SIZE], i, n;

for (i = 0; i < n; i++)
    while (!in(x, ?B[i]))
           ;
```

La operación `in` *devuelve un código de error de* 0 *si fracasa. El receptor vuelve a intentar obtener el valor hasta que tiene éxito.*

Una *barrera* es un punto en la ejecución de un programa en el que un grupo de procesos se sincronizan esperando hasta que todos los miembros del grupo llegan a ese punto. Una aplicación de espacio de tuplas puede implementar la sincronización por barrera para *n* procesos insertando una tupla de barrera que contenga el valor entero *n* como parte de su inicialización. Cuando un proceso de la aplicación llega a la barrera, decrementa el valor de la tupla de barrera (`in`, decrementar, `out`) y luego espera a que el valor de la tupla de barrera llegue a cero (`rd`). El examen continuo de la tupla de barrera hasta que llega a cero es una forma de *dar vueltas*, una especie de ocupado en espera.

Ejemplo 15.9

El siguiente pseudocódigo Richard implementa una barrera suponiendo que en el espacio de tuplas hay una tupla con ID `barrier_ID` *y valor* n.

```
while(!in(barrier_ID, ?i))
    ;
i--;
out(barrier_ID, i);
while (!rd(barrier_ID, ?i) || (i != 0))
    ;
```

Ejemplo 15.10

El siguiente pseudocódigo Linda implementa la barrera del ejemplo 15.9 sin dar vueltas.

```
in(barrier_ID, ?i);
i--;
out(barrier_ID, i);
if (i > 0)
   rd(barrier_ID, 0);
```

Cada proceso del ejemplo 15.9 realiza una operación `in` y una `out` para decrementar la cuenta de la barrera. Una implementación alternativa inicializa la barrera insertando *n* tuplas idénticas. Cada proceso saca una tupla cuando llega a la barrera y luego da vueltas hasta que se acaban todas las tuplas de barrera.

Ejemplo 15.11

El siguiente pseudocódigo Richard implementa una barrera suponiendo que en el espacio de tuplas hay exactamente n *copias de una tupla con ID* `barrier_ID`.

```
in(barrier_ID, ?i);
while(rd(barrier_ID, ?i))
   ;
```

Aunque el ejemplo 15.11 requiere el mismo número de operaciones `out` que el 15.9, podría requerir menos accesos al espacio de tuplas una vez que se llega a la barrera y permite un mayor paralelismo porque las operaciones `in` no tienen que esperar a que otros procesos terminen de decrementar.

15.3 Un espacio de tuplas Richard sencillo

En esta sección especificaremos un servidor Richard con operaciones `out`, `in` y `rd`. Primero se creará una versión completamente local de Richard y luego se describirá la forma de convertir el sistema de modo que el espacio de tuplas tenga un servidor remoto.

15.3.1 Estructuras de datos de tuplas

Una tupla Richard consiste en un ID de tupla entero y un valor. Representaremos las tuplas en una estructura de tipo `struct tuple_t` definida por las declaraciones del programa 15.1. La unión `tuple_val_u` contiene el valor de la tupla. El miembro `tuple_type` indica el tipo del valor. Estas definiciones no contienen enunciados `typedef` a fin de facilitar el uso de `rpcgen` más adelante. Con objeto de evitar que la gestión de memoria se vuelva demasiado compleja y permitir la implementación en términos de RPC, se supone que los arreglos tienen un tamaño máximo de `MAX_BUF`.

Programa 15.1: Las declaraciones de tuplas para Richard.

```
#define MAX_BUF 1024
enum tuple_value_type {
   INTEGER = 0,
   ARRAY = 1
};

struct array_t {
   unsigned int array_len;
   char *array_val;
};

struct tuple_val_t {
   enum tuple_value_type  tuple_type;
   union {
      int    ival;
      struct array_t array;
   } tuple_val_u;
};

struct tuple_t {
   unsigned int tuple_ID;
   struct tuple_val_t tuple_val;
};
```

Programa 15.1

Richard tiene dos tipos de valores de tupla: enteros y arreglos de caracteres. Si el valor es entero, está en la tupla, pero las tuplas cuyos valores son arreglos sólo contienen apuntadores. El programa debe apartar espacio adicional para contener el arreglo real y establecer un apuntador dirigido a él.

Ejemplo 15.2

El siguiente segmento de código en C crea una tupla con ID 45 *para contener la cadena* `"This is a test"`.

```
"This is a test".

char *p = "This is a test";
struct tuple_t *t;

t = (struct tuple_t *)malloc(sizeof(struct tuple_t) +
                             strlen(p) + 1);
t->tuple_ID = 45;
t->tuple_val.tuple_type = ARRAY;
t->tuple_val.tuple_val_u.array.array_len = strlen(p) + 1;
t->tuple_val.tuple_val_u.array.array_val =
                            (char *)t + sizeof(struct tuple_t);
memcpy(t + sizeof(struct tuple_t), p, strlen(p) + 1);
```

La llamada `malloc` del ejemplo 15.12 aparta espacio tanto para `struct tuple_t` como para los datos del arreglo. Posteriormente, se podrá usar una sola llamada `free` para liberar la memoria ocupada por ambos elementos. Utilice esta técnica de asignación de bloques para apartar espacio para los nodos que contienen tuplas en la tabla de dispersión que describiremos en seguida.

15.3.2 Representación del espacio de tuplas

Representaremos el espacio de tuplas Richard mediante una tabla de dispersión como la que se muestra en la figura 15.1. Las tuplas cuyos ID se dispersan al mismo valor se guardan en una lista enlazada a la cual apunta la entrada correspondiente de la tabla de dispersión. Las tuplas con el mismo ID en la lista enlazada se hacen una cola de modo que `in` y `rd` puedan recuperarlas según un régimen de primeras entradas-primeras salidas (FIFO). Mantenga un contador con el ID de tupla más grande que se haya asignado (`largest_ID`) a fin de asignar un ID desocupado a una tupla, si es necesario.

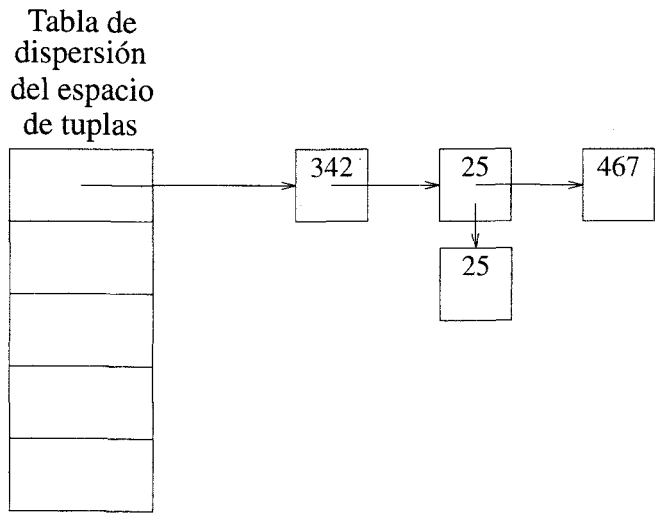


Figura 15.1: Representación de tabla de dispersión para el espacio de tuplas Richard.

Ejemplo 15.13

En la tabla de dispersión de la figura 15.1 las tuplas con valores clave `342`, `25` *y* `467` *se dispersan a la misma posición. Las dos tuplas con ID* `25` *se encolan.*

Escoja una representación de tabla de dispersión que sea eficiente para las búsquedas Richard (esto es, buscar por ID de tupla y buscar por tipo de tupla). Para efectuar la búsqueda por ID de tupla, calcule la entrada de la tabla de dispersión con base en el ID de la tupla y luego realice un examen buscando las tuplas que contengan ese ID.

En aplicaciones de búsqueda aleatoria como el problema de retroceso de la sección 15.4, la aleatoriedad (o falta de ésta) de la búsqueda por tipo de tupla puede afectar la solución que se obtenga. Si se mantienen cuentas de la cantidad de tuplas de cada tipo que se dispersan a cada valor, tal vez podría mejorarse el rendimiento de las búsquedas aleatorias por tipo de tupla. El programa puede aproximar una búsqueda aleatoria seleccionando primero una entrada de la tabla de dispersión al azar. Si la entrada contiene tuplas del tipo especificado, se utilizan las cuentas de la entrada de la tabla de dispersión para escoger al azar una tupla del tipo en cuestión.

La especificación supone que los ID de tupla nuevos son únicos durante la existencia del sistema. Es decir, si una operación `out` utiliza un valor de 0 para `tuple_ID`, se asignará a la tupla un ID que no se haya utilizado antes. Un `tuple_ID` de 32 bits es suficiente para realizar pruebas pero no para un sistema real. Una representación más realista podría incluir un número de encarnación. Esta representación equivale a mantener un ID de tupla de 64 bits.

15.3.3 Operaciones de espacio de tuplas

Implemente las operaciones del espacio de tuplas Richard `out`, `in` y `rd` como se describe a continuación. Coloque las estructuras de datos de la tabla de dispersión junto con las tres funciones de Richard en un archivo aparte. Escriba un programa principal de prueba que ejecute diversas combinaciones de llamadas a `out`, `in` y `rd` para verificar que la implementación local funciona correctamente.

- La función `out` inserta la tupla especificada en el espacio de tuplas. La función `out` tiene el prototipo

  ```
  unsigned int out(struct tuple_t val);
  ```

  Si `out` tiene éxito, devuelve el ID de la tupla insertada; si no, devuelve 0. Si el valor `val.tuple_ID` es positivo, la tupla se encola detrás de todas las demás tuplas existentes con el mismo valor de ID de tupla. Si el ID es mayor que `largest_ID`, se incrementa `largest_ID` a este valor. Si `val.tuple_ID` es 0, se asigna a la tupla un nuevo valor de ID igual a `largest_ID` más uno y se incrementa `largest_ID`.

- La función `in` saca una tupla que coincide con la plantilla especificada y devuelve el valor de la tupla. La función `in` tiene el prototipo

  ```
  struct tuple_t in(struct tuple_t val);
  ```

  La plantilla se basa en el valor de `val.tuple_ID`. Si este valor es positivo, se busca la primera tupla que tenga ese ID de tupla. Si `val.tuple_ID` es 0, se busca una tupla al azar del tipo `val.tuple_val.tuple_type` (idealmente, una tupla aleatoria uniformemente distribuida). Si ninguna tupla coincide con la plantilla, el miembro `tuple_ID` de la tupla devuelta será 0.

- La función `rd` se comporta igual que `in` excepto que no saca la tupla del espacio de tuplas. La función `rd` tiene el prototipo

```
struct tuple_t rd(struct tuple_t val);
```

A primera vista, estas especificaciones son de una engañosa sencillez, tanto así que podríamos sentirnos tentados a cambiar los valores de retorno de `in` y `rd` a apuntadores a `struct tuple_t` en lugar de las estructuras mismas. Resista la tentación. (También examine el servicio de números pseudoaleatorios mejorado que se especifica en el ejemplo 14.9.) La versión remota no puede devolver un apuntador al cliente, así que es mejor resolver el problema en la versión local desde un principio.

El problema se debe a que la definición de `struct tuple_t` tiene un apuntador a un arreglo en lugar del arreglo en sí. Si pasamos el parámetro `val` por valor en el programa invocador, el programa invocado podrá encontrar la cadena si es necesario. Si una operación `id` o `rd` devuelve una tupla de cadena, ¿dónde deberá guardarse la cadena?

Ejemplo 15.14

El siguiente segmento de código invoca la función `out` *para insertar un arreglo que contiene* `"Esta es una prueba"` *en el espacio de tuplas.*

```
struct tuple_t myval;
char myarray[MAX_BUF];

strcpy(myarray, "esta es una prueba");
myval.tuple_ID = 0;
myval.tuple_val.tuple_type = ARRAY;
myval.tuple_val.tuple_val_u.array.array_len = strlen(myarray) + 1;
myval.tuple_val.tuple_val_u.array.array_val = myarray;
if (!out(myval))
   fprintf(stderr, "Out failed\n");
```

Puesto que la tupla del ejemplo 15.14 es de tipo `ARRAY`, la función `out` realiza una sola asignación de bloque de un `struct hash_node` con suficiente espacio para contener los valores del arreglo. A continuación, `out` copia los valores `struct tuple_t`, establece explícitamente su apuntador `array_val` de modo que haga referencia al espacio adicional apartado junto con su nodo y ejecuta `memcpy` para copiar el arreglo al que se apunta en el parámetro en su propio espacio, de forma similar a la asignación del ejemplo 15.12. Después de la conversión a llamada remota, el parámetro `struct tuple_t` de `out` hará referencia a una variable estática en el talón de servidor.

La operación `in` busca en el espacio de tuplas un nodo que coincida con la plantilla de búsqueda. Una vez que lo encuentra, copia la porción `struct tuple_t` del nodo en un valor de retorno estático especial y libera el nodo. El programa invocador debe cerciorarse de copiar el valor del arreglo en su propio espacio antes de invocar otra vez a `in`, pues esta operación reutilizará el arreglo estático en la siguiente llamada de ese tipo. El problema es similar a los ocasionados por funciones de biblioteca que utilizan almacenamiento estático, como se explicó en la sección 1.5.

Ejemplo 15.15

El siguiente segmento de código que forma parte de `in` *maneja los valores de retorno de la versión local.*

```
static struct tuple_t return_tuple;
static char return_array[MAX_BUF];

/* dentro de la función in */
struct hash_node *p;

if ((p = find_and_remove(val)) == NULL) {
    return_tuple.tuple_ID = 0;
    return return_tuple;
}
return_tuple = p->tuple;
if (return_tuple.tuple_val.tuple_type == ARRAY) {
   return_tuple.tuple_val.tuple_val_u.array.array_val = return_array;
   memcpy(return_array,
          p->tuple.tuple_val.tuple_val_u.array.array_val,
          p->tuple.tuple_val.tuple_val_u.array.array_len);
}
free(p);
return return_tuple;
```

15.3.4 Conversión a un servidor remoto

Una vez que funcione la implementación local del espacio de tuplas, conviértala en un servidor al que se accede mediante llamadas a procedimiento remoto. Las funciones locales `out`, `in` y `rd` ahora contienen llamadas remotas al servidor de tuplas. Pruebe el servidor para asegurarse de que está funcionando. Diseñe formatos apropiados para pasar tuplas como se explicó en el capítulo 14.

Un área delicada es la unión `tuple_val_u` en la estructura `struct tuple_val_t` porque el lenguaje de especificación de RPC utiliza una unión conmutada en lugar de una unión. Sustituya la definición de `struct tuple_t` por algo equivalente a

```
/* Lenguaje de especificación de RPC de Sun. No es C */
const MAX_BUF = 1024;

enum tuple_value_type {
    INTEGER = 0,
    ARRAY = 1
};

union tuple_val_t switch(enum tuple_value_type tuple_type) {
    case INTEGER:
        int ival;
    case ARRAY:
        char array<MAX_BUF>;
};
```

```
struct tuple_t {
    unsigned int tuple_ID;
    struct tuple_val tuple_val;
};
```

El programa `rpcgen` utiliza la información de la unión conmutada para generar una estructura ordinaria. Es preciso fijar un límite superior explícito para el tamaño del arreglo de caracteres (`MAX_BUF`). La sintaxis de unión conmutada y los arreglos de tamaño variable no forman parte de C, sino del lenguaje de especificación de RPC de Sun. Esta sintaxis aparece en los archivos `.x`, y `rpcgen` genera código C a partir de la especificación. El programa 15.2 es un archivo de especificación para el servidor Richard. La estructura `struct array_t` ahora es un arreglo de caracteres de longitud variable.

Programa 15.2: El archivo de especificación `richard.x` para un servidor Richard remoto.

```
/*  richard.x: Lenguaje de especificación de RPC de Sun. No es C */

const MAX_BUF = 1024;

enum tuple_value_type {
    INTEGER = 0,
    ARRAY = 1
};

union tuple_val_t switch(enum tuple_value_type tuple_type) {
    case INTEGER:
        int ival;
    case ARRAY:
        char array<MAX_BUF>;
};

struct tuple_t {
    unsigned int tuple_ID;
    struct tuple_val tuple_val;
};

program RICHPROG {
    version RICHVERS {
        int OUT(tuple_t) = 1;
        tuple_t IN(tuple_t) = 2;
        tuple_t RD(tuple_t) = 3;
    } = 1;
} = 0x31234566;
```

Programa 15.2

Ejemplo 15.16

El comando

```
rpcgen -a -C richard.x
```

crea varios archivos de esqueleto y un archivo de cabecera `richard.h` *que incluye las siguientes definiciones de tipos:*

```
#define MAX_BUF 1024

enum tuple_value_type {
        INTEGER = 0,
        ARRAY = 1
};
typedef enum tuple_value_type tuple_value_type;

struct tuple_val_t {
        tuple_value_type tuple_type;
        union {
                int ival;
                struct {
                        u_int array_len;
                        char *array_val;
                } array;
        } tuple_val_t_u;
};
typedef struct tuple_val_t tuple_val_t;

struct tuple_t {
        u_int tuple_ID;
        struct tuple_val tuple_val;
};
typedef struct tuple_t tuple_t;
```

En el ejemplo 15.16 la unión conmutada de la especificación del programa 15.2 es reemplazada por una estructura que contiene una unión ordinaria. Observe que en estas declaraciones no aparece `MAX_BUF` explícitamente. Tenga cuidado de no exceder este límite en las tuplas con valor de arreglo porque los talones de cliente y de servidor pueden utilizar este límite al apartar espacio de *buffer* temporal.

Convierta la implementación local del espacio de tuplas en un servidor remoto al que se accede mediante llamadas de procedimiento remoto. El proceso de conversión no deberá presentar problemas si la versión local se ajusta a la especificación. Una vez que el servidor remoto esté funcionando correctamente, implemente los siguientes programas para efectuar una comunicación en orden, sin errores, a través del espacio de tuplas.

- Escriba un programa `sender` (emisor) que lea de la entrada estándar e inserte cada bloque de datos en el espacio de tuplas invocando a `out` con un valor de ID de tupla igual a `commun_ID`. Asigne arbitrariamente a `commun_ID` un valor previamente

convenido, análogo a los puertos bien conocidos cuando se usan *sockets*. El programa `sender` utiliza un tipo de tupla `ARRAY` para la transmisión. Cuando `sender` detecta un fin de archivo en la entrada estándar, invoca por última vez `out` con una tupla de tipo `INTEGER`. El valor de `commun_ID` se pasa como argumento de línea de comandos a `sender`.

- Escriba un programa `receiver` (receptor) que realice operaciones `in` para recuperar tuplas cuyo ID sea `commun_ID` del espacio de tuplas. Si una tupla es del tipo `ARRAY`, `receiver` escribe los datos recuperados en la salida estándar. Si la tupla es del tipo `INTEGER`, `receiver` envía un mensaje al error estándar y termina. El valor de `commun_ID` se pasa como argumento de línea de comandos a `receiver`. No olvide reintentar las operaciones `in` fallidas, pues no son bloqueadoras.

- Pruebe la implementación del espacio de tuplas utilizando los programas `sender` y `receiver` para transferir un archivo a través del espacio de tuplas. Por ejemplo, si quiere transferir el archivo `my.in` utilizando el ID de tupla `348` como puerto de comunicación, utilice

```
sender 348 <my.in
receiver 348 >my.out
diff my.in my.out
```

  Tanto `sender` como `receiver` aceptan el valor de `commun_ID` como argumento de línea de comandos. Ejecute `sender` y `receiver` en diferentes máquinas, o por lo menos en dos ventanas distintas. Cuando terminen los procesos, ejecute `diff` para comprobar que la transferencia se haya efectuado correctamente. Pruebe a ejecutar varios pares de emisor-receptor simultáneamente. Cada par tendrá un `commun_fd` distinto.

Como segunda prueba, implemente una barrera de Richard. Pruebe la barrera con por lo menos 10 clientes ficticios.

15.4 Pizarrones: Una aplicación de espacio de tuplas

En una comunidad o sistema social, los miembros interactúan, compiten por los recursos y cooperan para resolver problemas. Los investigadores han aplicado la analogía de los agentes cooperativos a sistemas de procesos distribuidos que buscan resolver problemas difíciles como las búsquedas heurísticas [39]. Huberman [40] ha demostrado que, en condiciones apropiadas, puede haber un incremento no lineal en el rendimiento del sistema global si los agentes cooperan. (Una dependencia lineal implica que una gráfica de rendimiento contra número de agentes tiene una curva con pendiente positiva constante. Huberman argumenta que la curva puede tener una pendiente positiva creciente.) Además, incrementar la diversidad de los agentes aumenta las posibilidades de lograr una solución sobresaliente.

A fin de comprender cómo funciona este enfoque de resolución de problemas, considere la situación en la que un grupo de personas participa en una sesión de "lluvia de ideas" para resolver un problema. El jefe del grupo se para junto a un pizarrón y transcribe ideas conforme los participantes las van gritando. Los beneficios de la interacción se obtienen cuando las ideas de una persona hacen que otros participantes exploren una línea de razonamiento inesperada. Si el jefe del grupo dirige la discusión o presenta al principio sugerencias de posibles estrategias, las ideas tienden a enfocarse mejor. Una discusión menos dirigida conduce a ideas más desperdigadas, pero también podría dar pie a enfoques más creativos para resolver el problema.

En términos más cuantitativos, la discusión dirigida lleva a la mayoría de los participantes a soluciones similares, y la distribución de la calidad de las soluciones suele tener un margen angosto. Si se adopta un enfoque más diverso y menos estructurado, la distribución de las soluciones seguramente será más amplia. Es muy posible que algunos miembros propongan soluciones muy deficientes, pero también podrían surgir soluciones sobresalientes. Puesto que sólo se necesita una solución, es más probable que se obtenga una solución sobresaliente en el segundo caso. Los elementos clave de la estrategia son:

- Un problema en el que los resultados parciales de un agente puede guiar los cálculos de otro agente.
- Suficiente diversidad para que la resolución del problema sea imaginativa.
- Suficiente complejidad de los agentes para que puedan lograr avances sin consultar el pizarrón con demasiada frecuencia.

Las técnicas de resolución cooperativa de problemas tienen varias características en común con los algoritmos genéticos.

En esta sección desarrollaremos un enfoque cooperativo para resolver el problema de las *n* reinas y problemas de optimización similares utilizando un pizarrón que se implementa con un espacio de tuplas. Programas llamados *agentes* obtienen configuraciones prometedoras de un pizarrón central y utilizan retroceso aleatorizado en un intento por calcular soluciones. Además, los agentes depositan soluciones parciales en el pizarrón para que otros agentes tengan oportunidad de extenderlas. En la figura 15.2 se muestra la disposición del servidor de pizarrón. Los agentes utilizan *out* para depositar soluciones parciales e *in* o *rd* para recuperarlas. No es necesario que los agentes sean idénticos; los algoritmos de resolución cooperativa de problemas a menudo funcionan mejor con una colección diversificada de agentes.

15.4.1 El problema de las *n* reinas

En ajedrez, una reina *amenaza* otra pieza si esta última está en la misma fila o columna o en la misma diagonal que la reina y no hay piezas que se interpongan. El problema clásico de las *ocho reinas* consiste en colocar ocho reinas en un tablero de ajedrez de 8×8 de modo que ninguna de las reinas amenace otra. Puesto que cada reina debe aparecer exactamente una vez en cada fila y cada columna, las posibles soluciones son permutaciones de los enteros de 0 a 7.

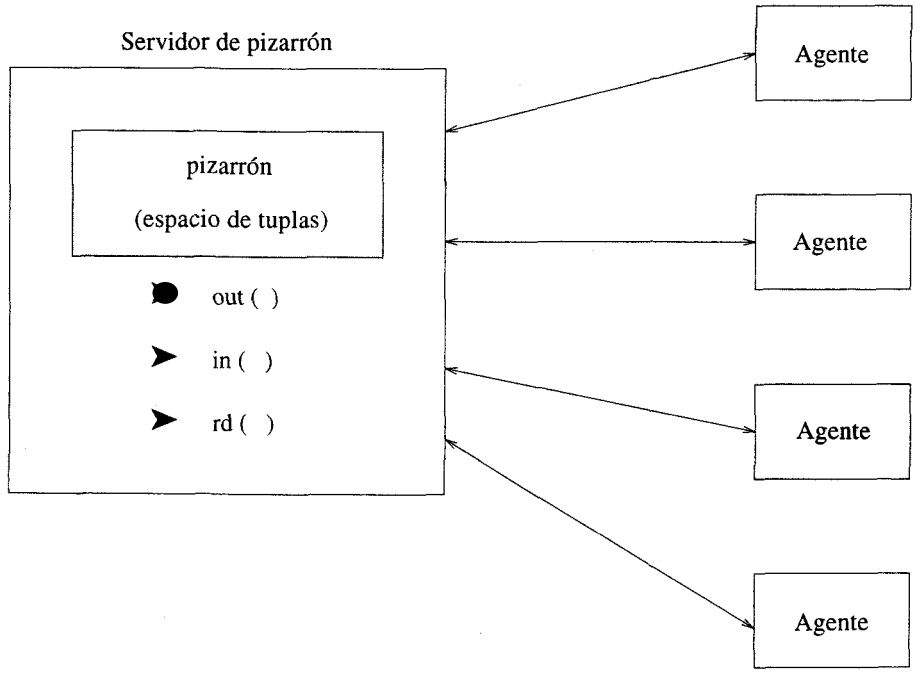


Figura 15.2: Pizarrón central con agentes remotos.

¡El número de permutaciones de los enteros de 0 a 7 es 8! = 40,320, así que resulta fácil encontrar las 92 soluciones si probamos cada una de las permutaciones para determinar si dos o más reinas están en la misma diagonal.

Ejemplo 15.17

La permutación (3, 5, 7, 1, 6, 0, 2, 4) *es una solución del problema de las ocho reinas. Aquí las filas y columnas son enteros entre* 0 *y* 7. *La permutación corresponde a la siguiente colocación de las reinas.*

Fila	*Columna*
0	*3*
1	*5*
2	*7*
3	*1*
4	*6*
5	*0*
6	*2*
7	*4*

El problema más general de colocar n reinas en un tablero de ajedrez de $n \times n$ es más difícil. Incluso con $n = 20$, la idea de probar todas las permutaciones de n 20 = 2, 432, 902, 008, 176, 640, 000 desanima a cualquiera. Los algoritmos probabilísticos han demostrado ser útiles para encontrar soluciones cuando n es grande. En lugar de iniciar una búsqueda sistemática, muchos algoritmos probabilísticos colocan algunas reinas al azar y luego aplican un algoritmo determinista, como el retroceso o la selección ávida [13].

En este proyecto, los agentes cooperan para realizar una búsqueda probabilística. Para la estrategia cooperativa se necesita un método mediante el cual un agente que está trabajando con el problema pueda comunicar información útil a otros agentes. Aquí los agentes anuncian las configuraciones más prometedoras que logran calcular. Una configuración *prometedora* es una colocación de k reinas ($0 < k \leq n$) en el tablero de modo que ninguna reina amenace a otra. Representaremos una configuración prometedora mediante un *arreglo de colocación* unidimensional. Una entrada, j, en la i-ésima posición del arreglo corresponde a la colocación de una reina en la i-ésima fila y la j-ésima columna del tablero. Las posiciones de fila que no estén en la configuración prometedora contendrán el valor –1.

Ejemplo 15.8

La configuración (3, –1, 7, 2, 6, –1, –1, –1) *es una colocación no amenazante de cuatro reinas en un tablero de* 8 × 8. *Ésta es una configuración prometedora. En la figura 15.3 se muestra la configuración.*

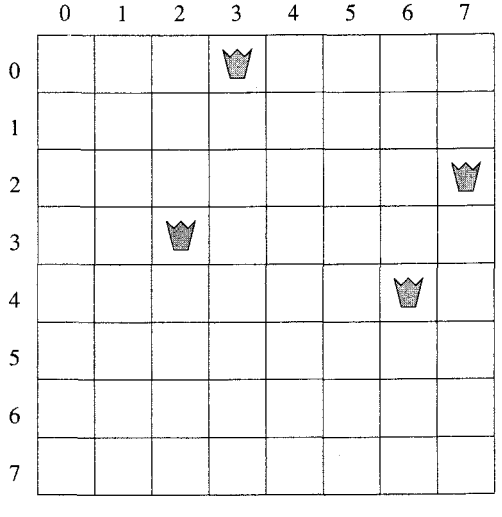


Figura 15.3: La configuración prometedora (3, –1, 7, 2, 6, –1, –1, –1).

No todas las permutaciones de n son soluciones del problema de las n reinas. De forma similar, no hay garantía de que un subconjunto arbitrario de una permutación sea una configu-

ración prometedora. La representación de arreglo unidimensional de las configuraciones prometedoras garantiza que nunca habrá dos piezas en la misma fila o columna, pero no evita que haya más de una pieza en una misma diagonal.

Ejemplo 15.9

La configuración (0, 1, 2, 3, 4, –1, –1, 7) *representa una colocación de seis reinas a lo largo de la diagonal principal del tablero en el problema de las ocho reinas. Esta configuración no es prometedora.*

La fila i del tablero está desocupada si la i-ésima entrada del arreglo de colocación contiene –1. De forma similar, una columna j del tablero está desocupada si el valor j no aparece en el arreglo de colocación. La información de ocupación de las diagonales no puede obtenerse directamente del arreglo de colocación.

Las diagonales paralelas a la diagonal principal se denominan *diagonales delanteras*. En una diagonal delantera dada, la diferencia entre el número de fila y de columna es constante. Dos reinas colocadas en las posiciones (i_1, j_1) e (i_2, j_2) ocupan la misma diagonal delantera si

$$(i_1 - j_1 + n - 1) = (i_2 - j_2 + n - 1)$$

Se suma el valor $n - 1$ a cada diferencia para que el resultado quede dentro del intervalo $[0, 2n - 2]$.

Las *diagonales traseras*, que son perpendiculares a la diagonal principal, se caracterizan por un valor constante de $i + j$. Estos valores también están dentro del intervalo $[0, 2n - 2]$. Dos reinas colocadas en las posiciones (i_1, j_1) e (i_2, j_2) ocupan la misma diagonal trasera si

$$(i_1 + j_1) = (i_2 + j_2)$$

Un *arreglo de ocupación* proporciona información útil sobre cuáles posiciones están ocupadas. Cada entrada representa una posición. Un valor de 1 en una entrada indica que la posición está ocupada y un valor de –1 indica que está desocupada. La ocupación de filas y columnas de una colocación se puede representar mediante arreglos de tamaño n, y la ocupación de las diagonales delanteras y de las traseras mediante arreglos de tamaño $2n - 1$.

Ejemplo 15.20

La configuración prometedora (3, –1, 7, 2, 6, –1, –1, –1) *tiene los siguientes arreglos de ocupación.*

filas:	(1, –1, 1, 1, 1, –1, –1, –1)
columnas:	(–1, –1, 1, 1, –1, –1, 1, 1)
diagonales delanteras:	(–1, –1, 1, –1, 1, 1, –1, –1, 1, –1, –1, –1, –1, –1, –1)
diagonales traseras:	(–1, –1, –1, 1, –1, 1, –1, –1, –1, 1, 1, –1, –1, –1,–1)

Obtenemos el arreglo de ocupación de filas a partir del arreglo de colocación sustituyendo todas las entradas no negativas por 1. Cabe señalar que en una configuración prometedora

cada uno de los arreglos de ocupación contiene tantos unos como reinas hay en la configuración. Representaremos las configuraciones prometedoras mediante una estructura del siguiente tipo:

```
typedef struct promising_type {
    int numberfixed;
    int boardsize;
    int values[MAXCHESSBOARDSIZE];
    int columns[MAXCHESSBOARDSIZE];
    int for_diagonals[2*MAXCHESSBOARDSIZE-1];
    int back_diagonals[2*MAXCHESSBOARDSIZE-1];
} promising_t;
```

`MAXCHESSBOARDSIZE` es una constante que define el tamaño máximo que puede tener el tablero. El miembro `boardsize` (tamaño del tablero) define el tamaño del problema y es 8 para el problema de las ocho reinas. El miembro `numberfixed` corresponde al número de reinas que ya están colocadas en posiciones no amenazantes. El miembro `values` (valores) representa la colocación, mientras que `columns`, `for_diagonals` y `back_diagonals` representan los arreglos de ocupación de las columnas, las diagonales delanteras y las diagonales traseras, respectivamente.

La función `add_if_promising` (añadir si es prometedora) del programa 15.3 agrega una reina en la posición (i, j) a una configuración prometedora `*rp` si la configuración resultante va a seguir siendo prometedora; devuelve 1 si agregó la reina y 0 si no lo hizo.

Programa 15.3: Agrega una reina a una configuración prometedora.

```
int add_if_promising(promising_t *rp, int i, int j)
{
   int f, b;

   f = (i - j + rp->boardsize - 1);
   b = (i + j);
   if ((rp ->values[i] == -1) && (rp ->columns[j] == -1) &&
       (rp->for_diagonals[f] == -1) &&
       (rp->back_diagonals[b] == -1)) {
       rp->columns[j] = 1;
       rp->for_diagonals[f] = 1;
       rp->back_diagonals[b] = 1;
       rp->values[i] = j;
       (rp->numberfixed)++;
       return 1;
   }
   return 0;
}
```

Programa 15.3

15.4.2 Retroceso ávido

Los algoritmos ávidos procuran maximizar las mejoras en cada paso sin tener en cuenta las posibles consecuencias futuras negativas. Un algoritmo ávido para el problema de las *n* reinas es:

- Escoger una fila desocupada al azar.
- Tratar de colocar una reina no amenazante en esa fila examinando todas las posiciones de columna desocupadas, comenzando por una columna desocupada al azar.
- Repetir hasta que la colocación represente una solución completa o resulte imposible colocar una reina en la fila seleccionada.

Los algoritmos ávidos pueden meterse en callejones sin salida y nunca encontrar una solución, y es por ello que a menudo se utiliza el retroceso para aumentar la probabilidad de llegar a una solución. El retroceso ávido obtiene una solución prometedora, elimina cierto número de reinas al azar y luego intenta extender la solución prometedora aplicando un algoritmo ávido.

El programa 15.4 implementa un algoritmo ávido aleatorizado para el problema de las *n* reinas. Este programa acepta dos argumentos de línea de comandos que especifican el tamaño del tablero y el número de iteraciones del algoritmo ávido que se efectuarán. La función `get_random_promising` produce una configuración prometedora intentando colocar reinas en filas elegidas al azar. La función `compute_queens` (calcular reinas) implementa el algoritmo de retroceso ávido que acabamos de describir. La constante `MAXDIVERSITY` (diversidad máxima) controla el nivel de retroceso que se aplicará. La función `post_promising` (asentar configuración prometedora) se utilizará posteriormente para insertar el resultado en el pizarrón en la implementación cooperativa, y `print_promising` envía una configuración prometedora al archivo especificado.

Programa 15.4: Programa que utiliza un algoritmo ávido aleatorizado para buscar soluciones del problema de las *n* reinas.

```
#include "greedy.h"
#define STARTSEED 400
/* El agente aleatoriza una sugerencia para obtener una
 * nueva configuración prometedora
 */
void main(int argc, char *argv[])
{
   struct promising_type r;
   int diversity;
   int i, iterations;

   if (argc != 4) {
      fprintf(stderr, "Usage: %s boardsize iterations\n", argv[0]);
      exit(1);
   }
   r.boardsize = atoi(argv[1]);
   iterations = atoi(argv[2]);
```

```
/* Inicializar después el generador de números pseudoaleatorios utilizando el PID */
   initialize_random(STARTSEED);
   for (i = 0; i < iterations; i++) {
      diversity = random_next(MAXDIVERSITY);
      get_random_promising(&r);
      compute_queens(&r, diversity);
      post_promising(r);
      if (r.numberfixed == r.boardsize) {
         fprintf(stdout, "\nA solution is: \n");
         print_promising(stdout, r);
      }
   }
}
```

Programa 15.4

Desarrolle y depure el algoritmo de retroceso ávido en un entorno no distribuido. Obtenga configuraciones prometedoras mediante la colocación aleatoria de reinas. Escriba las siguientes funciones a las que hace referencia el programa principal del programa 15.4. Coloque sus prototipos en un archivo de cabecera llamado `greedy.h`.

- La función `initialize_random` inicializa un generador de números pseudoaleatorios invocando a `srand48`. El prototipo es

  ```
  void initialize_random(int seed);
  ```

- La función `random_next` (siguiente aleatorio) genera números pseudoaleatorios en el intervalo de 0 a `upper - 1`. El prototipo es

  ```
  int random_next(int upper);
  ```

  `random_next` invoca a `drand48` para generar un `double` y luego lo convierte en un `int` que esté dentro del intervalo correcto.

- La función `get_random_promising` construye una configuración prometedora al azar. El prototipo es

  ```
  void get_random_promising(promising_t *rp);
  ```

  Aquí `rp` apunta a la configuración prometedora al azar que resulta. Escoja el número de entradas que fijará en la configuración al azar, pero no permita que el valor sea demasiado grande en comparación con el tamaño del tablero, pues en tal caso `get_random_promising` acabaría realizando una búsqueda exhaustiva del espacio de soluciones.

- La función `compute_queens` modifica una configuración prometedora empleando el algoritmo de retroceso ávido. El prototipo es

  ```
  void compute_queens(promising_t *rp, int diversity);
  ```

Aquí `rp` es un apuntador a la configuración prometedora. Haga que esta configuración sea aleatoria eliminando `diversity` entradas y extienda la configuración hasta donde sea posible utilizando el método ávido. La configuración prometedora modificada es una solución si `rp->numberfixed` es igual a `rp->boardsize`.

- La función `post_promising` es sólo un talón a estas alturas.
- La función `print_promising` envía una configuración prometedora al archivo indicado. El prototipo es

```
void print_promising(FILE *f, promising_t r);
```

Pruebe el programa con varios tamaños de tablero y números de iteraciones distintos. Verifique que el programa calcule correctamente las configuraciones prometedoras. Sustituya la constante `STARTSEED` con la que se inicializa el generador de números pseudoaleatorios en el programa principal por una llamada a `getpid` a fin de introducir variabilidad en las pruebas. Grafique el número de soluciones obtenidas contra el tamaño del tablero para 10,000 iteraciones variando el tamaño del tablero entre 4 y 32 en incrementos de 4.

15.4.3 Pizarrones y agentes

Un *pizarrón* es un depósito de información que los agentes pueden leer y en el cual pueden escribir; es un método de comunicación controlada entre los agentes. Se han utilizado pizarrones para organizar la resolución de muchos problemas de inteligencia artificial con bases en conocimientos. En fechas más recientes, los pizarrones han aparecido como mecanismos generales de comunicación entre procesos. El Blackboard Technology Group [14], por ejemplo, ha implementado una capa de soporte de aplicaciones distribuidas en la que cada nodo de una red mantiene un pizarrón. La computación y conferencias en grupos de trabajo son otras técnicas interactivas de resolución de problemas relacionadas con pizarrones que han adquirido popularidad en últimas fechas [56].

Es frecuente implementar los pizarrones con estructuras de datos de árbol complicadas a fin de organizar cantidades enormes de información interrelacionada de forma eficiente. Este proyecto representa un pizarrón mediante un espacio de tuplas. Los agentes asientan y solicitan configuraciones prometedoras para volverlas aleatorias y extenderlas.

Agregue un pizarrón que mantenga un conjunto de configuraciones prometedoras para el uso del programa ávido de *n* reinas. Implemente el pizarrón como un espacio de tuplas extendiendo las tuplas permisibles de la sección 15.3 para incluir arreglos de colocación. Implemente las siguientes funciones:

- La función `get_hint` (obtener sugerencia) obtiene una configuración prometedora del pizarrón. El prototipo es

```
promising_t get_hint(void);
```

Utilice `in` para obtener una configuración prometedora al azar del pizarrón y convierta la representación de arreglo de colocación de la configuración prometedora

en un `promising_t`. La función `get_hint` devuelve una configuración prometedora que no contiene reinas si la operación `in` no logra devolver un arreglo de colocación.

- La función `post_promising` agrega una configuración prometedora al espacio de tuplas invocando a `out`. El prototipo es

```
int post_promising(promising_t s);
```

Extraiga un arreglo de colocación de la configuración prometedora `s`, conviértalo en tupla e invoque a `out` para colocarla en el espacio de tuplas. La función `post_promising` devuelve 0 si la operación se realizó con éxito y –1 en caso contrario.

Sustituya la llamada a `get_random_promising` del programa principal del programa 15.4 por una llamada a `get_hint`. Añada código al programa principal para agregar un número especificado de configuraciones prometedoras al azar al espacio de tuplas antes de iniciar el algoritmo ávido.

Pruebe el programa utilizando `STARTSEED` como valor inicial para el generador de números pseudoaleatorios. Después de depurar el programa, sustituya `STARTSEED` por una llamada a `getpid`. Ejecute el programa con diferentes tamaños de tablero, diversidades y números de iteraciones. Grafique el número de soluciones halladas en función de la diversidad para diferentes tamaños de tablero.

Pruebe el sistema con una colección de agentes ejecutándose en diferentes máquinas. Instrumente los agentes de modo que mantengan un registro del tiempo que dedican a buscar un número especificado de configuraciones prometedoras. Grafique el tiempo de ejecución medio contra el número de agentes cuando cada agente se ejecuta en una máquina distinta. Compare los resultados con el tiempo de ejecución cuando todos los agentes están en la misma máquina pero el servidor del espacio de tuplas está en otra máquina. Compare también los resultados cuando el servidor y todos los agentes están en la misma máquina.

15.5 Tuplas activas en Richard

Linda utiliza tuplas activas para evaluar funciones. La tupla activa aplica una función especificada a sus argumentos e inserta el resultado en el espacio de tuplas. La semántica de la operación `eval` de Linda no es fácil de implementar sin apoyo por parte del compilador. Tampoco resulta cómoda para implementar un sistema distribuido tipo UNIX. En esta sección exploraremos algunos aspectos de las tuplas activas con la semántica de Richard, comenzando con la implementación más sencilla de una tupla activa.

15.5.1 Diseño simplificado

Incorpore una función `eval` sencilla en Richard como sigue. La función `eval` bifurca un hijo para ejecutar mediante `execvp` una línea de comandos especificada. El prototipo de la función es:

```
unsigned int eval(struct tuple_t val);
```

La función `eval` devuelve 1 si la bifurcación con `fork` tuvo éxito o 0 si hay algún error; no comprueba que `execvp` haya tenido éxito ni inserta resultados en el espacio de tuplas. La función `eval` debe invocar a `waitpid` con la opción `WNOHANG` en un ciclo a fin de eliminar cualesquier hijos creados en ejecuciones previas de `eval` que nadie esté esperando.

El programa 15.5 muestra la declaración de `struct tuple_val_t` que incluye un tipo de tupla de función. El valor de la tupla de función no es más que una cadena de caracteres que proporciona la línea de comandos que se ejecutará con `exec`. La longitud máxima de la cadena es `MAX_STR`.

Programa 15.5: Declaraciones de tuplas para una implementación local de Richard con una función `eval`.

```
#define MAX_BUF 1024
#define MAX_STR 256
enum tuple_value_type {
    INTEGER = 0,
    ARRAY = 1,
    FUNCTION = 2
};

struct array_t {
    unsigned int array_len;
    char *array_val;
};

struct tuple_val_t {
    enum tuple_value_type  tuple_type;
    union {
       int    ival;
       struct array_t array;
       char  *func;
    } tuple_val_u;
};

struct tuple_t {
    unsigned int tuple_ID;
    struct tuple_val_t tuple_val;
};
```

Programa 15.5

Ejemplo 15.21

El siguiente segmento de código en C invoca a `eval` *para ejecutar el comando* `ls -l`.

```
char *cmd = "ls -l";
struct tuple_t val;
```

```
    val.tuple_ID = 0;
    val.tuple_val.tuple_type = FUNCTION;
    val.tuple_val.tuple_val_u.func = cmd;
    if (!eval(val))
       fprintf(stderr, "The eval of %s failed\n", cmd);
```

Invoque la función `eval` con varias utilerías estándar como `ls` y `cat` como comandos. Haga la conversión a un servidor remoto utilizando un archivo de especificación como el del programa 15.6. Utilice un número de versión de `2` para el nuevo servicio Richard.

Programa 15.6: Especificación del servidor Richard con servicio `eval`.

```
/*        richard.x - versión 2      */
/*        Lenguaje de especificación RPC.  No es C.  */

const MAX_BUF = 1024;
const MAX_STR = 256;
enum tuple_value_type {
    INTEGER = 0,
    ARRAY = 1,
    FUNCTION = 2
};

union tuple_val_t switch(enum tuple_value_type tuple_type) {
    case INTEGER:
        int ival;
    case ARRAY:
        char array<MAX_BUF>;
    case FUNCTION:
        string func<MAX_STR>;
};

struct tuple_t {
    unsigned int tuple_ID;
    struct tuple_val_t tuple_val;
};


program RICHPROG {
    version RICHVERS {
        unsigned int OUT(tuple_t) = 1;
        tuple_t IN(tuple_t) = 2;
        tuple_t RD(tuple_t) = 3;
        unsigned int EVAL(tuple_t) = 4;
    } = 2;
} = 0x31234566;
```

Programa 15.6

Cuando el servidor remoto ejecuta `eval`, los comandos ejecutados con `exec` envían sus resultados a la salida estándar del servidor y esperan entradas de la entrada estándar del servidor. Si el servidor ejecuta varios comandos en sucesión rápida, sus entradas y salidas quedarán intercaladas de forma similar a como sucede con la cadena de procesos del programa 2.12. Otro problema de la implementación sencilla de `eval` es que los comandos ejecutados heredan su entorno y directorio de trabajo del servidor de tuplas.

15.5.2 Comunicación con `eval`

A fin de evitar los problemas de la implementación sencilla de `eval`, el servidor puede bifurcar un hijo intermedio que redirija la entrada estándar, la salida estándar y el error estándar de modo que pasen por el espacio de tuplas. El hijo intermedio se denomina `Control` y el nieto que ejecuta el comando se denomina `Command`. La entrada estándar de `Command` proviene de un conjunto de tuplas de cadena de caracteres, y su salida estándar y su error estándar van a otros dos conjuntos de tuplas de cadena. Este enfoque tiene ciertas características en común con los *shells* y los procesos de segundo plano, por lo que tal vez resulte útil repasar el capítulo 7 antes de implementar esta fase.

La figura 15.4 ilustra la estrategia de hijo intermedio. `Control` llama a `in 348` para obtener entradas del espacio de tuplas y escribe estos datos en la entrada estándar de `Command`. De forma similar, `Control` lee datos de la salida estándar y del error estándar de `Command` e invoca a `out` para insertarlos en el espacio de tuplas con los ID de tupla `349` y `350`, respectivamente. El ID de tupla `348` es el identificador base `base_ID`.

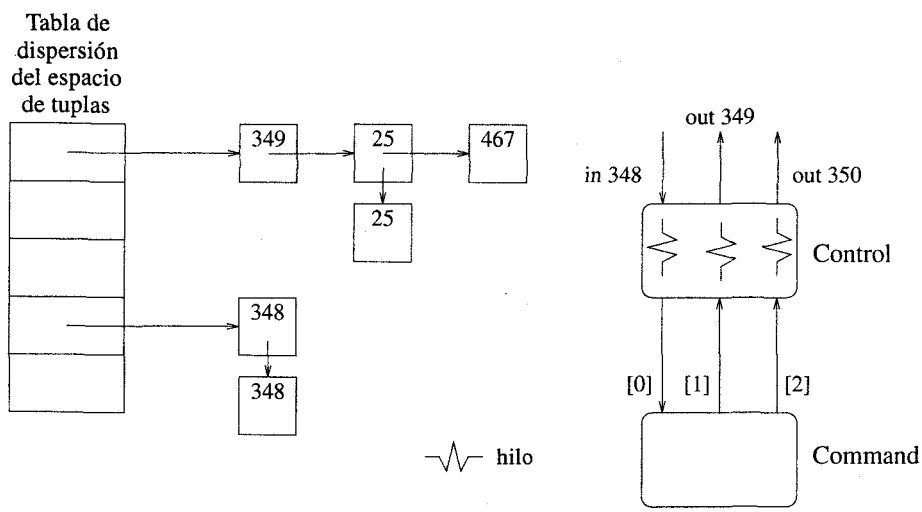


Figura 15.4: El `Control` proporciona la interfaz de `Command` con el espacio de tuplas; utiliza el ID de tupla `348` para la entrada estándar, el `349` para la salida estándar y el `350` para el error estándar.

Escriba una función `eval` que asigne tres ID de tupla consecutivos (`base_ID`, `base_ID + 1` y `base_ID + 2`) a la ejecución, bifurque un proceso `Control` y devuelva `base_ID` al invocador o –1 si la operación `fork` fracasa. En términos específicos, el algoritmo de la función `eval` es

- Asignar `largest_ID + 1` a `base_ID` e incrementar `largest_ID` en 3.
- Bifurcar un hijo para que ejecute el programa `Control`.
- Llamar a `waitpid` con la opción `WNOHANG` para eliminar los hijos previamente bifurcados.
- Devolver base_ID si `fork` tuvo éxito o –1 en caso contrario.

Al igual que en la primera implementación, el prototipo de `eval` es

```
int eval(struct tuple_t val);
```

y el tipo `val.tuple_val.tuple_type` debe ser `FUNCTION`.

El proceso `Control` crea tres tubos y bifurca un hijo (el nieto del servidor de espacio de tuplas) para ejecutar el programa `Command`. En términos específicos, el algoritmo de `Control` es

- Crear los tubos `pipe0`, `pipe1` y `pipe2`.
- Bifurcar un hijo que actúe como proceso `Command`.
- Cerrar `pipe0[0]`, `pipe1[1]` y `pipe2[1]`.
- Crear tres hilos, `handle_in`, `handle_out` y `handle_error`. Cada hilo crea una asa (*handle*) para el servidor de espacio de tuplas de RPC. Estas asas RPC son locales respecto de sus hilos.
  * El hilo `handle_in` se encarga de transferir información de entrada del espacio de tuplas a `Command`. En términos específicos, `handle_in` ejecuta una operación `in` con las tuplas que tienen como ID `base_ID` y escribe la información en `pipe0[1]`. Normalmente, la información se transfiere mediante tuplas de tipo `ARRAY`. Si `handle_in` encuentra una tupla `base_ID` de tipo `INTEGER`, supone que se encontró un fin de archivo, cierra `pipe0[1]`, destruye su asa de RPC y regresa. Puesto que Richard no tiene un `in` bloqueador, `handle_in` es difícil de implementar sin espera activa. Invente un mecanismo para hacerse atrás; es decir, si `in` falla, `handle_in` debe esperar un tiempo corto antes de volver a intentarlo.
  * El hilo `handle_out` se encarga de transferir la información de la salida estándar de `Command` al espacio de tuplas con el ID de tupla `base_ID + 1`. En términos específicos, `handle_out` lee información de `pipe1[0]` y ejecuta `out` con una tupla de tipo `ARRAY` que contiene los datos y tiene el ID `base_ID + 1`. Si `handle_out` se topa con un fin de archivo, cierra `pipe1[0]`, ejecuta `out`

con una tupla de tipo `INTEGER` y con ID `base_ID + 1`, destruye su asa de RPC y regresa.

* El hilo `handle_error` es similar a `handle_out`, excepto que transfiere información de `pipe2[0]` al espacio de tuplas con el ID de tupla `base_ID + 2`. Cuando `handle_error` se topa con un fin de archivo en `pipe2[0]`, cierra el descriptor `pipe2[0]` y ejecuta un `wait` hasta que el proceso `Command` termine. A continuación, `handle_error` ejecuta `out` con una tupla de tipo `INTEGER` y con ID `base_ID + 2`. El valor de esta tupla es la situación devuelta por `wait`. Por último, `handle_error` destruye la asa de RPC y regresa.

- El hilo principal se une a los tres hilos que creó y luego termina.

La estructura del hijo `Control` permite a cualquier proceso comunicarse con `Command` a través del espacio de tuplas. Es más, el servidor puede lanzar ejecutables ordinarios de UNIX como programas de `Command`. El proceso invocador obtiene la situación del hijo realizando una operación `in` con ID `base_ID + 2` hasta que encuentra una tupla de tipo `INTEGER`.

El proceso `Command` establece la comunicación antes de llamar a `execvp`. En términos específicos, el algoritmo de `Command` es

- Redirigir la entrada estándar a `pipe0[0]`.
- Redirigir la salida estándar a `pipe1[1]`.
- Redirigir el error estándar a `pipe2[1]`.
- Cerrar `pipe0[0]`, `pipe0[1]`, `pipe1[0]`, `pipe1[1]`, `pipe2[0]` y `pipe2[1]`.
- Invocar a `makeargv` de la figura 1.2 a fin de construir un arreglo de argumentos para el comando.
- Invocar a `execvp` para el comando.

Una vez que `eval` funcione, implemente el programa `ring` de la figura 4.1 utilizando espacios de tuplas y `eval` en lugar de entubamientos con `execvp`.

15.6 Espacios de tuplas como tuplas en Richard

En esta sección describiremos una extensión de Richard que permite la existencia de tuplas cuyo valor sea un espacio de tuplas. En la figura 15.5 se muestra un diagrama de los espacios de tuplas en un nodo dado. Cada nodo tiene un espacio de tuplas por omisión (predeterminadas o dadas por *default*) cuando se inicia Richard. El espacio de tuplas por omisión hace las veces de servidor de nombres para otros espacios de tuplas que residen en el nodo. El espacio de tuplas por omisión también puede contener tuplas ordinarias. La *lista no enlazada* contiene tuplas que

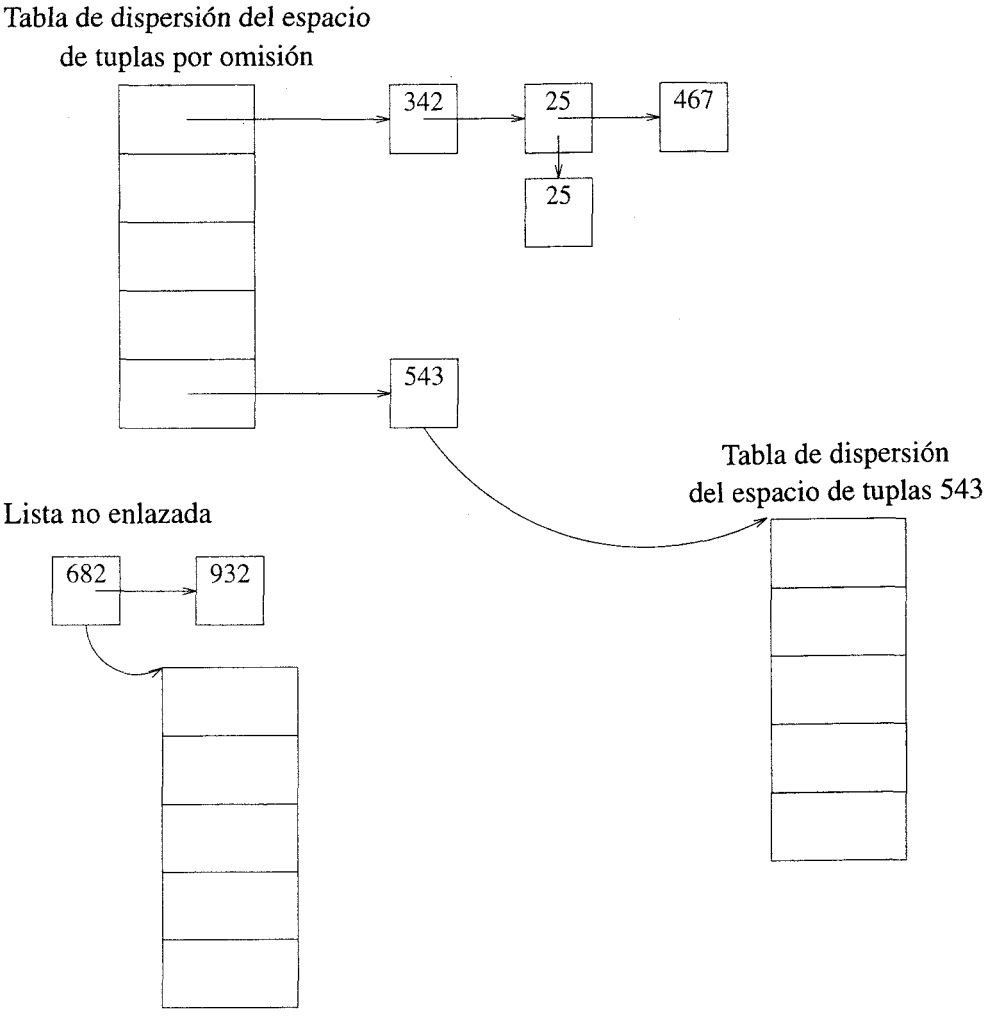


Figura 15.5: La tupla `543` tiene como valor un espacio de tuplas. La tupla contiene la dirección donde principia la tabla de dispersión que representa al espacio de tuplas.

no están en ningún espacio de tuplas. Un espacio de tuplas puede eliminarse o cambiarse de lugar sólo si está en la lista no enlazada.

Cada espacio de tuplas consiste en una tabla de dispersión. Se accede al espacio de tuplas presentando al servidor de tuplas de un nodo determinado la dirección del principio de la tabla de dispersión del espacio de tuplas. El servidor realiza operaciones con referencia a la tabla de dispersión especificada. Si una llamada no proporciona la dirección de una tabla de dispersión, el servidor utiliza la tabla por omisión.

El programa 15.7 contiene las declaraciones de tuplas para una implementación local de las tuplas con valores de espacio de tuplas. Escriba y pruebe todas las funciones localmente antes de convertirlas en llamadas remotas.

Programa 15.7: Declaraciones de tuplas extendidas para permitir valores de espacio de tuplas en una implementación local.

```
#define MAX_BUF 1024
#define MAX_STR 256

enum tuple_value_type {
    INTEGER = 0,
    ARRAY = 1,
    FUNCTION = 2,
    TUPLE_SPACE = 3
};

struct array_t {
    unsigned int array_len;
    char *array_val;
};

struct space_t {
    void *start_hash;
    int table_size;
};

struct tuple_val_t {
    enum tuple_value_type  tuple_type;
    union {
       int    ival;
       struct array_t array;
       char  *func;
       struct space_t space;
    } tuple_val_u;
};

struct tuple_t {
    unsigned int tuple_ID;
    unsigned int creator_key;
    unsigned int access_key;
    struct tuple_val_t tuple_val;
};
```

Programa 15.7

Ahora cada tupla tiene asociadas dos claves: `creator_key` (clave de creador) y `access_key` (clave de acceso). Para acceder a la información contenida en una tupla, una solicitud debe contener una `access_key` correcta. Una solicitud para eliminar la tupla o cambiar sus permisos o accesibilidad debe contener una `creator_key` correcta. Los valores de claves son `unsigned int`. Un valor de `access_key` de cero es una clave universal. Si una tupla tiene `access_key` de cero estará disponible para todas las solicitudes, excepto las de

eliminación o de cambio de permisos. Si una tupla tiene un valor de `access_key` positivo, toda solicitud deberá contener la clave de acceso correspondiente. El valor de la `creator_key` nunca puede ser cero. Esta clave confiere derechos de propiedad.

15.6.1 Operaciones con tuplas

Las operaciones de espacio de tuplas de Richard se complican por la existencia de múltiples espacios de tuplas. Los prototipos de las operaciones son

```
struct tuple_t out(struct tuple_cmd_t val);
struct tuple_t in(struct tuple_cmd_t val);
struct tuple_t rd(struct tuple_cmd_t val);
struct tuple_t eval(struct tuple_cmd_t val);
struct tuple_t mod(struct tuple_cmd_t val);
```

Cada función acepta un parámetro `struct tuple_cmd_t` en lugar de uno de tipo `struct tuple_t`. La operación `mod` realiza operaciones especiales como el cambio de permisos. La estructura `struct tuple_cmd_t` es

```
struct tuple_cmd_t {
   void *index_addr;
   unsigned int index_access_key;
   struct tuple_t tuple;
   enum mod_flag modify;
};
```

Cada nodo o servidor comienza con un espacio de tuplas por omisión que hace las veces de índice. El miembro `index_addr` especifica la dirección donde principia la tabla de dispersión que se utilizará para la operación. Si `index_addr` es `NULL`, la operación de tuplas se refiere al espacio de tuplas por omisión del servidor. Si `index_addr` no es `NULL`, apunta al espacio de tuplas que se usará para la operación. En este caso, `index_access_key` debe coincidir con la `access_key` del espacio de tuplas, pues de lo contrario la operación no podrá efectuarse.

Una operación `out` inserta una tupla nueva en el espacio de tuplas especificado. Si el tipo `val.tuple.tuple_val.tuple_type` es `TUPLE_SPACE`, el servidor creará un nuevo espacio de tuplas al que la nueva tupla apuntará. Cuando una operación `out` crea un espacio de tuplas nuevo, genera dos enteros sin signo al azar para las claves `creator_key` y `access_key`. La operación `out` devuelve estas claves, el identificador `tuple_ID` y la dirección donde principia la tabla de dispersión (`start_hash`) para que el invocador la utilice como `index_addr` en operaciones subsecuentes. Dicho de otro modo, `out` crea un espacio de tuplas al que sólo su creador puede acceder. El creador está en libertad de modificar los permisos u otorgar a otros procesos las claves de acceso o de creación.

Las dos operaciones `in` y `rd` devuelven una tupla. La operación `in` saca la tupla del espacio de tuplas, cosa que `rd` no hace. Cuando la tupla es del tipo `TUPLE_SPACE` (espacio de tuplas), la operación `in` no elimina la tabla de dispersión a la que la tupla apunta; en vez

de ello, saca la tupla del espacio de tuplas y la coloca en la lista no enlazada, `unlinked`. La operación `in` debe permitir el acceso atómico a una tupla y prohibir los demás accesos a esa tupla. Puesto que no es posible recuperar todo un espacio de tuplas con una operación `in`, lo que se recupera es un apuntador (la dirección donde principia la tabla de dispersión) y el espacio de tuplas se pasa del espacio de tuplas en el que residía a la lista no enlazada.

`mod` realiza operaciones de control con una tupla. Las posibles operaciones son

```
enum mod_flag{
   NOOP = 0,          /* no hacer nada */
   CHMOD = 1,         /* cambiar clave de acceso o de creador */
   DELETE = 2,        /* eliminar de la lista no enlazada */
   INSERT = 3,        /* insertar en un espacio de tuplas */
};
```

La operación `mod` fracasará si `val.tuple.creator_key` no es igual a la `creator_key` de la tupla. Las operaciones se refieren a la tupla cuyo ID es `val.tuple.tuple_ID`. Los valores de la bandera para la operación `mod` (`mod_flag`) son:

`CHMOD`:	Cambiar la `acces_key` de la tupla. La nueva clave de acceso será `val.tuple.tuple_val.access_key`.
`DELETE`:	Eliminar la tupla de la lista `unlinked`.
`INSERT`:	Pasar la tupla de la lista `unlinked` al espacio de tuplas `val.index_addr`. La clave de acceso del espacio de tuplas debe coincidir con `val.index_access_key`, pues de lo contrario la operación fracasará.

Pruebe la implementación exhaustivamente. Medite sobre cómo diseñar un sencillo sistema operativo distribuido basado en el modelo de tuplas. (Ya presentamos una buena cantidad de maquinaria aquí, así que esto no es tan imposible como suena.) Considere cuestiones como la asignación de nombres, la propiedad y la comunicación.

15.7 Un servidor multihilo para Richard

En esta sección consideraremos una extensión del sistema Richard básico a un sistema en el que el servidor utiliza hilos para manejar múltiples solicitudes simultáneas de espacio de tuplas. El servidor de espacio de tuplas es un candidato ideal para el uso de hilos con el fin de mejorar el rendimiento porque sus servicios operan sobre una estructura de datos grande compartida. El primer paso para construir un servidor multihilo es analizar la interacción entre los servicios. Implemente un servidor local con hilos para probar el análisis antes de implementar un servidor remoto.

Todos los servicios comparten y modifican la tabla de dispersión del espacio de tuplas, la cual debe estar protegida. Una estrategia consiste en tener un solo mutex para acceder a la tabla de dispersión. Esto hace que el acceso a la tabla de dispersión sea efectivamente secuencial y elimina gran parte del paralelismo que se logra al usar hilos. El otro extremo es utilizar un candado mutex para cada elemento del arreglo de la tabla de dispersión. Si este enfoque parece

razonable, lo más seguro es que la implementación anterior utilizara tablas de dispersión pequeñas. Una implementación más realista tiene tablas de dispersión con miles de entradas, y el costo de un candado mutex para cada entrada sería prohibitivo. Una estrategia intermedia utiliza candados mutex para cada sección de la tabla de dispersión.

Además del tamaño de la tabla de dispersión, considere cuántos hilos de ejecución simultáneos podrían estar accediendo a la tabla de dispersión. El programa `rpcgen` de Sun supone un máximo por omisión de 16. La idea de 16 hilos activándose y peleando por el mismo candado mutex no es muy atractiva. La implementación que desarrollaremos en esta sección utiliza una combinación de candados mutex de regiones y variables de condición.

Considere una implementación de espacio de tuplas en la que una tabla de dispersión de tamaño `HASH_SIZE` está asociada a un arreglo de tamaño `LOCK_SIZE` que contiene pares de candados mutex y variables de condición. El valor de `LOCK_SIZE` satisface $1 \leq$ `LOCK_SIZE` $\leq$ `HASH_SIZE`. Si `LOCK_SIZE` es 1, un solo par de candado mutex-variable de condición protegerá toda la tabla, mientras que si `LOCK_SIZE` es igual a `HASH_SIZE`, cada elemento de la tabla de dispersión tendrá su propio par de mutex y variable de condición. Si `LOCK_SIZE` tiene un valor intermedio, la tabla de dispersión tendrá candados de región.

Ejemplo 15.22

Suponga que `HASH_SIZE` *es* 1000 *y* `LOCK_SIZE` *es* 100. *Cada candado protege diez elementos de la tabla de dispersión. Utilice una correspondencia sencilla, como* `hash % 100`, *para determinar el número de candados a partir del índice de la tabla de dispersión.*

El programa 15.8 contiene las declaraciones de las tablas de dispersión. Cada elemento de la tabla contiene una bandera llamada `accessed` que es 1 si el elemento está siendo accesado y 0 en caso contrario. Defina un tipo, `struct hash_node`, para las entradas de la tabla de dispersión. Las cuentas de `struct hash_element` resultan útiles para búsquedas por tipo aleatorias. Escriba una función iniciadora de tabla de dispersión que aparte memoria para la tabla de dispersión y la inicialice como es debido. En la versión local, invoque a la función de iniciación al principio de la ejecución del programa principal. En la función remota, inserte una llamada a esta función en el talón de servidor a fin de iniciar la tabla de dispersión por omisión cuando el servidor comience su ejecución.

Programa 15.8: Declaraciones de tipos para una tabla de dispersión protegida.

```
struct hash_element {
    int number_ints;
    int number_arrays;
    int number_functions;
    int number_spaces;
    unsigned int accessed;
    struct hash_node *node_ptr;
};
```

```
struct lock_element {
    pthread_mutex_t lockv;
    pthread_cond_t condv;
};

typedef struct hash_table {
    struct hash_element hash[HASH_SIZE];
    struct lock_element lock[LOCK_SIZE];
} hash_t;
```

Programa 15.8

En un servidor con hilos, el ciclo de evento principal del talón de servidor vigila los descriptores de entrada para detectar solicitudes remotas. Cada vez que el talón de servidor lee una solicitud, crea un hilo nuevo para ejecutar el servicio correspondiente (`eval`, `in`, `out` o `rd`). Suponga que una variable `h` de tipo `hash_t` apunta a la tabla de dispersión. Para acceder a la entrada `hash_in` de la tabla de dispersión, cada hilo deberá hacer lo siguiente:

- Calcular el número de candado así:

```
lock_in = hash_in % LOCK_SIZE;
```

- Adquirir el mutex `h->lock[lock_in].lockv`.
- Si `h->hash[hash_in].accessed` es 0, asignarle 1 y liberar el candado

  `h->lock[lock_in].lockv`.

- En caso contrario, si `h->hash[hash_in].accessed` es 1, ejecutar una espera condicionada según la variable de condición `h->lock[lock_in].condv`
- Verificar otra vez la bandera `h->hash[hash_in].accessed` al despertar de la espera condicionada. Si sigue siendo 1, reanudar la espera condicionada; si es 0, asignarle 1 y liberar el mutex.

Una vez que un hilo obtiene acceso al elemento de la tabla de dispersión y realiza la operación necesaria, deberá liberar el elemento ejecutando lo siguiente:

- Adquirir el mutex `h->lock[lock_in].lockv`.
- Asignar 0 a `h->hash[hash_in].accessed`.
- Realizar una difusión de condición según `h->lock[lock_in].condv`.
- Liberar el mutex `h->lock[lock_in].lockv`.

Pruebe el servidor local con hilos escribiendo un programa principal que genere un número significativo de hilos para realizar operaciones simultáneas. Cuando el servidor sincronice correctamente los accesos simultáneos, consulte la sección 14.9 para recordar la forma de convertir el servidor local en remoto.

Después de la conversión a servidor remoto, realice pruebas del tiempo de ejecución del algoritmo de búsqueda cooperativa de la sección 15.4 con múltiples agentes. Compare los tiempos de ejecución cuando el servidor utiliza hilos y cuando no lo hace.

15.8 Lecturas adicionales

La reseña "How to write parallel programs: a guide to the perplexed" explica el enfoque general de espacio de tuplas para la computación en paralelo [16]. El artículo "Generative communication in Linda" describe el desarrollo original de la estructura de Linda [32]. La semántica simplificada de Richard tiene cierto parecido con Kernel Linda, que es la base del sistema operativo QIX [46]. Otros artículos de interés son [17] y [33]. El artículo "The Linda alternative to message-passing systems" compara el rendimiento de Linda y con el PVM [18].

Apéndice A

Fundamentos de UNIX

Unix está en vías de contar con una norma común, aunque todavía existen variaciones de un desarrollador a otro, en asuntos como formatos para documentación en línea, opciones para compilación de programas y localización de las bibliotecas de sistemas. Desafortunadamente, estos son los servicios que un programador de sistemas necesita desde el primer día. Sería imposible contemplar en un solo apéndice todo lo que necesita un usuario para convertirse en un programador de sistemas UNIX. Este apéndice se enfoca en cómo accesar la documentación UNIX, y en cómo compilar y ejecutar programas escritos en lenguaje C. Se proporciona la información mínima requerida para lograrlo (pues se supone que el lector está familiarizado con ello). También se ofrecen líneas de dirección para deducir datos determinados sobre cierto sistema.

A.1 Cómo obtener ayuda

La mayoría de los sistemas UNIX contienen documentación en línea llamada *páginas del manual (man pages).* Aquí "man (por sus siglas en inglés)" tiene un significado similar al de "manual", como en manual del sistema. Los usuarios que trabajan con UNIX consideran de utilidad el uso de estas páginas del manual. No obstante, las páginas del manual contienen pocos ejemplos y son difíciles de entender, a menos que el usuario ya esté familiarizado con el material.

Tradicionalmente las páginas del manual están divididas en secciones al igual que la tabla A.1. Cada sección cuenta con una introducción que resume puntos importantes relacionados con las convenciones usadas en esta sección. Si no está familiarizado con las páginas del manual, lea las introducciones a las tres primeras secciones. Las páginas del manual se refieren a puntos cuyos números de sección aparecen entre paréntesis (por ejemplo, `intro(1)` se refiere a la introducción a la sección uno).

Sección	Contenidos
1	Comandos del usuario
2	Llamadas del sistema
3	Funciones de biblioteca de lenguaje C
4	Dispositivos e interfaces de redes
5	Formatos de archivos
6	Juegos y demostraciones
7	Entornos, tablas y macros `troff`
8	Mantenimiento del sistema

Tabla A.1: Contenidos típicos de una tabla UNIX de páginas del manual.

La figura A.1 muestra la salida de la utilidad `man` cuando el comando `man whatis` es ejecutado en una estación de trabajo Sun, que está operando bajo un sistema operativo Solaris 2.3. (Solaris es el nombre que Sun Microsystems le ha dado a la última versión del sistema operativo UNIX.) La primera línea de la página del manual es la línea de encabezado que contiene el nombre del comando seguido por el número de sección de la hoja del manual. El `whatis(1)` en la figura A.1 se refiere al comando `whatis` descrito en la sección uno de las páginas del manual. No trate de ejecutar `whatis(1)`. El sufijo `(1)` no es parte del nombre del comando, sino el indicador de la sección de una página del manual.

Cada página del manual cubre algún aspecto de UNIX (por ejemplo, un comando, una utilidad, una llamada del sistema). Las páginas del manual individuales están organizadas en secciones, en forma similar a como se presenta el comando `whatis` en la figura A.1. Algunos de los títulos más comunes incluyen:

`HEADER:`	Título individual de una página del manual.
`NAME:`	Resumen de una línea.
`SYNOPSIS:`	Describe el uso.
`AVAILABILITY:`	Indica la disponibilidad sobre el sistema.
`DESCRIPTION:`	Describe lo que hace el comando o la función.
`RETURN VALUES:`	Los valores de regreso si son aplicables.
`ERRORS:`	Resume los valores de `errno` y las condiciones de errores.
`FILES:`	Enista los archivos de sistema que usan los comandos o funciones.
`SEE ALSO:`	Lista de otros comandos relevantes y secciones adicionales del manual.
`ENVIRONMENT:`	Lista de variables relevantes en el entorno.
`NOTES:`	Provee información de herramientas poco comunes en su uso e implementación.
`BUGS:`	Enlista bugs conocidos y otras advertencias (¡esta sección siempre es de gran tamaño!).

```
whatis(1)                  User Commands                  whatis(1)

NAME
     whatis - display a one-line summary about a keyword

SYNOPSIS
     whatis command...

AVAILABILITY
     SUNWdoc

DESCRIPTION
     whatis looks up a given command and displays the header line
     from the manual section. You can then run the man(1) com-
     mand to get more information. If the line starts
     'name(section) ...' you can do 'man -s section name' to get
     the documentation for it. Try 'whatis ed' and then you
     should do 'man -s 1 ed' to get the manual page for ed(1).

     whatis is actually just the -f option to the man(1) command.

     whatis uses the /usr/share/man/windex database. This data-
     base is created by catman(1M). If this database does not
     exist, whatis will fail.


FILES
     /usr/share/man/windex    table of contents and keyword data-
                              base

SEE ALSO
     apropos(1), man(1), catman(1M)
```

Figura A.1: Ejemplo de un página del manual del comando `whatis` del paquete Sun Solaris (reimpreso con el permiso de SunSoft, Inc. Derechos reservados por SunSoft, Inc.).

Ejercicio A.1

El siguiente comando proporciona ayuda en línea utilizando el comando `man`.

```
man man
```

Ejecute este comando y trate de entender su descripción. (¡Buena suerte!) Es probable que aparezca una línea en la parte inferior izquierda de la pantalla que se verá así:

```
--More--(41%)
```

El porcentaje dentro del paréntesis indica la cantidad de información `man` que ya ha sido mostrada. Para ir hacia la siguiente pantalla, oprima la barra de espacio. Para avanzar sólo una línea, oprima la tecla de retorno.

Ejemplo A.1

El siguiente comando despliega información introductoria acerca de la sección uno de las páginas del manual (por ejemplo, `intro(1)`*).*

```
man intro
```

Cuando `man` es llamado a través de la opción `-k`, este comando resume toda la información de una página del manual que tiene un nombre específico. La opción `-k` puede usarse si se dispone del archivo *summary-database*. Esta base de datos es también usada por el comando `whatis`. La página del manual de `whatis` mostrada en la figura A.1 indica que el archivo summary-database para Sun Solaris es `/usr/share/man/windex`.

Ejemplo A.2

El comando `man -k intro` *puede producir la siguiente salida relacionada con las tres primeras secciones de las páginas del manual.*

```
Intro Intro (1) - Introducción a comandos y
                  programas de aplicaciones
Intro Intro (1m)- Introducción a comandos de mantenimiento y
                  programas de aplicaciones
Intro Intro (2) - Introducción a llamadas del sistema y
                  a números de error
Intro Intro (3) - Introducción a funciones y bibliotecas
```

Es posible que para un comando específico puedan existir varias piezas de información en una página del manual, y `man` sólo despliega la primera pieza de información, a menos que específicamente se requiera para una sección posterior o para todas las piezas de información relacionadas con el comando en cuestión. Sun Solaris usa la opción `-a` para indicar todas las piezas de información de páginas del manual relacionadas con un comando, y la opción `-s` para indicar una sección en particular de las páginas del manual. Otros sistemas utilizan el número de sección especificado después de `man`. En este apéndice se utilizan las especificaciones de Sun Solaris.

Ejemplo A.3

Sun Solaris usa el comando `man -s 2 intro` *para mostrar la sección dos de* `intro` *contenida en las páginas del manual, mientras que algunos otros sistemas utilizan el comando* `man 2 intro`.

El comando `man write` habilita una página del manual para ejecutar el comando `write (1)`. El comando `write` en la sección uno de las páginas del manual es un comando ejecutado desde la línea de comando. Este `write` no es el más consultado por los programadores en las páginas del manual. Más bien, necesitan información en la llamada de sistema `write` usada en los programas hechos en lenguaje C.

Ejemplo A.4

El siguiente comando despliega todas las páginas del manual relacionadas con `write`.

```
man -a write
```

A.1.1 Llamadas del sistema y funciones de biblioteca en lenguaje C

La sección NAME de una página del manual enlista los nombres de puntos descritos en esta página. Las páginas del manual contienen información de una gran cantidad de puntos. La página del manual sobre `write(1)` mencionada en los ejemplos anteriores contiene información sobre un comando, y `write(2)` describe la llamada del sistema. Los dos `write` tienen propósitos completamente diferentes. Vea la sección de SYNOPSIS para determinar cuáles son cada uno de los propósitos. La sección de SYNOPSIS presenta un resumen de cómo un comando o función son llamados. La sección de SYNOPSIS para llamadas del sistema y funciones de biblioteca en lenguaje C contiene funciones prototipo junto con los encabezados de archivo requeridos. El `write(2)` es un llamado del sistema descrito en la sección dos de las páginas del manual y requiere un encabezado de archivo. Esta función es llamada desde un programa hecho en C y contrasta con `write(1)`, que es ejecutada desde el panel de comandos o un intérprete de comandos.

Ejemplo A.5

Las siguientes líneas muestran una sinopsis de una página del manual para `write(2)`.

```
SINOPSIS
   #include <unistd.h>
   ssize_t write(int fildes, const void *buf, size_t nbyte);
```

La función `write (2)` toma tres parámetros y regresa un valor del tipo `ssize-t`. La sinopsis especifica que para que los programas funcionen correctamente deben incluir el encabezado de archivo `unistd.h`. La página del manual especifica que `write` traslada `nbyte` número de bytes de `buf` a un archivo que es especificado por `fildes`.

La E/S a nivel de sistema UNIX utiliza descriptores de archivos en lugar de apuntadores de una biblioteca normal de C. En la sección 3.3 se analiza la representación de archivo, aunque los dos siguientes ejemplos dan una idea de ello.

Ejemplo A.6

El siguiente segmento de código escribe en pantalla el siguiente mensaje: "hello world".

```
#include <unistd.h>
#include <string.h>
#define MYMSEG "hello world\n"
int bytes;

bytes = write(STDOUT_FILENO, MYMESG, strlen(MYMESG));
```

STDOUT_FILENO designa al descriptor de archivo para la salida estándar. La entrada estándar tiene un descriptor de archivo STDIN_FILENO, y un error estándar tiene el descriptor de archivo STDERR_FILENO. Por lo general, el valor numérico de los tres descriptores de archivos mencionados antes es 0,1 y 2, respectivamente; asegúrese de usarlas ahora que estas constantes han sido normalizadas.

Ejemplo A.7

Compare el ejemplo 6 con la siguiente y más común instrucción de `fprintf`.

```
#include <stdio.h>
fprintf(stdout, "hello world\n");
```

El `fprintf` utiliza apuntadores de archivo en lugar de descriptores de archivos. El `fprintf` es parte de una biblioteca normalizada E/S de un ISO C, mientras que `write` es una llamada del sistema. El `write` puede tener una salida arbitraria sin modificación (incluyendo datos binarios). El `fprintf` provee un formato que produce una salida cuyo formato puede ser leído por humanos.

Ejercicio A.2

¿Qué pasa si los archivos de encabezado del ejemplo A.6 no están en el programa?

Respuesta:

El identificador `STDOUT_FILENO` quedaría sin identificación y el compilador generaría un mensaje de error.

Una interpretación errónea del prototipo de la función bastante común está relacionada con la declaración de apuntadores. Cuando una designación de apuntador aparece en la función prototipo, el programa debería definir una variable del mismo tipo de la que está apuntando y pasar un apuntador a esa variable, utilizándolo como el parámetro.

Ejemplo A.8

Es probable que el siguiente segmento de código produzca un error de segmentación, puesto que `mybuf` *es un apuntador no iniciado.*

```
#include <unistd.h>

char *mybuf;
size_t nbytes;
write(STDOUT_FILENO, mybuf, nbytes);
```

Ejemplo A.9

En el siguiente segmento de código, `mybuf` *es la dirección de un almacenamiento temporal (*buffer*). (El programa ha iniciado* `mybuf` *y* `nbytes` *antes de haber llamado a* `write`*; en caso contrario, la instrucción produciría una salida llena de basura.)*

```
#include <unistd.h>

char mybuf[100];
size_t nbytes;
write(STDOUT_FILENO, mybuf, nbytes);
```

La función `write` regresa un valor del tipo `ssize_t`. Siempre capture este valor y verifique que no haya errores. La sección de `RETURN VALUES` de las páginas del manual para `write(1)` especifica que en caso de error, `write` regresa a `-1` e inicializa `errno` a uno de los

valores contenidos en la sección ERRORS de las páginas del manual. (Vea la sección 1.5 para mayor información.)

A.1.2 Comandos y utilidades de UNIX

Los comandos y utilidades de UNIX son programas internos que no requieren compilación. El usuario los puede ejecutar directamente de la línea de comando. El comando `write(1)` es utilizado para mandar información de un usuario a otro. El siguiente texto utiliza el siguiente formato para mostrar una sinopsis.

SINOPSIS

```
write user [line]
```

POSIX.2, Spec 1170

El comando `write(1)` cuenta con el operando necesario `user` y un operando opcional `line`. Puesto que la entrada es en la sección uno de las páginas del manual, en su sinopsis no tiene archivos de encabezado y el comando `write(1)` es ejecutado desde el intérprete de programas. La hilera en la parte inferior izquierda del cuadro nos indica a qué tipo de norma se está acogiendo la sinopsis. La utilidad `man` no enlista las normas en esta sinopsis.

Ejercicio A.3

La siguiente descripción de comando está dada en una página `man` para `write(1)`.

```
write copies lines from the terminal to that of another user.
```

Para mandar información a un usuario llamado `annie`, escriba el siguiente comando:

```
write annie
```

Desde este punto hasta encontrar un fin de archivo (por ejemplo, `ctrl-d`), lo que sea escrito en la terminal también aparecerá en la pantalla de `annie`.

La descripción de `write(1)` también indica que el argumento opcional `line` designa a una terminal en particular para usuarios que se han registrado más de una vez. Desafortunadamente, las páginas del manual no dan ninguna indicación de cómo encontrar qué usuarios están registrados, qué terminal está usando cada uno o cómo se envía un fin de archivo.

Ejercicio A.4

Ejecute el comando `who -H` para ver quién está trabajando y en qué terminal se encuentra (el comando `who` forma parte del Spec 1170, pero no de POSIX.2.)

Ejercicio A.5

Normalmente un `ctrl-d` tecleado en la terminal indica un fin de archivo. (Mantenga presionada la tecla de `ctrl` y presione la tecla `d`. Utilice un `ssty -a` para verificar el indicador de fin de archivo en la terminal. El comando `stty` probablemente utiliza `^d` para designar a `ctrl-d`.)

Por convención, los comandos de UNIX pueden tener tres clases de argumentos de línea de comandos: opciones, opción-argumentos y operandos. Las opciones constan de guiones y letras o dígitos individuales. Algunas opciones son seguidas por opción-argumentos. Los operandos son argumentos que van seguidos de opciones u opción-argumentos. Los corchetes ([ ]) son usados para encerrar puntos opcionales, mientras que los argumentos necesarios no requieren este tipo de paréntesis. POSIX.2 (una norma IEEE para intérpretes de comandos y utilidades UNIX) provee las siguientes directrices para especificar argumentos de línea de comandos:

- Una opción consta de un solo carácter alfanumérico.
- Todas las opciones van precedidas por "-".
- Opciones que no tienen argumentos pueden ser agrupadas después de un solo "-".
- El primer argumento de una opción debe ser precedido de un carácter de tabulador o de espacio. (Algunas utilidades tradicionales no respetan este lineamiento.)
- Los argumentos de las opciones no pueden ser opcionales.
- Los grupos de argumentos de opciones que sigan a una opción deben estar separados ya sea por comas o por caracteres de tabulador o de espacio, y deben ir entre comillas (`-o xxx, z, yy u -o "xxx z yy"`).
- Todas las opciones deben preceder a los operandos en la línea de comando.
- "--" puede ser usado para indicar que se ha llegado al final de las opciones.
- El orden relativo de los operandos puede afectar su significado en la forma determinada por el comando con el que aparecen.
- "-" precedido y seguido de un carácter de espacio sólo debe ser usado para indicar la entrada estándar.

Utilice la función `getopt` para analizar opciones en utilidades escritas por usuarios.

Ejemplo A.10

El comando `ls` *enlista los archivos. La utilidad* `man` *de Sun Solaris 2 da la siguiente sinopsis para* `ls`.

```
SINOPSIS
     ls [ -abcCdfFgilLmnopqrsRstux1 ] [ names ]
```

Cada letra en el primer `[ ]` del ejemplo A.10 representa una opción descrita en la sección de `DESCRIPTION` o en la de `OPTIONS` de la página del manual. Aquí `[names]` es un operando opcional.

A.1.3 Comandos relacionados con man

El comando `apropos x` despliega los números y nombres de la sección de las páginas del manual cuyas líneas de NAME contengan a x.

Ejemplo A.11

El siguiente comando enlista todas las piezas de información de las páginas del manual que contienen la frase `wait` *en la sección de* NAME.

```
apropos wait
```

El comando `whatis x` proporciona una sinopsis de una línea del comando x. El comando `which x` da el nombre de ruta completo del archivo llamado x, que sería ejecutado si el usuario tecleara x como un comando. La utilidad `which` usa a la variable de entorno PATH del intérprete de comandos y otros alias asignados por el usuario o por el administrador del sistema. En el capítulo 3 se discute la variable de entorno PATH.

Ejemplo A.12

El comando `which kill` *puede producir la respuesta* `/bin/kill`, *lo cual indica que el comando* `kill` *del directorio* `/bin` *es ejecutado cuando un usuario teclea un comando como* `kill %1`.

Los comandos `which`, `whatis` y `apropos` no son parte de POSIX o de las normas Spec 1170, pero la mayoría de los sistemas los tienen. El comando `find` da muchas opciones para localizar archivos con diferentes atributos.

SINOPSIS

```
find path... [operand-expression...]
```

POSIX.2, Spec 1170

Ejemplo A.13

El siguiente comando enlista todos los archivos en el árbol del directorio de trabajo cuyo nombre termina en `.c`.

```
find . -name "*.c" -print
```

El primer argumento de `find` en el ejemplo A.13 especifica el árbol del directorio en donde hay que empezar la búsqueda. El parámetro `-name` especifica el patrón que se debe encontrar.

A.2 Compilación

El compilador de C, `cc`, traduce programas fuente en C a módulos objeto o a módulos ejecutables. Un *módulo ejecutable* está listo para ser cargado y ejecutado. El proceso de

compilación se ejecuta por etapas. En la primera etapa, un preprocesador expande a macro-instrucciones e incluye a los archivos de encabezado. El compilador ejecuta en seguida varios pasos de compilación a través del código para traducir el código al lenguaje ensamblador de la máquina de destino primero y luego al código de máquina. El resultado es un *módulo objeto* que consiste en código de máquina y tablas de referencias no resueltas. La etapa final de compilación une una colección de módulos objeto para formar un módulo ejecutable en donde todas las referencias han sido resueltas. El ejecutable contiene exactamente una función `main`.

Ejemplo A.14

El siguiente comando compila a `mine.c` y produce un ejecutable `mine`.

```
cc -o mine mine.c
```

Si la opción `-o mine` del ejemplo A.14 se omitiera, el compilador C produciría un ejecutable llamado `a.out`. Utilice la opción `-o` si no desea obtener un nombre por omisión (predeterminado o por *default*) que no significa nada.

Ejemplo A.15

El siguiente archivo fuente `mine.c` *contiene una referencia no definida a la función* `serr`.

```
void serr(char *msg);

void main(int argc, char *argv[])
{
    serr("This program does not do much\n");
}
```

Cuando el programa `mine.c` del ejemplo A.15 es compilado como se hace en el ejemplo A.14, el compilador C muestra un mensaje que indica que `serr` representa una referencia no resuelta y no produce ningún ejecutable.

La mayoría de los programas no están contenidos en un solo archivo fuente, por lo que se requiere se unan múltiples archivos fuentes. Todos los archivos fuente que van a ser compilados pueden ser especificados en un solo comando `cc`. Asimismo, el usuario puede compilar la fuente en varios módulos objeto separados y unirlos en un paso por separado, para formar un módulo ejecutable.

Ejemplo A.16

Suponga que la función `serr` *se halla contenida en el archivo fuente* `minelib.c`. *El siguiente comando compila el archivo fuente* `mine.c` *del ejemplo A.15 con el archivo fuente* `minelib.c` *para producir un módulo ejecutable llamado* `mine`.

```
cc -o mine mine.c minelib.c
```

La opción `-c` de `cc` ocasiona que el compilador C produzca un módulo objeto en lugar de un ejecutable. Un módulo objeto no puede ser cargado en memoria o ejecutado hasta que sea unido a bibliotecas y otros módulos que resuelvan referencias. Una variable mal deletreada o una función de biblioteca perdida puede no ser detectada hasta que ese módulo objeto sea unido a un ejecutable.

Ejemplo A.17

El siguiente comando produce un módulo objeto `mine.o`.

```
cc -c mine.c
```

Cuando la opción `-c` es usada, el compilador C produce un módulo objeto con un nombre cuya extensión es `.o`. El `mine.o` producido por el comando `cc` del ejemplo A.17 puede ser unido más tarde con otro archivo objeto (por ejemplo, `minelib.o`) para producir un ejecutable.

Ejemplo A.18

El siguiente ejemplo une los módulos objeto `mine.o` *con* `minelib.o` *para producir el ejecutable* `mine`.

```
cc -o mine mine.o minelib.o
```

A.3 Archivos makefile

La utilidad `make` permite a los usuarios recompilar en forma incremental una colección de programas módulos objeto. Aun para compilar un solo programa es conveniente usar `make`, pues ayuda a evitar errores.

Para usar `make` el usuario debe especificar, en un *archivo de descripción,* qué dependencias o relaciones existen entre los módulos. Los nombres por omisión para los archivos de descripción son `makefile` y `Makefile`. Cuando el usuario escribe `make`, esta utilidad busca los archivos `makefile` o `Makefile` en el directorio actual y verifica este archivo de descripción para ver si algo necesita ser actualizado.

Los archivos de descripción, como su nombre lo indica, describen las relaciones de dependencia que existen entre varios programas módulos. Las líneas que comienzan con el carácter # son comentarios. Las dependencias en los archivos de descripción presentan las siguientes formas:

```
target:          components
TAB              rule
```

La primera línea es llamada *dependencia* y la segunda línea *regla.* El primer carácter en una línea de regla en un archivo de descripción debe ser el carácter TAB (tabulación). Una dependencia puede ir seguida de una o más líneas de regla.

Ejemplo A.19

En el ejemplo A.18 el ejecutable `mine` *depende de los archivos objeto* `mine.o` *y* `minelib.o`. *El siguiente segmento describe la relación de dependencia.*

```
mine:  mine.o minelib.o
       cc -o mine mine.o minelib.o
```

La descripción en el ejemplo A.19 dice que `mine` depende de `mine.o` y `minelib.o`. Si cualquiera de estos dos últimos archivos ha sido modificado desde la última vez que se cambió a `mine`, entonces `mine` debe ser actualizado ejecutando un `cc -o mine mine.o minelib.o`. Si la descripción se encuentra en un archivo, éste es llamado `makefile` y sólo es necesario escribir la palabra `make` para ejecutar todas las actualizaciones necesarias. (Nota: Las líneas de descripción deben empezar con un carácter TAB [tabulación], por lo que existe un cáracter invisible TAB antes que el `cc`.)

En el ejemplo A.19, el nombre de archivo `mine`, anterior a los dos puntos, es llamado el *objetivo.* El objetivo depende de sus *componentes* (por ejemplo, `mine.o` o `minelib.o`). La línea contigua a la especificación de la dependencia es el *comando* o *regla* para actualizar el objetivo, si éste es más antiguo que cualquiera de sus componentes. Así, si `mine.o` o `minelib.o` cambian, ejecute un `make` para actualizar `mine`.

Ejemplo A.20

En situaciones más complicadas, un objetivo específico puede depender de componentes que a su vez son objetivos. El siguiente archivo de descripción `makefile` *tiene tres objetivos.*

```
my:     my.o mylib.o
        cc -o my my.o mylib.o

my.o:   my.c myinc.h
        cc -c my.c

mylib.o:  mylib.c myinc.h
        cc -c mylib.c
```

Sólo teclee `make` *para realizar las actualizaciones requeridas.*

En algunas ocasiones es útil visualizar las dependencias en el archivo de descripción por medio del uso de una gráfica dirigida. Utilice nodo (sin duplicados) para representar los objetivos y sus componentes. Dibuje una flecha dirigida del nodo A al nodo B, si el blanco A depende del B. Una gráfica de las dependencias correctamente preparada no debe contener ciclos. La figura A.2 muestra una gráfica de dependencias para el archivo de descripción del ejemplo A.20.

Un archivo de descripción también puede tener *definiciones de macros* de la forma:

```
NAME = value
```

Siempre que aparezca `$(NAME)` en el archivo de descripción, value será sustituido antes de que la frase sea procesada. No utilice caracteres de tabulación en macros.

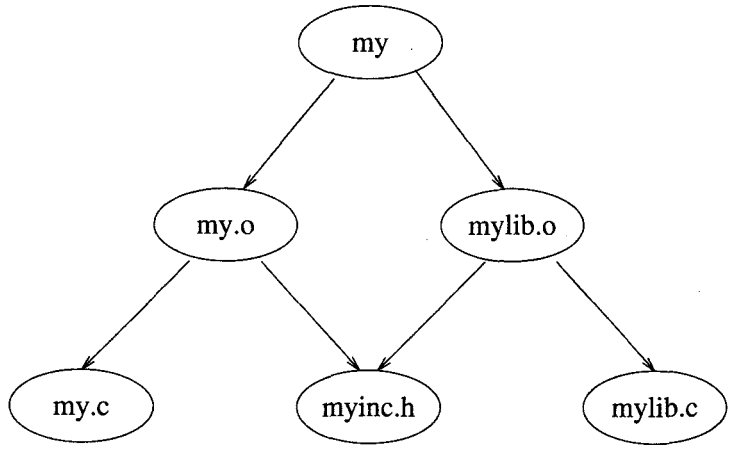


Figura A.2: Una gráfica de dependencias para `makefile` del ejemplo A.20.

Ejemplo A.21

Los siguientes archivos de descripción utilizan una macro para representar las opciones de compilación para que las opciones del compilador sean cambiadas en un solo lugar, en lugar de hacerlo a través de todo el archivo.

```
OPTS = -O -H

my:     my.c my.h
        cc $(OPTS) -o my my.c
```

El comando `make` también permite que se especifique el nombre de un objetivo en la línea de comando. En este caso, `make` actualiza solamente el objetivo especificado. Cuando se estén desarrollando múltiples objetivos en el mismo directorio (por ejemplo, emisores y receptores de programas), utilice esta herramienta para depurar un objetivo a la vez. Si en la línea de comando no hay objetivos explícitamente especificados, `make` verifica sólo el primer objetivo en el archivo de descripción. Frecuentemente, el primer objetivo de un archivo de descripción recibe el nombre de `all`, que depende de todos los otros objetivos. Utilice la opción `-f` con `make` para archivos de descripción con nombres que no sean `makefile` o `Makefile`.

Ejemplo A.22

El siguiente comando actualiza el objetivo `target1` *del archivo de descripción* `mymake`.

```
make -f mymake target1
```

A.4 Archivos de encabezado

El preprocesador C copia los archivos de encabezado especificados en los comandos `#include`, en archivos fuente, antes de iniciar la compilación. Por convención, los archivos de encabezado siempre terminan con una extensión `.h`. Coloque declaraciones de constantes, tipos y funciones dentro de los archivos de encabezado. *No coloque declaraciones de variables en los archivos de encabezado porque esto puede dar como resultado la multidefinición de variables.* El siguiente ejercicio muestra las dificultades causadas por declaraciones de variables colocadas en archivos de encabezado.

Ejercicio A.6

¿Qué valores arrojaría el programa si los archivos `myinc.h`, `my.c` y `mylib.c` contienen los segmentos especificados a continuación? El archivo `myinc.h` del ejemplo A.20 contiene el siguiente segmento.

```
#include <stdio.h>
static int num;
void changenum(void);
```

El archivo `my.c` contiene el siguiente programa `main`.

```
#include "myinc.h"
void main (void)
{
    num = 10;
    changenum();
    printf("num es %d\n", num);
    exit(0);
}
```

El archivo `mylib.c` contiene la siguiente función.

```
#include "myinc.h"
void changenum(void)
{
    num = 20;
}
```

Respuesta:

Ambos archivos, `my.c` y `mylib.c`, contienen una variable `num` puesto que su definición aparece en el archivo `myinc.h`. El llamado de `changenum` por el programa `main` no afecta el valor de la variable `num` definida en el archivo `my.c`. El programa despliega el valor `10` en lugar del valor `20`.

Ponga entre comillas los nombres de archivos de encabezado personales de la siguiente manera:

```
#include "myinc.h"
```

Las comillas indican al compilador que busque al archivo de encabezado en el directorio que contiene el archivo fuente antes de buscar en el lugar estándar. Ponga entre picoparéntesis los

archivos de encabezado definidos por el sistema (como en `#include <stdio.h>`), pues así el compilador buscará directamente en el lugar estándar. Lo que se considere el lugar estándar es dependiente de la implementación, pero la página del manual par `cc` usualmente describe cómo ocurre una búsqueda. El directorio `/usr/include` contiene muchos de los archivos de encabezado estándar. Los archivos en este directorio frecuentemente contienen otros archivos `.h` en subdirectorios de `/usr/include`. El directorio `/usr/include/sys` es un lugar estándar para muchos de los archivos `.h` requeridos en este libro. Cuando vaya a utilizar llamadas del sistema o funciones de biblioteca, asegúrese de que incluya los archivos de encabezado especificados en la `SYNOPSIS` de la página del manual.

Ejercicio A.7

Un programa usa el símbolo de error `EAGAIN` conjuntamente con un llamado a `write`. El compilador se queja de que `EAGAIN` no está definida. ¿Qué hay que hacer?

Respuesta:

Trate los siguientes pasos para resolver el problema:

- Asegúrese de que incluya todos los archivos de encabezado mencionados en la sinopsis de `write`. La página del manual especifica el archivo de encabezado `<unistd.h>`.
- En algún lugar de las páginas del manual se menciona que `errno.h` debe ser incluido en programas donde se refieran símbolos de error. Si el programa incluye el archivo `errno.h`, el problema está resuelto.
- Si no se encuentra en la página del manual ninguna mención sobre el archivo `errno.h`, busque el símbolo `EAGAIN` en los archivos de encabezado del sistema usando:

  ```
  cd /usr/include
  grep EAGAIN *
  ```

  El `grep` busca la cadena de caracteres `EAGAIN` en todos los archivos del directorio `/usr/include`. Desgraciadamente, el símbolo `EAGAIN` no está en ninguno de los archivos del directorio `/usr/include`.
- Cambie al directorio `/usr/include/sys` e inténtelo con un `grep` otra vez. La siguiente es una respuesta típica al `grep`.

  ```
  errno.h:#define EAGAIN 11
  errno.h:#define EWOULDBLOCK         EAGAIN
  ```

  Puede ser tentador eliminar el problema incluyendo al archivo `sys/errno.h` en el código fuente, pero lo que el compilador realmente quiere es el archivo `errno.h`. Es mejor utilizar el archivo `errno.h` directamente, ya que éste incluye al archivo `sys/errno.h`, y también contiene definiciones adicionales.

A.5 Enlace y bibliotecas

Que el programa tenga los archivos de encabezado correctos no quiere decir que se hayan terminado los problemas. Un archivo de encabezado da definiciones de símbolos y prototipos

de funciones, pero no suministra el código para las funciones de biblioteca o las llamadas del sistema.

Ejercicio A.8

El archivo fuente `mylog.c` calcula el valor de un logaritmo. Después de incluir el archivo `math.h` en ese archivo fuente, el usuario compila el programa y recibe un mensaje de error que dice que la función `log` no pudo ser encontrada. ¿Por qué?

Respuesta:

El archivo de encabezado `math.h` sólo le indica al compilador C cuál es la forma (prototipo) de la función `log`. En realidad no lo proporciona la función.

La compilación se lleva a cabo en dos fases diferentes. En la primera el compilador C traduce los archivos fuente en C a código objeto. La opción `cc -c` se detiene en este punto. El código objeto no está listo para ser ejecutado porque referencias del programa a elementos definidos fuera de este módulo no han sido todavía resueltos. Para que un módulo ejecutable pueda ser producido, todos los símbolos no definidos (*referencias externas no resueltas)* deben ser encontrados. El compilador `cc` llama al editor de ligado (link editor) `ld` para poder realizar esta tarea.

Ejemplo A.23

El siguiente comando compila el archivo fuente `mylog.c` *en conjunto con la biblioteca matemática del sistema, para producir un ejecutable llamado* `mylog`.

```
cc -o mylog mylog.c -lm
```

Para poder usar las funciones matemáticas de la biblioteca de C, coloque `#include <math.h>` en el archivo fuente y también especifique que el programa debe ser ligado con la biblioteca matemática `-lm`, cuando éste sea compilado.

Los nombres de las bibliotecas son especificados con el uso de la opción `-l`. Los archivos fuente son procesados en el orden en que aparecen en la línea de comando `cc`, por lo que el lugar que ocupa `cc` en la línea `-l` resulta significativo; debe ser posterior al de los archivos objeto porque sólo aquellas entradas iguales a referencias no resueltas serán cargadas. Por omisión, es automáticamente buscada en la biblioteca estándar C.

Ejercicio A.9

Suponga que en el ejemplo A.23 la biblioteca matemática ha sido ligada, pero el archivo de encabezado `math.h` no fue incluido en la fuente. ¿Qué pasaría?

Respuesta:

El programa puede producir una respuesta incorrecta. Por ejemplo, el compilador asume que `log` tiene un valor de regreso del tipo `int` en lugar del tipo `double`. Si el programa llama la función `log`, el cálculo produciría un resultado numérico incorrecto. Es posible que el compilador no produzca un error o mensajes de advertencia. Sin embargo, `lint` (sección A.6) indica que `log` ha sido implícitamente declarado para regresar `int`.

Ejemplo A.24

El siguiente comando de ligado procesa archivos objeto en el orden `my.o`, *la biblioteca matemática y después* `mylib.o`.

```
cc -o my my.o -lm ,ylib.o
```

Sólo aquellos objetos en la biblioteca que correspondan a referencias no resueltas son incluidos en el módulo ejecutable. Por lo tanto, si `mylib.o` del ejemplo A.24 contuviera una referencia a la biblioteca matemática, esa referencia no sería resuelta.

Indicar `-lx` es una abreviación usada ya sea para `libx.a` (*un archivo de biblioteca)* o para `libx.so` (*una biblioteca compartida*). Cuál de ellas se carga por omisión, depende de cómo se instaló el sistema. Para un archivo de biblioteca, especifique `-Bstatic -lx` en el comando `cc` y para una biblioteca compartida especifique `-Bdynamic -lx` en el mismo comando. El compilador rastrea las *bibliotecas compartidas* para encontrar referencias, pero no coloca las funciones dentro del archivo ejecutable de salida. En vez de esto son cargados durante el tiempo de ejecución usando cargado y enlazado dinámico.

En un sistema, pueden existir diferentes versiones de una biblioteca en particular, por lo menos una por cada versión del compilador C. El orden en que los directorios buscan las bibliotecas depende del sistema. A continuación presentamos la siguiente estrategia típica:

- Directorios `-L`,
- Directorios indicados por `LD_LIBRARY_PATH`,
- Directorios para bibliotecas estándares (por ejemplo, `/usr/lib`).

La opción `-L` de `cc` es utilizada para especificar explícitamente nombres de ruta de directorios que serán buscados para ver si contienen bibliotecas. La variable de entorno `LD_LIBRARY_PATH` puede ser utilizada para especificar nombres de ruta predeterminados que serán buscados para ver si contienen bibliotecas de carga, generalmente nombres de ruta de directorios donde los compiladores han sido instalados, así como directorios tales como `/usr/local/lib`. Es probable que el administrador del sistema haya iniciado la variable `LD_LIBRARY_PATH` para ser usada con compiladores estándar. (Véase la sección A.7.)

A.6 Ayudas para depuración

El programa `lint` encuentra errores e inconsistencias en programas fuentes C. En muchas áreas, es más detallado que el compilador C. Ejecuta pruebas más minuciosas, trata de detectar enunciados de código inalcanzables y señala código que podría ser innecesario o no portable. La herramienta `lint` también detecta una variedad de errores comunes, como el uso de = en lugar de == u omitir el & en argumentos de `scanf`. También explora buscando inconsistencias entre módulos. Es necesario que se utilice estas herramientas en todos los programas y que se analice seriamente los mensajes de advertencia resultantes. El compilador C asume que todo lo anterior ha sido efectuado e implementado en todos los programas, para así trabajar en forma rápida en lugar de detenerse en detalles.

Ejercicio A.10

Incluya las siguientes líneas en el archivo de descripción del ejemplo A.20 para ejecutar la herramienta `lint` en las fuentes.

```
lintall: my.c mylib.c myinc.h
         lint my.c mylib.c > my.lint
```

Escriba `make lintall` para ejecutar la herramienta `lint` en todos los programas. La salida de `lint` está en `my.lint`.

Ejercicio A.11

¿Cómo debe ser interpretado el siguiente mensaje de `lint`?

```
implicitly declared to return int:
   (14) strtok
```

Respuesta:

Este mensaje indica que en el programa no se incluyó el archivo de encabezado `string.h` asociado con `strtok` que aparece en la línea 14. Sin mayor información, el compilador asume que `strtok` regresa `int`. La falta del encabezado puede dar resultados desastrosos en el momento de la ejecución.

Ejercicio A.12

¿Cómo debe ser interpretado el siguiente mensaje de `lint`?

```
(5) warning: variable may be used before set: p
```

Respuesta:

Suponga que el mensaje de `lint` se refiere al siguiente segmento de código.

```
char *p;
scanf("%s", p);
```

El apuntador `p` no está apuntando a un almacenamiento temporal de caracteres (*buffer*) apropiado. El código se compila bien, pero probablemente producirá un error de segmentación a la hora de ser ejecutado.

Los depuradores (debuggers) son utilidades de la etapa de ejecución (runtime) que monitorean y controlan la ejecución del programa. Los depuradores (debuggers) UNIX conocidos incluyen a `dbx`, `adb`, `sdb` y `debug`. Los depuradores (debuggers) le permiten al usuario correr el programa paso por paso y monitorear cambios en variables especificadas. Para usar un depurador (debugger), compile el programa con la opción `-g`.

Ejercicio A.13

Compile el programa `my.c` con la opción `-g` de la forma siguiente para poder habilitar el ejecutable con un control de depurador (debugger).

```
cc -g -o my my.c
```

Ejecute `my` bajo el depurador (debugger) `dbx` escribiendo:

```
dbx my
```

El depurador (debugger) responde con el siguiente mensaje:

```
(dbx)
```

Responda con `help` para obtener una lista de comandos o use un `run` para ejecutar el programa. Establezca un punto de parada con `stop` o habilite la facilidad de marcar cuando una variable cambia mediante un `trace` antes de teclear un `run`.

Muchos programadores encuentran de mucha utilidad los depuradores, especialmente programadores novatos con problemas de apuntadores. Algunos depuradores tienen interfaces gráficas, lo cual hace que sean de fácil manejo. Los depuradores normales son menos útiles en un entorno concurrente donde los procesos interactúan o donde el tiempo puede cambiar el comportamiento de un programa. Los depuradores de hilos (threads debuggers) están también disponibles, pero en forma limitada. Los depuradores pueden ayudar a encontrar un error de ejecución en particular, pero el uso de un depurador no es un sustituto del plan de prueba del programa. Una buena detección de errores para llamadas del sistema es probablemente la mejor estrategia de depuración que hay que seguir.

Para una depuración efectuada al momento de ejecución (runtime debugging), el comando `truss` es útil. Produce un rastreo de llamadas del sistema y de señales (vistas en el capítulo 5) que son incurridas cuando un proceso en particular está siendo ejecutado. Use la opción `-f` en el comando `truss` para rastrear las llamadas de los hijos del proceso (visto en el capítulo 2). El comando `truss` no forma parte de POSIX o de Spec 1170, y no está disponible en todos los sistemas.

Ejercicio A.14

Suponga que un programa llamado `dvips` es instalado en un sistema y este programa no puede encontrar el archivo `psfonts.map` que necesita. Existe una copia disponible de este archivo `psfonts.map`, pero `dvips` asume que este archivo está en un directorio en particular. ¿Cómo podemos resolver este problema?

Respuesta:

Trate de ejecutar el siguiente comando (de un intérprete de comandos de C).

```
truss dvips -f t.dvi |& grep psfonts.map
```

El símbolo `|&` causa que ambos, la salida estándar y el error estándar de `truss`, se direccionen a la entrada estándar de `grep`. La salida puede verse como sigue:

```
open("./psfonts.map", O_RDONLY, 0666)        Err#2 ENOENT
open("/usr/local/tex/dvips/psfonts.map", O_RDONLY, 0666) Err#2 ENOENT
```

El programa `truss` ejecuta el comando `dvips -f t.dvi` y `grep` despliega las líneas de salida que contienen `psfonts.map`. La salida indica que el programa primero buscó en el directorio actual y luego en el directorio `/usr/local/tex/dvips`. Correr el programa `dvips` sin utilizar la herramienta `truss` sólo produce un mensaje `unable to open file`, que no da ninguna información adicional. ¡Copie el archivo `psfonts.map` a `/usr/local/tex/dvips` y todo estará listo para empezar!

La mayoría de compiladores C tienen opciones para generar el perfil de ejecución de programas. Este tipo de perfil acumula información estadística, como tiempo de ejecución de bloques básicos y frecuencia de las llamadas. Consulte las páginas del manual para `prof`,

`gprof`, `monitor`, `profil` y `tcov`, así como para `cc`, a fin de obtener información adicional acerca de estos programas de generación de perfiles.

A.7 Ambiente del usuario

Cuando un usuario entra en el sistema, un intérprete de líneas de comando llamado *intérprete de comandos (shell)* es ejecutado. El *intérprete de comandos (shell)* es un programa que despliega un indicador, espera por un comando y lo ejecuta. Cuando el indicador aparece en la pantalla de la terminal, el intérprete de comandos ha desplegado su mensaje del indicador y está en espera de una entrada.

Tres intérpretes de comandos UNIX comunes en el mercado son el intérprete C (`csh`), el intérprete Bourne (`sh`) y el intérprete KornShell (ksh). El intérprete C (c-shell) ejecuta comandos de arranque de un archivo llamado `.cshrc` cuando comienza la ejecución. Los intérpretes Bourne y KornShell obtienen sus comandos de arranque de un archivo llamado `.profile`. Los usuarios pueden adecuar su ambiente de ejecución colocando los comandos apropiados en los archivos de arranque del intérprete de comandos. POSIX.2 ha normalizado ciertos aspectos de intérpretes de comandos, basados en el intérprete KornShell.

UNIX utiliza *variables de ambiente* para adecuar su uso e indicar preferencias del usuario. En cierta forma, las *variables de ambiente* son parámetros globales que le indican a las utilidades y aplicaciones del sistema cómo se deben hacer las cosas. Las variables de ambiente más comunes incluyen:

- `HOME`=directorio hogar
- `SHELL`=intérprete de comandos del usuario
- `PATH`=nombres de ruta de directorios que serán buscados para localizar comandos por ejecutar
- `LOGNAME`=nombre de entrada
- `TERM`=tipo de terminal
- `USER`=nombre del usuario
- `MANPATH`=nombres de ruta de directorios que serán buscados cuando se localicen páginas del manual
- `DISPLAY`=Despliegue que se usará para el Sistema de Ventanas X

Las variables de ambiente anteriores son normalmente iniciadas cuando una cuenta es generada. Otras variables de ambiente pueden ser definidas según se necesite. Los nombres de variables de ambiente son tradicionalmente escritas en mayúsculas. Generalmente el usuario coloca variables de ambiente de uso frecuente en el archivo de arranque del intérprete de comandos, para que estas variables sean definidas al momento de entrada al sistema. El comando para enlistar las variables de ambiente y sus valores cuando se usa el intérprete C es `setenv`, mientras que para los intérpretes Bourne y KornShell se usa el comando `export`.

Ejercicio A.15

Pruebe los comandos `setenv` y `export` para determinar si el intérprete de comandos actual es un intérprete C o un KornShell. Si ninguno de estos comandos funciona,

llame a un administrador del sistema para mayor información sobre el intérprete de comandos.

Ejercicio A.16

La variable de ambiente `EDITOR` especifica qué editor se usará para editar mensajes cuando se esté utilizando correo electrónico. Introduzca la siguiente línea en `.cshrc` para usar `vi` cuando se esté editando correo electrónico bajo el intérprete de comandos C:

```
setenv EDITOR vi
```

Bajo los intérpretes de comandos Bourne o KornShell introduzca la siguiente línea en `.profile`:

```
export EDITOR=vi
```

La *lista de variables de ambiente* es un arreglo de apuntadores a *cadenas de caracteres ambientales* de la forma *name=value*. La porción *name* es tradicionalmente especificada en mayúsculas y *value* es una cadena de caracteres terminada en nulo. El comando `env` modifica al ambiente actual y luego llama a una utilidad especificada con el ambiente modificado.

SINOPSIS

```
env [-i] [name=value]... [utility [argument..]]
```

POSIX.1, Spec 1170

Si el parámetro `utility` es omitido, `env` saca su listado del ambiente actual hacia una salida estándar. El parámetro `argument` es una cadena de argumentos de línea de comando para `utility`.

Las aplicaciones utilizan la lista de variables de ambiente para obtener información acerca del ambiente del proceso. Un programa accesa su lista ambiental a través de la variable externa `environ` que apunta a un arreglo de apuntadores a cadenas de caracteres ambientales terminados en nulo, similar al arreglo `argv`.

Ejemplo A.25

El siguiente programa en C imprime la lista ambiental.

```
#include <stdlib.h>
#include <stdio.h>
extern char **environ;
void main(int argc, char *argv[])
{
    int i;
    for (i = 0; *(environ + i) !=NULL; i++)
        printf("%s\n", *(environ + i));
    exit(0);
}
```

La manera más fácil de encontrar una variable de ambiente en un programa en particular es con `getenv`.

SINOPSIS

```
#include <stdlib.h>

char *getenv(const char *name);
```

POSIX.1, Spec 1170

La función `getenv` regresa al apuntador al valor asociado con `name` o con `NULL` si es que no hay asociación.

Ejemplo A.26

El siguiente segmento imprime el valor de la variable ambiental `DISPLAY`.

```
#include <stdio.h>
#include <stdlib.h>

char *displayp;
displayp = getenv("DISPLAY");
if (displayp != NULL)
   printf("The DISPLAY variable is %s\n", displayp);
else
   printf("The DISPLAY variable is undefined\n");
```

A.8 Lecturas adicionales

El texto *UNIX SYSTEM V: A Practical Guide*, 3ª edición, de Sobell [81], es una referencia actualizada de cómo usar las utilidades de UNIX. El libro *UNIX System Administration Handbook*, 2ª edición, de Nemeth y otros [50], es una excelente introducción de fácil lectura para muchos de los temas de configuración que se presentan cuando se está instalando sistemas UNIX. O'Reilly Press tiene una gran cantidad de libros relacionados con muchos de los temas tratados en este apéndice, incluyendo `emacs` [15], las bibliotecas [22], `lint` [23], `make` [88] y `vi` [47].

Apéndice B

Implementación de UICI

Este apéndice contiene código fuente para las tres implementaciones de UICI que se vieron en el capítulo 12.

B.1 Prototipos de UICI

Programa B.1: El archivo `uici.h`, que contiene los prototipos de las funciones de UICI.

```
/************************************ uici.h ************************************/
/*   diversas definiciones necesarias para las funciones de UICI   */
/********************************************************************************/
typedef unsigned short u_port_t;
int u_open(u_port_t port);
int u_listen(int fd, char *hostn);
int u_connect(u_port_t port, char *inetp);
int u_close(int fd);
ssize_t u_read(int fd, void *buf, size_t nbyte);
ssize_t u_write(int fd, void *buf, size_t nbyte);
void u_error(char *s);
int u_sync(int fd);
```

Programa B.1

B.2 Implementación con *sockets*

El programa B.2 es un programa cliente para probar UICI; acepta dos argumentos de línea de comandos, una máquina de destino y un número de puerto. Después de establecer una conexión, el cliente envía a la máquina remota todo lo que llega por la entrada estándar. El cliente atrapa las señales `SIGUSR1` y `SIGUSR2`, demostrando que la transmisión funciona correctamente incluso cuando hay señales presentes. El programa utiliza bloques de tamaño pequeño para que la transmisión de un archivo requiera un gran número de transferencias incluso si el tamaño del archivo es moderado. El cliente termina cuando detecta un fin de archivo en la entrada estándar.

Programa B.2: Programa cliente para probar las implementaciones de UICI con *sockets* y TLI.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <sys/types.h>
#include "uici.h"

#define BLKSIZE  1024

void usr1handler(int s)
{
   fprintf(stderr, "SIGUSR1 signal caught\n");
}

void usr2handler(int s)
{
   fprintf(stderr, "SIGUSR2 signal caught\n");
}


void installusrhandlers()
{
    struct  sigaction  newact;
    newact.sa_handler = usr1handler;  /* establecer nuevo controlador usr1 */
    sigemptyset(&newact.sa_mask);          /* no se bloquean otras señales */
    newact.sa_flags = 0;                  /* nada especial en las opciones */
    if (sigaction(SIGUSR1, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR1 signal handler");
       return;
    }
    newact.sa_handler = usr2handler;  /* establecer nuevo controlador usr2 */
    if (sigaction(SIGUSR2, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR2 signal handler");
```

```
        return;
    }
    fprintf(stderr,
        "Client process %ld set to use SIGUSR1 and SIGUSR2\n",
        (long)getpid());
}

void main(int argc, char *argv[])
/*
 *  Esta es una prueba de cliente de UICI; abre una conexión
 *  a una máquina especificada por nodo y número de puerto;
 *  lee un archivo de stdin en bloques de tamano BLKSIZE y lo
 *  envía a la conexión.
 */
{
    unsigned short portnumber;
    int outfd;
    ssize_t bytesread;
    ssize_t byteswritten;
    char buf[BLKSIZE];

    if (argc != 3) {
        fprintf(stderr, "Usage: %s host port\n", argv[0]);
        exit(1);
    }

    installusrhandlers();

    portnumber = (unsigned short)atoi(argv[2]);

    if ((outfd = u_connect(portnumber, argv[1])) < 0) {
        u_error("Unable to establish an Internet connection");
        exit(1);
    }

    fprintf(stderr, "Connection has been made to %s\n", argv[1]);

    for ( ; ; ) {
        bytesread = read(STDIN_FILENO, buf, BLKSIZE);
        if ( (bytesread == -1) && (errno == EINTR) )
            fprintf(stderr, "Client restarting read\n");
        else if (bytesread <= 0) break;
        else {
            byteswritten = u_write(outfd, buf, bytesread);
            if (byteswritten != bytesread) {
                fprintf(stderr,
                    "Error writing %ld bytes, %ld bytes written\n",
                    (long)bytesread, (long)byteswritten);
                break;
```

```
            }
        }
    }
    u_close(outfd);
    exit(0);
}
```

Programa B.2

El programa B.3 es un programa servidor para probar UICI; acepta un solo argumento de línea de comandos, que es el puerto en el que escuchará. Una vez establecida una conexión, el servidor lee información de la red y la envía a la salida estándar. El programa termina cuando detecta un fin de archivo del nodo remoto.

Programa B.3: Programa servidor para probar las implementaciones de UICI con *sockets* y TLI.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/uio.h>
#include "uici.h"

#define BLKSIZE  1024
void usr1handler(int s)
{
   fprintf(stderr, "SIGUSR1 signal caught by server\n");
}

void usr2handler(int s)
{
   fprintf(stderr, "SIGUSR2 signal caught by server\n");
}

void installusrhandlers()
{
    struct  sigaction  newact;
    newact.sa_handler = usr1handler;  /* establecer nuevo controlador usr1 */
    sigemptyset(&newact.sa_mask);          /* no se bloquean otras señales */
    newact.sa_flags = 0;                  /* nada especial en las opciones */
    if (sigaction(SIGUSR1, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR1 signal handler");
       return;
    }
    newact.sa_handler = usr2handler;  /* establecer nuevo controlador usr2 */
```

```
    if (sigaction(SIGUSR2, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR2 signal handler");
       return;
    }
    fprintf(stderr,
       "Server process %ld set to use SIGUSR1 and SIGUSR2\n",
       (long)getpid());
}

void main(int argc, char *argv[])
/*
 *  Esta es una prueba de servidor que abre una conexión con
 *  un número de puerto y escucha para detectar solicitud;
 *  luego abre un puerto de comunicación y lee de la conexión
 *  hasta que ésta se cancela. Se hace eco en la
 *  salida estándar de cada bloque leído.
 */
{
    unsigned short portnumber;
    int listenfd;
    int communfd;
    ssize_t bytesread;
    ssize_t byteswritten;
    char buf[BLKSIZE];
    char remote[MAX_CANON];

    if (argc != 2) {
        fprintf(stderr, "Usage: %s port\n", argv[0]);
        exit(1);
    }

    installusrhandlers();

    portnumber = (unsigned short)atoi(argv[1]);

    if ((listenfd = u_open(portnumber)) < 0) {
       u_error("Unable to establish a port connection");
       exit(1);
    }

    if ((communfd = u_listen(listenfd, remote)) < 0) {
       u_error("Failure to listen on server");
       exit(1);
    }
    fprintf(stderr, "Connection has been made to %s\n", remote);

    while( (bytesread = u_read(communfd, buf, BLKSIZE)) > 0) {
       byteswritten = write(STDOUT_FILENO, buf, bytesread);
       if (bytesread != byteswritten) {
```

```
            fprintf(stderr,
               "Error writing %ld bytes, %ld bytes written\n",
               (long)bytesread, (long)byteswritten);
            break;
         }
      }
      u_close(listenfd);
      u_close(communfd);
      exit(0);
}
```

Programa B.3

Programa B.4: Una implementación completa de UICI en términos de *sockets*.

```
/* uici.c  implementación con sockets */

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <errno.h>
#include "uici.h"

#define MAXBACKLOG 5

/* devolver 1 si hay error, 0 si OK */
int u_ignore_sigpipe()
{
   struct sigaction act;

   if (sigaction(SIGPIPE, (struct sigaction *)NULL, &act) < 0)
      return 1;
   if (act.sa_handler == SIG_DFL) {
      act.sa_handler = SIG_IGN;
      if (sigaction(SIGPIPE, &act, (struct sigaction *)NULL) < 0)
         return 1;
   }
   return 0;
}

/*
```

```
 *                              u_open
 * Devolver un descriptor de archivo vinculado con el puerto dado.
 *
 * parámetro:
 *         s = número de puerto con el cual vincularse
 * devuelve: descriptor de archivo si hay éxito y -1 si hay error
 */
int u_open(u_port_t port)
{
   int sock;
   struct sockaddr_in server;

   if ( (u_ignore_sigpipe() != 0) ||
        ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) )
      return -1;

   server.sin_family = AF_INET;
   server.sin_addr.s_addr = INADDR_ANY;
   server.sin_port = htons((short)port);

   if ( (bind(sock, (struct sockaddr *)&server, sizeof(server)) < 0) ||
        (listen(sock, MAXBACKLOG) < 0) )
      return -1;
   return sock;
}

/*
 *                              u_listen
 * Escuchar para detectar solicitud del nodo especificado
   en el puerto especificado.
 *
 * parámetros:
 *      fd = descriptor de archivo previamente vinculado
 *           al puerto en el que se escucha
 *      hostn = nombre del nodo que se va a escuchar
 * devuelve: descriptor de archivo para comunicación o -1 si hay error
 *
 * comentarios: El servidor usa esta función para escuchar
 * comunicación; se bloquea hasta que se recibe una solicitud
 * remota del puerto vinculado con el descriptor de archivo dado.
 * Se asigna a hostn una cadena ASCII que contiene el nombre
 * del nodo remoto; debe apuntar a una cadena lo bastante
 * grande como para contener este nombre.
 */
int u_listen(int fd, char *hostn)
{
   struct sockaddr_in net_client;
   int len = sizeof(struct sockaddr);
   int retval;
   struct hostent *hostptr;
```

```
   while ( ((retval =
           accept(fd, (struct sockaddr *)(&net_client), &len)) == -1) &&
          (errno == EINTR) )
      ;
   if (retval == -1)
      return retval;
   hostptr =
      gethostbyaddr((char *)&(net_client.sin_addr.s_addr), 4, AF_INET);
   if (hostptr == NULL)
      strcpy(hostn, "unknown");
   else
      strcpy(hostn, (*hostptr).h_name);
   return retval;
}

/*
 *                              u_connect
 * Iniciar la comunicación con un servidor remoto.
 *
 * parámetros:
 *     port = puerto bien conocido en servidor remoto
 *     inept = cadena de caracteres con el nombre de Internet
 *             de la máquina remota
 * devuelve: descriptor de archivo para comunicación o -1 si hay error
 */
int u_connect(u_port_t port, char *hostn)
{
   struct sockaddr_in server;
   struct hostent *hp;
   int sock;
   int retval;

   if ( (u_ignore_sigpipe() != 0) ||
        !(hp = gethostbyname(hostn)) ||
        ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) )
      return -1;

   memcpy((char *)&server.sin_addr, hp->h_addr_list[0], hp->h_length);

   server.sin_port = htons((short)port);
   server.sin_family = AF_INET;

   while ( ((retval =
       connect(sock, (struct sockaddr *)&server, sizeof(server))) == -1)
        && (errno == EINTR) )
      ;
   if (retval == -1) {
       close(sock);
       return -1;
```

```
    }
    return sock;
}

/*
 *                              u_close
 * Cerrar comunicación para el descriptor de archivo dado.
 * parámetro:
 *       fd = descriptor de archivo de la conexión de socket por cerrar
 * devuelve:
 *       un valor negativo indica que hubo un error
 */

int u_close(int fd)
{
    return close(fd);
}

/*
 *                              u_read
 *
 * Recuperar información de un descriptor de archivo abierto con u_open.
 *
 * parámetros:
 *     fd = descriptor de archivo de TLI
 *     buf = buffer para la salida
 *     nbyte = número de bytes por recuperar
 * devuelve:
 *     un valor negativo indica que hubo un error
 *     en caso contrario devuelve número de bytes leídos
 */

ssize_t u_read(int fd, void *buf, size_t size)
{
    ssize_t retval;

    while (retval = read(fd, buf, size), retval == -1 && errno == EINTR)

    return retval;
}

/*
 *                              u_write
 *
 * Enviar información a un descriptor de archivo abierto con u_open.
 *
 * parámetros:
 *     fd = descriptor de archivo de TLI
 *     buf = buffer para la salida
```

```
 *      nbyte = número de bytes por enviar
 * devuelve:
 *      un valor negativo indica que hubo un error
 *      en caso contrario devuelve número de bytes escritos
 */

ssize_t u_write(int fd, void *buf, size_t size)
{
   ssize_t retval;

   while (retval = write(fd, buf, size), retval == -1 && errno == EINTR)
      ;
   return retval;
}

/*
 *                              u_error
 *  Exhibir mensaje de error como hacen perror o t_error.
 *
 *  parámetro:
 *        s = cadena que precederá al mensaje de error del sistema
 *  devuelve:  0
 *
 *  algoritmo: Puesto que el único tipo de error proviene de una llamada de
 *             sistema, basta con llamar a perror. Esta implementación
 *             debe ser tan segura respecto de multihilos como los sockets
 *             subyacentes.
 */

void u_error(char *s)
{
   perror(s);
}

/*
 *                              u_sync
 *
 * Esta función debe invocarse después de exec o dup para anexar un
 * descriptor de archivo abierto en implementaciones sencillas de
 * UICI. No se necesita cuando se usan sockets.
 *
*/
/*ARGSUSED*/
int u_sync(int fd)
{
/* No se necesita para sockets. Se conserva por compatibilidad */
   return 0;
}
```

Programa B.4

B.3 Implementación TLI

La implementación TLI puede utilizar los mismos programas de prueba para el cliente y el servidor dados en los programas B2 y B3 para *sockets*.

Programa B.5: Implementación completa de UICI en términos de TLI.

```
/* uici.c  Implementación TLI */

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <tiuser.h>
#include <errno.h>
#include <string.h>
#include "uici.h"

#define RETRIES 5

/*
 *                              u_open
 * Devolver un descriptor de archivo vinculado con el puerto dado.
 *
 * parámetro:
 *      s = número de puerto al cual vincularse
 * devuelve: descriptor de archivo si hay éxito y -1 si hay error
 */
int u_open(u_port_t port)
{
   int fd;
   struct t_info info;
   struct t_bind req;
   struct sockaddr_in server;

   fd = t_open("/dev/tcp", O_RDWR, &info);
   if (fd < 0)
      return -1;

   /* Crear servidor con comodines y vincular socket a puerto */
   memset(&server, 0, (size_t)sizeof(server));
   server.sin_family = AF_INET;
   server.sin_addr.s_addr = INADDR_ANY;
   server.sin_port = htons((short)port);
```

```
    req.addr.maxlen = sizeof(server);
    req.addr.len = sizeof(server);
    req.addr.buf = (char *)&server;
    req.qlen = 1;

    if (t_bind(fd, &req, NULL) < 0)
       return -1;
    return fd;
}

/*
 *                                    u_listen
 * Escuchar para detectar solicitud de cierto nodo en el puerto especificado.
 *
 * parámetros:
 *       fd = descriptor de archivo previamente vinculado al puerto en el
 *            que se escucha
 *       hostn = nombre del nodo que se va a escuchar
 * devuelve: descriptor de archivo para comunicación o -1 si hay error
 *
 * comentarios: El servidor usa esta función para escuchar
 * comunicación; se bloquea hasta que recibe una solicitud
 * remota del puerto vinculado con el descriptor de archivo dado.
 * Se asigna a hostn una cadena ASCII que contiene el nombre
 * del nodo remoto; debe apuntar a una cadena lo bastante
 * grande como para contener este nombre.
 */
int u_listen(int fd, char *hostn)
{
    struct t_call *callptr;
    struct sockaddr_in *client;
    struct hostent *hostptr;
    int newfd;
    int tret;

    if ( (callptr =
          (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
       return -1;
    }

    while( ((tret = t_listen(fd, callptr)) < 0) &&
           (t_errno == TSYSERR) && (errno == EINTR) )
       ;                          /* se interrumpió y reinició t_listen */
    if ( (tret < 0) ||
         ((newfd = t_open("/dev/tcp", O_RDWR, NULL)) < 0 ) ||
         (t_bind(newfd, NULL, NULL) < 0 ) ||
         (t_accept(fd, newfd, callptr) < 0) ) {
       t_free((char *)callptr, T_CALL);
       return -1;
    }
```

```
    client = (struct sockaddr_in *)(callptr->addr).buf;
    hostptr =
        gethostbyaddr((char *)&(client->sin_addr.s_addr), 4, AF_INET);
    if (hostptr == NULL)
        strcpy(hostn, "unknown");
    else
        strcpy(hostn, hostptr->h_name);
    t_free((char *)callptr, T_CALL);
    return newfd;
}

/*
 *                              u_connect
 * Iniciar la comunicación con un servidor remoto.
 *
 * parámetros:
 *     port = puerto bien conocido en servidor remoto
 *     inept = cadena de caracteres con el nombre de Internet
 *             de la máquina remota
 * devuelve: descriptor de archivo para comunicación o -1 si hay error
 */
int u_connect(u_port_t port, char *inetp)
{
    struct t_info info;
    struct t_call  *callptr;
    struct sockaddr_in server;
    struct hostent *hp;
    int fd;
    int trynum;
    unsigned int slptime;

    fd = t_open("/dev/tcp", O_RDWR, &info);
    if (fd < 0)
        return -1;

    /* Crear direcciones TCP para comunicarse con el puerto */
    memset(&server, 0, (size_t)sizeof(server));
    server.sin_family = AF_INET;        /* familia de direcciones de Internet */
    hp = gethostbyname(inetp);     /* convertir nombre en dirección de Internet */
                                    /* copiar dirección de Internet en buffer */
    if (hp == NULL) {
        fprintf(stderr, "gethostbyname returned NULL\n");
        return -1;
    }
    memcpy(&(server.sin_addr.s_addr), hp->h_addr_list[0],
           (size_t)hp->h_length);
    server.sin_port = htons((short)port);

                            /* establecer buffer con información del destino */
```

```
    if (t_bind(fd, NULL, NULL) < 0)
        return -1;

                                /* asignar memoria a estructura para conectarse */
    if ( (callptr =
            (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL)
        return -1;

    callptr -> addr.maxlen = sizeof(server);
    callptr -> addr.len = sizeof(server);
    callptr -> addr.buf = (char *)&server;
    callptr -> opt.len = 0;
    callptr -> udata.len = 0;

                        /* Seguir intentando hasta que el servidor esté listo */
    for (trynum=0,slptime=1;
         (trynum < RETRIES) &&
         (t_connect(fd, callptr, NULL) < 0);
         trynum++,slptime=2*slptime) {
        t_rcvdis(fd, NULL);
        sleep(slptime);
    }
    callptr -> addr.buf = NULL;
    t_free((char *)callptr, T_CALL);
    if (trynum >= RETRIES)
        return -1;
    return fd;
}

/*
 *                              u_close
 * Cerrar comunicación para el descriptor de archivo TLI dado.
 * parámetro:
 *       fd = descriptor de archivo de la conexión TLI por cerrar
 * devuelve:
 *       un valor negativo indica que hubo un error
 */
int u_close(int fd)
{
    return t_close(fd);
}

/*
 *                              u_read
 *
 * Recuperar información de un descriptor de archivo abierto con u_open.
 *
 * parámetros:
 *       fd = descriptor de archivo de TLI
```

```
 *       buf = buffer para la salida
 *       nbyte = número de bytes por recuperar
 * devuelve:
 *      un valor negativo indica que hubo un error
 *      en caso contrario devuelve número de bytes leídos
 */

ssize_t u_read(int fd, void *buf, size_t nbyte)
{
   int rcvflag;
   ssize_t retval;

   rcvflag = 0;
   while ( ((retval=
           (ssize_t)t_rcv(fd, buf, (unsigned)nbyte, &rcvflag)) == -1) &&
           (t_errno == TSYSERR) && (errno == EINTR) )
       ;
   return retval;
}

/*
 *                          u_write
 *
 * Enviar información a un descriptor de archivo abierto con u_open
 *
 * parámetros:
 *    fd = descriptor de archivo de TLI
 *    buf = buffer para la salida
 *    nbyte = número de bytes por enviar
 * devuelve:
 *    un valor negativo indica que hubo un error
 *    en caso contrario devuelve número de bytes escritos
 */
ssize_t u_write(int fd, void *buf, size_t nbyte)
{
   ssize_t retval;
   while ( ( (retval =
           (ssize_t)t_snd(fd, buf, (unsigned)nbyte,0)) == -1) &&
           (t_errno == TSYSERR) && (errno == EINTR) )
      ;
   return retval;
}

/*
 *                          u_error
 *  Exhibir mensaje de error como hacen perror o t_error.
 *
 *  parámetro:
 *     s = cadena que precederá al mensaje de error del sistema
```

```
 *  devuelve:  0
 *
 *  algoritmo: Puesto que el único tipo de error proviene de una
 *             llamada TLI, basta con llamar a t_error. Esta implementación
 *             debe ser tan segura en relación con los multihilos como la
 *             TLI subyacente.
 */
void u_error(char *s)
{
   t_error(s);
}

/*
 *                          u_sync
 *
 * Esta función debe invocarse después de exec o dup para anexar un
 * descriptor de archivo abierto a las rutinas TLI
 *
 * parámetro:
 *       fd = descriptor del archivo por anexar
 * devuelve: -1 si hay error y algo distinto si tiene éxito
 */
int u_sync(int fd)
{
   return t_sync(fd);
}
```

Programa B.5

B.4 Implementación con flujos

Programa B.6: Programa cliente para probar la implementación de UICI con flujos.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "uici.h"

#define BLKSIZE 1024

void main(int argc, char *argv[])
/*
 *                          Cliente UICI
 *  Solicitar conexión UICI al nodo y puerto especificados
 *  como argumentos de línea de comandos. Leer de la entrada
 *  estándar y escribir en el descriptor de archivo para
 *  comunicación UICI hasta fin de archivo.
```

```
 */
{
   u_port_t portnumber;
   int communfd;
   int bytesread;
   int done = 0;
   char buf[BLKSIZE];

   if (argc != 2) {
      fprintf(stderr, "Usage: %s port\n", argv[0]);
      exit(1);
   }

   portnumber = (u_port_t)atoi(argv[1]);

   if ((communfd = u_connect(portnumber, NULL)) < 0) {
      u_error("Unable to establish a connection");
      exit(1);
   }
   fprintf(stderr, "A connection has been made.\n");

   while (!done) {
      bytesread = read(STDIN_FILENO, buf, BLKSIZE);
      if (bytesread == 0) {
         fprintf(stderr, "Client detected end of file on input.\n");
         done = 1;
      }
      else if (bytesread < 0) {
         fprintf(stderr, "Client read error.\n");
         done = 1;
      }
      else {
         done = (bytesread != u_write(communfd, buf, bytesread));
         if (done)
            u_error("Client write_error");
      }
   }
   u_close(communfd);
   exit(0);
}
```

Programa B.6

Programa B.7: Programa servidor para probar la implementación de UICI con flujos.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#include <limits.h>
#include "uici.h"

#define BLKSIZE 1024

void main(int argc, char *argv[])
/*
 *                    Servidor UICI
 *  Abrir puerto UICI especificado como argumento de línea de
 *  comandos y escuchar para detectar solicitud. Cuando llega
 *  solicitud, usar el descriptor de archivo para comunicación
 *  provisto para leer de la conexión UICI y hacer eco en
 *  until the connection is terminated.
 */
{
   u_port_t portnumber;
   int listenfd, communfd;
   int bytesread;
   int done = 0;
   char buf[BLKSIZE];
   char client[MAX_CANON];

   if (argc != 2) {
      fprintf(stderr, "Usage: %s port\n", argv[0]);
      exit(1);
   }

   portnumber = (u_port_t)atoi(argv[1]);
   if ((listenfd = u_open(portnumber)) == -1) {
      u_error("Unable to establish a port connection");
      exit(1);
   }

   if ((communfd = u_listen(listenfd, client)) == -1) {
      u_error("Failure to listen on server");
      exit(1);
   }
   fprintf(stderr, "A connection has been made.\n");

   while(!done) {
      bytesread = u_read(communfd, buf, BLKSIZE);
      if (bytesread == 0) {
         fprintf(stderr, "Server detected end of file on input.\n");
         done = 1;
      }
      else if (bytesread < 0) {
         u_error("Server read error");
         done = 1;
      }
```

```
        else {
            done = (bytesread != write(STDOUT_FILENO, buf, bytesread));
            if (done)
                u_error("Server write error");
        }
    }
    u_close(communfd);
    u_close(listenfd);
    exit(0);
}
```

Programa B.7

Programa B.8: Implementación de UICI con flujos.

```
/* Funciones UICI para la implementación con STREAMS */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/conf.h>
#include <sys/types.h>
#include <fcntl.h>
#include <stropts.h>

#include "uici.h"

#define MTMODE 0600

/* esto ya está en sys/stropts.h sin la typedef */
typedef struct strrecvfd1 {
    union {
        struct file *fp;
        int fd;             /* descriptor nuevo */
    } f;
    uid_t   uid;    /* ID de usuario efectivo del emisor */
    uid_t   gid;    /* ID de grupo efectivo del emisor */
    char    fill[8];
} mystrrecvfd;

/*
 *                          u_open
 * Devolver un descriptor de archivo vinculado con el puerto dado.
 *
 * parámetro:
 *        s = número de puerto con el cual vincularse
```

```
 * devuelve: descriptor de archivo si hay éxito y -1 si hay error
 */
int u_open(u_port_t port)
{
    char buffer[MAX_CANON];
    int mtpoint;
    int fd[2];

    sprintf(buffer, "/tmp/streams_uici_%d", (int)port);
    unlink(buffer);
    if ( ((mtpoint = creat(buffer, MTMODE)) != -1) &&
        (close(mtpoint) != -1) &&
        (pipe(fd) != -1) &&
        (ioctl(fd[1], I_PUSH, "connld") != -1) &&
        (fattach(fd[1], buffer) != -1) )
        return fd[0];
    close (fd[0]);
    close (fd[1]);
    return -1;
}

/*
 *                              u_listen
 * Escuchar para detectar solicitud de cierto nodo en el puerto especificado.
 *
 * parámetros:
 *      fd = descriptor de archivo previamente vinculado
 *           con el puerto en el que se escucha
 *      hostn = nombre del nodo que se va a escuchar
 * devuelve: descriptor de archivo para comunicación o -1 si hay error
 *
 * comentarios: El servidor usa esta función para escuchar
 * comunicación; se bloquea hasta que se recibe una solicitud
 * remota del puerto vinculado con el descriptor de archivo dado.
 * No se usa hostn porque las conexiones STREAMS no pueden usar la red.
 */
int u_listen(int fd, char *hostn)
{
   mystrrecvfd conversation_info;
   int retval;

   while ( ((retval =
          ioctl(fd, I_RECVFD, &conversation_info)) == -1) &&
         (errno == EINTR) )
      ;
   if (retval == -1)
      return -1;
   *hostn = 0;
   return conversation_info.f.fd;
}
```

```
/*
 *                              u_connect
 * Iniciar la comunicación con un servidor remoto.
 *
 * parámetros:
 *    port = puerto bien conocido en servidor remoto
 *    inept = no se usa
 * devuelve: descriptor de archivo para comunicación o -1 si hay error
 */
int u_connect(u_port_t port, char *hostn)
{
    char buffer[MAX_CANON];

    sprintf(buffer, "/tmp/streams_uici_%d", (int)port);
    return open(buffer, O_RDWR);
}

/*
 *                              u_close
 * Cerrar comunicación para el descriptor de archivo dado.
 * parámetro:
 *    fd = descriptor de archivo de la conexión por cerrar
 * devuelve:
 *    un valor negativo indica que hubo un error
 */
int u_close(int fd)
{
   return close(fd);
}

/*
 *                              u_read
 *
 * Recuperar información de un descriptor de archivo abierto con u_open.
 *
 * parámetros:
 *    fd = descriptor de archivo de TLI
 *    buf = buffer para la salida
 *    nbyte = número de bytes por recuperar
 * devuelve:
 *    un valor negativo indica que hubo un error
 *    en caso contrario, devuelve número de bytes leídos
 */
ssize_t u_read(int fd, void *buf, size_t size)
{
   ssize_t retval;

   while (retval = read(fd, buf, size), retval == -1 && errno == EINTR)
```

```
        ;
    return retval;
}

/*
 *                              u_write
 *
 * Enviar información a un descriptor de archivo abierto con u_open
 *
 * parámetros:
 *     fd = descriptor de archivo de TLI
 *     buf = buffer para la salida
 *     nbyte = número de bytes por enviar
 * devuelve:
 *     un valor negativo indica que hubo un error
 *     en caso contrario, devuelve número de bytes escritos
 */

ssize_t u_write(int fd, void *buf, size_t size)
{
    ssize_t retval;

    while (retval = write(fd, buf, size), retval == -1 && errno == EINTR)
        ;
    return retval;
}

/*
 *                              u_error
 *  Exhibir mensaje de error como hacen perror o t_error.
 *
 *  parámetro:
 *          s = cadena que precederá al mensaje de error del sistema
 *  devuelve:  0
 *
 *  algoritmo: Puesto que el único tipo de error proviene de una llamada
 *             de sistema, basta con llamar a perror. Esta implementación
 *             debe ser tan segura respecto de multihilos como la TLI
 *             subyacente.
 */

void u_error(char *s)
{
    perror(s);
}

/*
 *                              u_sync
 *
 * Esta función debe invocarse después de exec o dup para
```

```
 * anexar un descriptor de archivo abierto en implementaciones sencillas
 * de UICI. No se necesita cuando se usan flujos.
 *
*/
/*ARGSUSED*/
int u_sync(int fd)
{
/* No se necesita para flujos. Se conserva por compatibilidad */
   return 0;
}
```

Programa B.8

B.5 Implementación de UICI segura respecto de los hilos

Esta sección contiene el código fuente completo para una UICI segura respecto de los hilos implementada con TLI.

Programa B.9: El archivo `uici.h` que contiene los prototipos de la funciones UICI para la versión segura respecto de los hilos.

```
/********************************* uici.h ********************************/
/*  Diversas definiciones necesarias para las funciones UICI.            */
/*  Estas versiones seguras respecto de hilos devuelven un número de error. */
/*************************************************************************/
#define GETHOSTNOERROR     0
#define GETHOSTBYNAMEERROR 1
#define GETHOSTBYADDRERROR 2
#define HOSTBUFFERLENGTH 100
typedef struct {
   int tli_error;
   int syserr;
   int hosterr;
} uerror_t;
int u_open_r(unsigned short port, uerror_t *errorp);
int u_listen_r(int fd, char *hostn, uerror_t *errorp);
int u_connect_r(unsigned short port, char *inetp, uerror_t *errorp);
int u_close_r(int fd, uerror_t *errorp);
ssize_t u_read_r(int fd, void *buf, size_t nbyte, uerror_t *errorp);
ssize_t u_write_r(int fd, void *buf, size_t nbyte, uerror_t *errorp);
void u_error_r(char *s, uerror_t error);
int u_sync_r(int fd, uerror_t *errorp);
```

Programa B.9

Programa B.10: Programa cliente similar al programa B.2, que utiliza la versión de UICI segura respecto de los hilos.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <errno.h>
#include <sys/types.h>
#include "uici.h"

#define BLKSIZE  100

void usr1handler(int s)
{
   fprintf(stderr, "SIGUSR1 signal caught\n"
}

void usr2handler(int s)
{
   fprintf(stderr, "SIGUSR2 signal caught\n");
}


void installusrhandlers()
{
    struct  sigaction  newact;
    newact.sa_handler = usr1handler;  /* establecer nuevo controlador usr1 */
    sigemptyset(&newact.sa_mask);          /* no se bloquean otras señales */
    newact.sa_flags = 0;                  /* nada especial en las opciones */
    if (sigaction(SIGUSR1, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR1 signal handler");
       return;
    }
    newact.sa_handler = usr2handler;  /* establecer nuevo controlador usr2 */
    if (sigaction(SIGUSR2, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR2 signal handler");
       return;
    }
    fprintf(stderr,
       "Client process %ld set to use SIGUSR1 and SIGUSR2\n",
       (long)getpid());
}

void main(int argc, char *argv[])
/*
 *  Esta es una prueba de cliente de UICI; abre una conexión
 *  a una máquina especificada por nodo y número de puerto.
```

```
 *  lee un archivo de stdin en bloques de tamaño BLKSIZE y lo
 *  envía a la conexión.
 */
{
    unsigned short portnumber;
    int outfd;
    ssize_t bytesread;
    ssize_t byteswritten;
    char buf[BLKSIZE];
    uerror_t error;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s host port\n", argv[0]);
        exit(1);
    }

    installusrhandlers();

    portnumber = (unsigned short)atoi(argv[2]);

    if ((outfd = u_connect_r(portnumber, argv[1], &error)) < 0) {
       u_error_r("Unable to establish an Internet connection", error);
       exit(1);
    }

    fprintf(stderr, "Connection has been made to %s\n", argv[1]);

    for ( ; ; ) {
         bytesread = read(STDIN_FILENO, buf, BLKSIZE);
         if ( (bytesread == -1) && (errno == EINTR) )
            fprintf(stderr,"Client restarting read\n");
         else if (bytesread <= 0) break;
         else {
            byteswritten =
               u_write_r(outfd, buf, bytesread, (uerror_t *)NULL);
            if (byteswritten != bytesread) {
               fprintf(stderr,
                  "Error writing %ld bytes, %ld bytes written\n",
                  (long)bytesread, (long)byteswritten);
               break;
            }
         }
    }
    u_close_r(outfd, (uerror_t *)NULL);
    exit(0);
}
```

Programa B.10

Programa B.11: Programa servidor similar al programa B.3 que utiliza la versión de UICI segura respecto de los hilos.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/unistd.h>
#include <sys/uio.h>
#include "uici.h"

#define BLKSIZE  100
void usr1handler(int s)
{
   fprintf(stderr, "SIGUSR1 signal caught by server\n");
}

void usr2handler(int s)
{
   fprintf(stderr, "SIGUSR2 signal caught by server\n");
}

void installusrhandlers()
{
    struct  sigaction  newact;
    newact.sa_handler = usr1handler;  /* establecer nuevo controlador usr1 */
    sigemptyset(&newact.sa_mask);          /* no se bloquean otras señales */
    newact.sa_flags = 0;                  /* nada especial en las opciones */
    if (sigaction(SIGUSR1, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR1 signal handler");
       return;
    }
    newact.sa_handler = usr2handler; /* establecer nuevo controlador usr2 */
    if (sigaction(SIGUSR2, &newact, (struct sigaction *)NULL) == -1) {
       perror("Could not install SIGUSR2 signal handler");
       return;
    }
    fprintf(stderr,
       "Server process %ld set to use SIGUSR1 and SIGUSR2\n",
       (long)getpid());
}

void main(int argc, char *argv[])
/*
 *  Esta es una prueba de servidor que abre una conexión con
 *  un número de puerto y escucha para detectar solicitud;
```

```
 *  luego abre un puerto de comunicación y lee de la conexión
 *  hasta que ésta se cancela. Se hace eco en la
 *  salida estándar de cada bloque leído.
 */
{
    unsigned short portnumber;
    int listenfd;
    int communfd;
    ssize_t bytesread;
    ssize_t byteswritten;
    char buf[BLKSIZE];
    char remote[MAX_CANON];
    uerror_t error;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s port\n", argv[0]);
        exit(1);
    }

    installusrhandlers();
    portnumber = (unsigned short)atoi(argv[1]);

    if ((listenfd = u_open_r(portnumber, &error)) < 0) {
       u_error_r("Unable to establish a port connection", error);
       exit(1);
    }

    if ((communfd = u_listen_r(listenfd, remote, &error)) < 0) {
       u_error_r("Failure to listen on server", error);
       exit(1);
    }
    fprintf(stderr, "Connection has been made to %s\n", remote);

    while((bytesread =
       u_read_r(communfd, buf, BLKSIZE, (uerror_t *)NULL)) > 0) {
       byteswritten = write(STDOUT_FILENO, buf, bytesread);
       if (bytesread != byteswritten) {
          fprintf(stderr,
            "Error writing %ld bytes, %ld bytes written\n",
            (long)bytesread, (long)byteswritten);
          break;
       }
    }
    u_close_r(listenfd, (uerror_t *)NULL);
    u_close_r(communfd, (uerror_t *)NULL);
    exit(0);
}
```

Programa B.11

Programa B.12: Implementación completa de UICI segura respecto de hilos en términos de TLI.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <tiuser.h>
#include <errno.h>
#include <string.h>
#include "uici.h"

#define RETRIES 5

static void u_set_error(uerror_t *errorp)
{
   if (errorp == NULL)
      return;
   errorp -> hosterr = GETHOSTNOERROR;
   errorp -> tli_error = t_errno;
   if (t_errno == TSYSERR)
      errorp ->syserr = errno;
}

/*
 *                          u_open_r
 * Devolver un descriptor de archivo vinculado con el puerto dado.
 *
 * parámetro:
 *     s = número de puerto con el cual vincularse
 * devuelve: descriptor de archivo si hay éxito y -1 si hay error.
 *    Si hay error, se asigna un número de error al último
 *    parámetro; si no hay error, *errnump no cambia.
 */
int u_open_r(unsigned short port, uerror_t *errorp)
{
  int fd;
  struct t_info info;
  struct t_bind req,bret;
  struct sockaddr_in server;
  struct sockaddr_in bindinfo;

  fd = t_open("/dev/tcp", O_RDWR, &info);
  if (fd < 0) {
     u_set_error(errorp);
     return -1;
  }
```

```
        /* Crear servidor con comodines y vincular socket a puerto */
  memset(&server, 0, (size_t)sizeof(server));
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = htonl(INADDR_ANY);
  server.sin_port = htons((short)port);

  req.addr.maxlen = sizeof(server);
  req.addr.len = sizeof(server);
  req.addr.buf = (char *)&server;
  req.qlen = 1;

  bret.addr.maxlen = sizeof(bindinfo);
  bret.addr.len = sizeof(bindinfo);
  bret.addr.buf = (char *)&bindinfo;

  if (t_bind(fd, &req, &bret) < 0) {
     u_set_error(errorp);
     return -1;
  }
  return fd;
}

/*
 *                          u_listen_r
 * Escuchar para detectar solicitud de cierto nodo en el puerto especificado.
 *
 * parámetros:
 *      fd = descriptor de archivo previamente vinculado
 *           con el puerto en el que se escucha
 *      hostn = nombre del nodo que se va a escuchar
 * devuelve: descriptor de archivo para comunicación o -1 si hay error
 *
 * comentarios: El servidor usa esta función para escuchar
 * comunicación; se bloquea hasta que recibe una solicitud
 * remota del puerto vinculado con el descriptor de archivo dado.
 * Se asigna a hostn una cadena ASCII que contiene el nombre
 * del nodo remoto; debe apuntar a una cadena lo bastante
 * grande como para contener este nombre.
 */
int u_listen_r(int fd, char *hostn, uerror_t *errorp)
{
   struct t_call *callptr;
   struct sockaddr_in *client;
   struct hostent *hostptr;
   struct hostent hostresult;
   int newfd;
   int tret;
   int herror;
   char hostbuf[HOSTBUFFERLENGTH];
```

```
   if ( (callptr =
         (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) {
      u_set_error(errorp);
      return -1;
   }

   while( ((tret = t_listen(fd, callptr)) < 0) &&
          (t_errno == TSYSERR) && (errno == EINTR) )
          fprintf(stderr, "t_listen interrupted and restarted\n");
   if  ( (tret < 0) ||
         ((newfd = t_open("/dev/tcp", O_RDWR, NULL)) < 0 ) ||
         (t_bind(newfd, NULL, NULL) < 0 ) ||
         (t_accept(fd, newfd, callptr) < 0) ) {
      t_free((char *)callptr, T_CALL);
      u_set_error(errorp);
      return -1;
   }
   client = (struct sockaddr_in *)(callptr->addr).buf;
   hostptr = gethostbyaddr_r((char *)&(client->sin_addr.s_addr), 4,
      AF_INET, &hostresult, hostbuf, HOSTBUFFERLENGTH, &herror);
   if (hostptr == NULL) {
      errorp -> hosterr = GETHOSTBYADDRERROR;
      return -1;
   }
   strcpy(hostn, hostresult.h_name);
   t_free((char *)callptr, T_CALL);
   return newfd;
}

/*
 *                          u_connect_r
 * Iniciar la comunicación con un servidor remoto.
 *
 * parámetros:
 *    port = puerto bien conocido en el servidor remoto
 *    inept = nombre de Internet de la máquina remota
 * devuelve: descriptor de archivo para comunicación o -1 si hay error
 */
int u_connect_r(unsigned short port, char *inetp, uerror_t *errorp)
{
   struct t_info info;
   struct t_call *callptr;
   struct sockaddr_in server;
   struct hostent *hp;
   struct hostent hostresult;
   int fd;
   int trynum;
   unsigned int slptime;
   int herror;
```

```
char hostbuf[HOSTBUFFERLENGTH];

fd = t_open("/dev/tcp", O_RDWR, &info);
if (fd < 0) {
    u_set_error(errorp);
    return -1;
}

/* Crear direcciones TCP para comunicarse con el puerto */
memset(&server, 0, (size_t)sizeof(server));
server.sin_family = AF_INET;          /* familia de direcciones de Internet */
                                /* convertir nombre en dirección de Internet */
hp = gethostbyname_r(inetp, &hostresult, hostbuf, HOSTBUFFERLENGTH,
                     &herror);
if (hp == NULL) {
    errorp -> hosterr = GETHOSTBYNAMEERROR;
    return -1;
}
                                    /* copiar dirección de Internet en buffer */
memcpy(&(server.sin_addr.s_addr), hostresult.h_addr_list[0],
   (size_t)hostresult.h_length);
server.sin_port = htons(port);

                           /* establecer buffer con información del destino */
if ( (t_bind(fd, NULL, NULL) < 0) ||
     ((callptr =
         (struct t_call *)t_alloc(fd, T_CALL, T_ADDR)) == NULL) ) {
   u_set_error(errorp);
   return -1;
}

callptr -> addr.maxlen = sizeof(server);
callptr -> addr.len = sizeof(server);
callptr -> addr.buf = (char *)&server;
callptr -> opt.len = 0;
callptr -> udata.len = 0;

                                                /* Reintentar si esto fracasa */
for (trynum = 0, slptime = 1;
    (trynum < RETRIES) &&
       (t_connect(fd, callptr, NULL) < 0);
    trynum++, slptime = 2*slptime) {
    t_rcvdis(fd, NULL);
   sleep(slptime);
}

callptr -> addr.buf = NULL;
t_free((char *)callptr, T_CALL);
if (trynum >= RETRIES) {
```

```
        u_set_error(errorp);
        return -1;
    }
    return fd;
}

/*
 *                              u_close_r
 * Cerrar comunicación para el descriptor de archivo TLI dado
 * parámetro:
 *       fd = descriptor de archivo de la conexión TLI por cerrar
 * devuelve:
 *       un valor negativo indica que hubo un error
 */
int u_close_r(int fd, uerror_t *errorp)
{
    int retval;
    retval = t_close(fd);
    if (retval < 0)
        u_set_error(errorp);
    return retval;
}

/*
 *                              u_read_r
 *
 * Recuperar información de un descriptor de archivo abierto con u_open.
 *
 * parámetros:
 *     fd = descriptor de archivo de TLI
 *     buf = buffer para la salida
 *     nbyte = número de bytes por recuperar
 * devuelve:
 *     un valor negativo indica que hubo un error
 */

ssize_t u_read_r(int fd, void *buf, size_t nbyte, uerror_t *errorp)
{
    int rcvflag;
    ssize_t retval;

    rcvflag = 0;
    while ( ( (retval=
            (ssize_t)t_rcv(fd, buf, (unsigned)nbyte, &rcvflag)) == -1) &&
        (t_errno == TSYSERR) && (errno == EINTR) )
        ;                                          /* se reinicia la lectura */
    if (retval < 0)
        u_set_error(errorp);
    return retval;
```

```
}

/*
 *                          u_write_r
 *
 * Enviar información a un descriptor de archivo abierto con u_open.
 *
 * parámetros:
 *       fd = descriptor de archivo de TLI
 *       buf = buffer para la salida
 *       nbyte = número de bytes por enviar
 * devuelve:
 *      un valor negativo indica que hubo un error
 */
ssize_t u_write_r(int fd, void *buf, size_t nbyte, uerror_t *errorp)
{
   ssize_t retval;
   while ( ( (retval =
           (ssize_t)t_snd(fd, buf, (unsigned)nbyte, 0)) == -1) &&
      (t_errno == TSYSERR) && (errno == EINTR) )
      ;                                          /* se reinicia la escritura */
   if (retval < 0)
       u_set_error(errorp);
   return retval;
}

/*
 *                          u_error_r
 *  Exhibir mensaje de error como hacen perror o t_error.
 *
 *  parámetro:
 *     s = cadena que precederá al mensaje de error del sistema
 *  devuelve:  0
 *
 *  algoritmo: Primero se ve si hay un error en el campo hosterr
 *             y se imprime un mensaje si este error está establecido.
 *             Si no, se verifica tli_error. Si éste es TSYSERR, se exhibe
 *             el error de sistema usando strerror; si es otro,
 *             se imprime el error de TLI desde t_errlist.
 */
void u_error_r(char *s, uerror_t error)
{
   if (error.hosterr == GETHOSTBYNAMEERROR)
      fprintf(stderr,"%s: error in getting name of remote host\n", s);
   else if (error.hosterr == GETHOSTBYADDRERROR)
      fprintf(stderr,"%s: error converting host name to address\n", s);
   else if (error.tli_error == TSYSERR)
      fprintf(stderr,"%s: %s\n", s, strerror(error.syserr));
   else
```

```
        fprintf(stderr,"%s: %s\n", s, t_errlist[error.tli_error]);
}

/*
 *                              u_sync_r
 *
 * Esta función debe invocarse después de exec o dup para
 * anexar un descriptor de archivo abierto a las funciones TLI.
 *
 * parámetro:
 *       fd = descriptor del archivo por anexar
 * devuelve: -1 si hay error y algo distinto si tiene éxito
 */
int u_sync_r(int fd, uerror_t *errorp)
{
   int syncret;
   syncret = t_sync(fd);
   if (syncret == -1)
      u_set_error(errorp);
   return syncret;
}
```

Programa B.12

Bibliografía

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A new kernel foundation for UNIX development," *Proc. Summer 1986 USENIX Conference*, 1986, pp. 93–112.

[2] ANS X3.159-1989, Programming Language C.

[3] T. Anderson, B. Bershad, E. Lazowska and H. Levy, "Scheduler activations: Efficient kernel support for the user-level management of parallelism," *Proc. 13th ACM Symposium on Operating Systems Principles*, 1991, pp. 95–109.

[4] L. J. Arthur, *UNIX Shell Programming*, John Wiley & Sons, 1990.

[5] H. Attiya, M. Snir and M. K. Warmuth, "Computing on an anonymous ring," *Journal of the ACM*, vol. 35, no. 4, 1988, pp. 845–875.

[6] M. J. Bach, *The Design of the UNIX Operating System*, Prentice Hall, 1986.

[7] A. Beguelin, J. Dongarra, A. Geist and V. Sunderam, "Visualization and debugging in a heterogeneous environment," *Computer*, vol. 26, no. 6, 1993, pp. 88–95.

[8] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. on Computer Systems*, vol. 2, no. 1, 1984, pp. 39–59.

[9] A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter, "Distribution and abstract types in Emerald," *IEEE Trans. Software Engineering*, vol. SE-13, no. 1, 1987, pp. 65–76.

[10] D. Black, "Scheduling support for concurrency and parallelism in the Mach operating system," *IEEE Computer*, vol. 23, no. 5, 1990, pp. 35–43.

[11] J. Bloomer, *Power Programming with RPC*, O'Reilly & Associates, 1992.

[12] M. I. Bolsky and D. G. Korn, *The New KornShell Command and Programming Language*, 2nd ed., Prentice Hall, 1995.

[13] G. Brassard and P. Bratley, *Algorithmics: Theory & Practice*, Prentice Hall, 1988.

[14] Blackboard Technology Group, Inc. Amherst, Mass. as advertised in *Communications of the ACM*, December 1992.

[15] D. Cameron and B. Rosenblatt, *Learning GNU Emacs*, O'Reilly & Associates, 1991.

[16] N. Carriero and D. Gelernter, "How to write parallel programs: A guide to the perplexed," *ACM Computing Surveys*, Sept. 1989, pp. 323-357.

[17] N. Carriero and D. Gelernter, "Linda in context," *Communications of ACM*, vol. 32, no. 4, 1989, pp. 444–458.

[18] N. J. Carriero, D. Gelernter, T. G. Mattson and A. H. Sherman, "The Linda alternative to message-passing systems," *Parallel Computing*, vol. 20, 1994, pp. 633–655.

[19] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes," *Commun. of ACM*, vol. 22, no. 5, 1979, pp. 281–283.

[20] C. Comaford, "Viewpoint: Why people don't succeed with client/server," *IEEE Spectrum*, Jan. 1995, pp. 46–47.

[21] G. Coulouris, J. Dollimore and T. Kindberg, *Distributed Systems: Concept and Design*, 2nd ed., Addison-Wesley, 1994.

[22] D. Curry, *Using C on the UNIX System*, O'Reilly & Associates, 1989.

[23] I. F. Darwin, *Checking C Programs with* `lint`, O'Reilly & Associates, 1988.

[24] P. Dasgupta, R. J. LeBlanc, Jr., M. Ahamad and U. Ramachandran, "The Clouds distributed operating system," *IEEE Computer*, vol. 24, no. 11, 1991, pp. 34–44.

[25] E. W. Dijkstra, "Co-operating sequential srocesses," in *Programming Languages*, F. Genuys(ed.), Academic Press, 1968, pp. 43–112.

[26] L. Dowdy and C. Lowery, *P.S. to Operating Systems*, Prentice Hall, 1993.

[27] R. P. Draves, B. N. Bershad, R. F. Rashid and R. W. Dean, "Using continuations to implement thread managment and communication in operating systems," *Proc. 13th Symp. on Operating Systems Principles*, 1991, pp. 122–136.

[28] P. DuBois, *Using* `csh` *and* `tsch`, O'Reilly & Associates, 1995.

[29] G. A. Geist and V. S. Sunderam, "Experiences with network-based concurrent computing on the PVM system," *Concurrency: Practice and Experience*, vol. 4, no. 4, 1992, pp. 392–311.

[30] B. Furht, D. Kalra, F. L. Kitson, A. A. Rodriguez and W. E. Wall, "Design issues for interactive television systems," *IEEE Computer*, vol. 28, no. 5, 1995, pp. 25–39.

[31] B. Gallmeister, *POSIX.4: Programming for the Real World*, O'Reilly & Associates, 1995.

[32] D. Gelernter, "Generative communication in Linda," *ACM Trans. Prog. Lang Systems*, vol. 7, no. 1, 1985, pp. 80–112.

[33] D. Gelernter, "Getting the job done," *Byte*, vol. 13, no. 11, 1988, pp. 301–308.

[34] D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. V. Rangan and L. A. Rowe, "Multimedia storage servers: A tutorial," *IEEE Computer*, vol. 28, no. 5, 1995, pp. 40–49.

[35] *Lord of the flies*, a novel by William Golding, Faber and Faber, London, 1954.

[36] R. Govindan and D. P. Anderson, "Scheduling and IPC mechanisms for continuous media," *Proc. 13th ACM Symposium on Operating Systems Principles*, 1991 pp. 68–80.

[37] S. P. Harbison and G. L. Steele, Jr., *C: A Reference Manual*, 4th ed., Prentice Hall, 1995.

[38] G. Held, *Data Communications Networking Devices*, 2nd ed., John Wiley & Sons, 1989.

[39] B. A. Huberman and T. Hogg, "The behavior of computational ecologies," *The Ecology of Computation*, ed. B. A. Huberman, North-Holland, Amsterdam, 1988, pp. 71–115.

[40] B. A. Huberman, "The performance of cooperative processes," *Physica*, vol. 42D, 1990, pp. 38–47.

[41] ISO/IEC 9899: 1990, Programming Languages—C.

[42] A. Itai and M. Rodeh, "Symmetry breaking in distributive networks," *Proc. Twenty-Second Annual IEEE Symposium on the Foundations of Computer Science*, 1981, pp. 150–158.

[43] A. Jones and A. Hopper, "Handling audio and video streams in a distributed environment," *Proc. Fourteenth ACM Symposium on Operating Systems Principles*, 1993, pp. 231–243.

[44] S. Kleiman, D. Shah and B. Smaalders, *Programming with Threads*, Prentice Hall, forthcoming.

[45] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin-Cummings, 1994.

[46] W. Leler, "Linda meets UNIX," *Computer*, Feb. 1990, pp. 43–54.

[47] L. Lamb, *Learning the* `vi` *Editor*, 5th ed., O'Reilly & Associates, 1990.

[48] B. Marsh, M. Scott, T. LeBlanc and E. Markatos, "First-class user-level threads," *Proc. Thirteenth ACM Symposium on Operating Systems Principles*, 1991, pp. 110–121.

[49] K. Nahrstedt and R. Steinmetz, "Resource management in networked multimedia systems," *IEEE Computer*, vol. 28, no. 5, 1995, pp. 52–63.

[50] E. Nemeth, G. Snyder, S. Seebass and T. R. Hein, *UNIX System Administration Handbook*, 2nd ed., Prentice Hall, 1995.

[51] S. Nishio, K. F. Li and E. G. Manning, "A resilient mutual exclusion algorithm for computer networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 3, 1990, pp. 344–355.

[52] *IEEE Standard for Information Technology. Portable Operating System Interface (POSIX). Part 1: System Application Program Interface (API) [C Language]*, ISO/IEC 9945-1, IEEE std. 1003.1, 1990.

[53] *IEEE Standard for Information Technology. Portable Operating System Interface (POSIX). Part 1: System Application Program Interface (API)-Amendment 2: Realtime Extension [C Language]*, IEEE std. 1003.1b, 1993.

[54] "Draft Standard for Information Technology Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API)-Amendment 2: Threads Extension [C Language]", P1003.1c, IEEE DS5314, Draft 10, September 1994.

[55] *IEEE Standard for Information Technology Portable Operating System Interface (POSIX) Part 2: Shells and Utilities*, IEEE std. 1003.2, 1992.

[56] H. Ishii and N. Miyake, "Toward an open shared workspace: Computer and video fusion approach of teamworkstation," *Communications of the ACM*, vol. 34, no. 12, 1991, pp. 37–50.

[57] S. Khanna, M. Sebree and John Zolnowsky, "Realtime scheduling in SunOS 5.0," SunSoft Incorporated, reprint, 1992.

[58] B. W. Kernighan and D. M. Ritchie, *The C Programming Language, 2nd Ed.*, Prentice Hall, 1988

[59] S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, *The Design and Implementation of the 4.3 BSD UNIX Operating System*, Addison-Wesley, 1989.

[60] T. G. Lewis, "Where is computing heading?" *Computer*, vol. 27, no. 8, 1994, pp. 59–63.

[61] "Chapter 2: Lightweight Processes," *SunOS 4.1.2 Programming Utilities and Libraries*, Sun Microsystems.

[62] B. Liskow, "Distributed programming in Argus," *Communications of the ACM*, vol. 31, no. 3, 1988, pp. 300–312.

[63] M. R. Macedonia and D. P. Brutzman, "MBone provides audio and video across the Internet," *Computer*, vol. 27, no. 4, 1994, pp. 30–36.

[64] M. Maekawa, A. E. Oldehoeft and R. R. Oldehoeft, *Operating Systems: Advanced Concepts*, Benjamin/Cummings, 1987.

[65] *Webster's Third New International Dictionary of the English Language Unabridged*, Merriam-Webster Inc, Springfield, Mass., 1981.

[66] "SunOS 5.0 multithread architecture," *Sun Microsystems White Paper*, 1991.

[67] A. Oram and S. Talbott, *Managing Projects with* `make`, 2nd ed., O'Reilly & Associates, 1991.

[68] *UNIX in a Nutshell: A Desktop Quick Reference for System V*, O'Reilly & Associates, 1987.

[69] R. Perlman, *Interconnections: Bridges and Routers*, Addison-Wesley, 1992.

[70] P. J. Plauger, *The Standard C Library*, Prentice Hall, 1992.

[71] S. A. Rago, *UNIX System V Network Programming*, Addison-Wesley, 1993.

[72] K. A. Robbins, N. R. Wagner and D. J. Wenzel, "Virtual rings: An introduction to concurrency," `SIGCSE`, vol. 21, 1989, pp. 23–28.

[73] A. A. Rodriguez and L. A. Rowe, "Multimedia systems and applications," *IEEE Computer*, vol. 28, no. 5, 1995, pp. 20–22.

[74] B. Rosenblatt, *Learning the Korn Shell*, O'Reilly & Associates, 1993.

[75] R. Rybacki, K. Robbins and S. Robbins, "Ethercom: A study of audio processes and synchronization," *Proc. Twenty-Fourth SIGCSE Technical Symp. on Computer Science Education*, 1993, pp. 218–222.

[76] R. Sandberg, "The SUN network file system: Design, implementation, and experience," Sun Microsystems White Paper, 1989.

[77] C. Schimmel, *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers*, Addison-Wesley, 1994.

[78] S. Shrivastava, G. N. Dixon and G. D. Parrington, "An overview of the Arjuna distributed programming system," *IEEE Software*, January 1991, pp. 66–73.

[79] R. W. Sevick, "Viewpoint: For commercial multiprocessing the choice is SMP," *IEEE Spectrum*, January 1995, pp. 50.

[80] A. Silberschatz and P. B. Galvin, *Operating Systems Concepts*, 4th ed., Addison-Wesley Publishing, 1994.

[81] M. G. Sobell, *UNIX System V: A Practical Guide*, 3rd ed., Benjamin/Cummings. 1995.

[82] M. G. Sobell, *A Practical Guide to the UNIX System*, 3rd ed., Benjamin/Cummings, 1995.

[83] *Sun OS5.3 Writing Device Drivers*, SunSoft Incorporated, 1993.

[84] W. Stallings, *Local and Metropolitan Area Networks*, 4th ed., Macmillan, 1993.

[85] W. R. Stevens, *UNIX Network Programming*, Prentice Hall, 1990.

[86] W. R. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992.

[87] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *J. of Concurrency: Practice and Experience*, vol. 2, no. 4, 1990, pp. 315–339.

[88] S. Talbott, *Managing Projects with* `make`, O'Reilly & Associates, 1991.

[89] A. S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice Hall, 1987.

[90] A. S. Tanenbaum, *Computer Networks*, Prentice Hall, 1989.

[91] A. S. Tanenbaum, R. van Renesse, H. van Staveren, G. Sharp, S. Mullender, J. Jansen and G. van Rossum, "Experiences with the Amoeba distributed operating system," *Communications of the ACM*, vol. 33, no. 12, 1990, pp. 46–63.

[92] A. S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 1992.

[93] A. S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, 1995.

[94] D. B. Terry and D. C. Swinehart, "Managing stored voice in the Etherphone system," *ACM Transactions on Computer Systems*, vol. 6, 1988, pp. 3–27.

[95] G. Varghese and T. Lauck, "Hashed and hierarchical timing wheels: Data structures for efficient implementation of a Timer Facility," *Proc. Eleventh ACM Symp. on OS Principles*, 1987, pp. 25–38.

[96] *X/Open CAE Specification: System Interface Definitions*, Issue 4, Version 2, X/Open Company Ltd., 1994.

[97] *X/Open CAE Specification: System Interfaces and Headers*, Issue 4, Version 2, X/Open Company Ltd., 1994.

[98] *X/Open CAE Specification: Commands and Utilities*, Issue 4, Version 2, X/Open Company Ltd., 1994.

[99] *X/Open CAE Specification: Networking Services*, Issue 4, X/Open Company Ltd., 1994.

[100] *Solaris Multithreaded Programming Guide*, SunSoft Incorporated, 1995.

Índice

Las entradas se refieren a números de página en el texto. Se utilizan **negritas** para las referencias más importantes.

Esta obra se terminó de imprimir en diciembre del 2004
en los talleres de Ultradigital Press, S.A. de C.V.
Centeno 162-3. Col. Granjas Esmeralda
CP 09810, México, D.F.

KAY A. ROBBINS • STEVEN ROBBINS

UNIX® PROGRAMACIÓN PRÁCTICA

Guía para la Concurrencia, la Comunicación y los Multihilos

Fundamentación del software que explota todo el poder de los sistemas UNIX de hoy.

Para utilizar completamente las plataformas actuales, el nuevo software depende cada vez más de las complejas técnicas de comunicación, concurrencia y multihilos. ***UNIX Programación Práctica*** es una guía clara, comprensible y actualizada de estas técnicas esenciales.

Usted, no sólo aprenderá los detalles de la comunicación, la concurrencia y los multihilos, sino que entenderá *por qué* estas técnicas son tan importantes y *dónde* y *cómo* usarlas. Los autores presentan cientos de ejemplos breves, proyectos de muestra más extensos y técnicas de trabajo para ayudarle a encontrar el sentido de éstos y otros temas complejos:

- RPC, comunicación por redes y el modelo cliente–servidor
- Señales, incluyendo las nuevas señales de tiempo real POSIX
- STREAMS, sockets y TLI
- Hilos y semáforos
- Nuevos estándares POSIX y Spec 1170

Usted aprenderá a usar la comunicación, la concurrencia y los multihilos en aplicaciones realistas. Dominará el difícil arte de *poner a prueba* los programas concurrentes. ***UNIX Programación Práctica*** incluso ofrece bibliotecas simplificadas que puede usar en sus propias aplicaciones de comunicación en red.

En un mundo de redes, de sistemas multiprocesadores y de aplicaciones cliente–servidor, las técnicas que aquí se estudian se han vuelto críticas ante el desarrollo de software UNIX. Este libro no sólo le ayudará a dominar dichas técnicas, sino que también servirá como una referencia excelente en los años por venir.

KAY y STEVEN ROBBINS hicieron su doctorado en el MIT y son miembros facultativos de la División de Computación de la University of Texas, en San Antonio.

ISBN 968-880-959-4

9 789688 809594 90000

Pearson Educación