Sistemas operativos

MARTÍN SILVA



Alfaomega

Índice general

Mensaje del editor IX

Agradecimientos XI

Sobre el autor XIII

Prólogo XXV

1. Introducción 1

1.1. Funciones y objetivos de los sistemas operativos 1

1.1.1. Funciones 3

1.2. Estructuras de los sistemas operativos 3

1.2.1. Gestión de procesos 3

1.2.2. Gestión de la memoria principal 4

1.2.3. Gestión de archivos 4

1.2.4. Gestión de dispositivos de entrada y salida 5

1.2.5. Servicios 5

1.2.6. Tipos de sistemas operativos 5

1.2.6.1. El núcleo del sistema operativo 8

1.3. Estructura y funcionamiento de la computadora 9

1.3.1. Componentes de una computadora 10

1.3.1.1. Dinámica de una operación de entrada y salida 10

1.3.1.2. El vector de interrupciones 11

1.3.1.3. Hardware de las interrupciones 11

1.3.1.4. Tratamiento de la interrupción 12

1.3.1.5. Tratamiento diferido . . . 13
1.3.1.6. Clock tick . . . 13
1.3.1.7. Llamadas al sistema . . . 14
1.3.2. El proceso de arranque . . . 16
1.4. Evolución de los sistemas operativos . . . 17
1.4.1. La API de Windows . . . 27
1.5. Trabajos prácticos . . . 28
1.5.1. Práctica con Linux . . . 28
1.5.1.1. procfs . . . 29
1.5.1.2. Llamadas a sistema . . . 30
1.5.1.3. Apagando Linux . . . 31
1.5.2. Práctica con Windows . . . 31
1.5.2.1. ProcFeatures . . . 32
1.5.2.1.1. Ejemplo 1 . . . 32
1.5.2.1.2. Ejemplo 2 . . . 32
1.5.2.2. CoreInfo . . . 33
1.5.2.2.1. Ejemplo 1 . . . 33
1.5.2.2.2. Ejemplo 2 . . . 33
1.5.2.3. Llamadas a sistema . . . 33
1.5.2.4. Administrador de tareas . . . 34
1.5.3. Una analogía . . . 34

2. Procesos **35**
2.1. Estados de un proceso . . . 37
2.1.1. Bloque de control de procesos . . . 39
2.1.2. Creación y terminación de procesos . . . 40
2.1.2.1. Jerarquía de procesos . . . 41
2.1.3. Planificador y activador . . . 41
2.1.4. Cambio de contexto . . . 42
2.2. Procesos ligeros, hilos o threads . . . 43
2.2.1. Motivaciones para el uso de hilos . . . 45
2.2.2. Modelos de organización . . . 48
2.2.3. Aspectos del diseño de hilos . . . 49
2.2.3.1. Creación de los hilos . . . 49
2.2.3.2. Terminación de los hilos . . . 49
2.2.3.3. Sincronización de los hilos . . . 50
2.2.3.4. Planificación de los hilos . . . 50
2.2.3.5. Manejo de señales . . . 51
2.2.4. Implementación . . . 52
2.2.4.1. Ventajas y desventajas de los dos enfoques . . . 53
2.2.5. Fibras . . . 54
2.3. Comunicación . . . 55
2.3.1. Procesos independientes . . . 55
2.3.2. Procesos cooperantes . . . 55
2.3.3. Procesos concurrentes . . . 56

2.3.4. Paradigmas de comunicación entre procesos 58
2.3.4.1. Memoria compartida 58
2.3.4.2. Pase de mensajes 58
2.3.4.3. Puerto 58
2.3.4.4. Señales 60
2.3.4.5. Temporizadores 60
2.3.4.6. Socket 60
2.3.4.7. Tubería 61
2.3.4.8. Cola de mensajes 61
2.3.4.9. Archivos 61
2.3.4.10. Archivos proyectados en memoria 61
2.3.5. Excepciones 62
2.4. Trabajos prácticos 62
2.4.1. Práctica con Linux 62
2.4.1.1. Estados de los procesos 62
2.4.1.2. Obtener el PID 63
2.4.1.3. Crear procesos con fork() 64
2.4.2. Práctica con Windows 65
2.4.2.1. tlist 65
2.4.2.1.1. Ejemplo 1 66
2.4.2.1.2. Ejemplo 2 66
2.4.2.2. Información de los procesos con el Administrador de Tareas 67

3. Memoria **69**
3.1. Funciones y operaciones del administrador de memoria 70
3.2. Modelo de memoria de un proceso 71
3.2.1. Fases en la generación de un ejecutable 72
3.2.2. Formato del ejecutable 74
3.2.2.1. El formato COFF 75
3.2.2.2. El formato ELF 75
3.2.2.3. El formato PE 76
3.2.3. Mapa de memoria de un proceso 77
3.3. Diferentes esquemas de administración 79
3.3.1. Mono programación 79
3.3.1.1. Partición absoluta única 79
3.3.1.2. Partición reubicable única 80
3.3.1.3. Superposiciones 81
3.3.2. Multiprogramación 82
3.3.2.1. Múltiples particiones fijas 82
3.3.2.2. Múltiples particiones variables 84
3.3.2.2.1. Algoritmos de selección de la partición: 85
3.3.2.2.2. Cola de procesos por partición: 86
3.3.2.2.3. Cola única: 86
3.3.2.3. Sistema de compañeras 86
3.3.2.3.1. Slab Allocator 86

3.3.2.4. Reubicación . . . 87
3.3.3. Paginación simple . . . 88
3.3.4. Segmentación simple . . . 90
3.3.5. Segmentación con paginación . . . 92
3.3.6. Tablas de páginas y de segmentos . . . 92
3.3.7. Memoria asociativa . . . 93
3.3.8. Tabla de páginas invertida . . . 94
3.3.9. Ventaja adicional del paginado: las páginas compartidas . . . 95
3.3.10. Intercambio . . . 96
3.4. Memoria virtual . . . 97
3.4.1. Paginación por demanda . . . 99
3.4.1.0.1. Ejemplo . . . 100
3.4.2. Localidad de referencia . . . 101
3.4.3. Traba de páginas . . . 102
3.4.4. Tamaño de la página . . . 103
3.4.5. Algoritmos de reemplazo de páginas . . . 103
3.4.6. El rendimiento de los algoritmos . . . 104
3.4.7. Políticas de asignación . . . 105
3.4.7.0.2. Ejemplo . . . 106
3.4.7.1. Hiperpaginación . . . 107
3.4.8. Conjunto de trabajo . . . 107
3.4.9. Prepaginado . . . 107
3.4.10. Segmentación . . . 108
3.5. Trabajos prácticos . . . 108
3.5.1. Práctica con Linux . . . 108
3.5.1.1. Estadísticas de la memoria virtual en Linux . . . 108
3.5.1.2. El programa mapa.c . . . 110
3.5.1.3. La estructura de ELF . . . 110
3.5.1.4. El archivo /proc/pid/maps . . . 110
3.5.2. Práctica con Windows . . . 112
3.5.2.1. Memoria virtual en Windows . . . 112
3.5.2.2. La estructura de PE . . . 113
3.5.2.3. PhysMem . . . 113
3.5.2.4. CacheSet . . . 114

4. Planificación **115**
4.1. Conceptos . . . 115
4.1.1. Supuestos subyacentes . . . 116
4.1.2. Características . . . 116
4.1.3. Contexto . . . 118
4.1.4. Desalojo . . . 118
4.1.5. Colas . . . 120
4.1.6. Criterios . . . 121
4.1.7. Notación de Landau . . . 122
4.2. Tipos de planificación . . . 122

4.3. Algoritmos de planificación 124
4.3.1. FCFS (First-Come, First-Serve) 124
4.3.1.1. El efecto convoy 124
4.3.2. SJF (Shortest-Job-First) 124
4.3.2.1. SRPT (Shortest-Remaining-Processing-Time) 125
4.3.3. Por prioridad 125
4.3.4. RR (Round-Robin) 126
4.3.5. Colas multiniveles 126
4.3.5.1. MLQ 127
4.3.5.2. MLFQ 127
4.4. Planificación en POSIX 127
4.5. Planificación en Linux 129
4.5.1. Primeras versiones 130
4.5.1.1. Version 0.96 130
4.5.1.2. Versión 1.2 130
4.5.1.3. Version 2.0.33 131
4.5.1.4. Versión 2.2 131
4.5.1.5. Versión 2.4 132
4.5.1.6. Versión 2.6.8.1 132
4.5.2. Versiones recientes 133
4.5.2.1. Versión 2.6.21 135
4.5.2.2. Versión 2.6.23 135
4.6. Planificación en Windows 136
4.6.1. Niveles de prioridad 139
4.6.2. Base de datos del activador 140
4.7. Práctica con Linux 142
4.7.1. Cambio de contexto 143
4.7.2. Una analogía 144
5. Sincronización 146
5.1. Antecedentes 147
5.2. Alternativas de sincronización 147
5.2.1. Sección crítica 148
5.2.2. Exclusión mutua 149
5.2.3. Soluciones hardware 149
5.2.3.1. Inhabilitación de las interrupciones 150
5.2.3.2. Instrucciones especiales de máquina 150
5.2.3.2.1. TSL 151
5.2.3.2.2. CAS 151
5.2.3.2.3. Fetch-and-add 151
5.2.3.2.4. Read-modify-write 151
5.2.3.2.5. Load-link/store-conditional 151
5.2.3.2.6. Soluciones libres de espera 152
5.2.4. Soluciones software 152
5.2.4.1. El problema de la sección crítica 153

5.2.4.1.1. Algoritmo de Dekker . . . 153
5.2.4.1.2. Algoritmo de Peterson . . . 153
5.2.4.1.3. Requisitos de una solución . . . 154
5.2.4.2. Variables mutex . . . 155
5.2.4.3. Variables de condición . . . 156
5.2.4.4. Spinlock . . . 156
5.3. Problemas clásicos . . . 157
5.3.1. Los filósofos comensales . . . 158
5.3.2. El peluquero dormilón . . . 158
5.3.3. Problema del productor-consumidor . . . 159
5.3.4. Problema de los lectores-escritores . . . 159
5.3.5. Comunicación cliente-servidor . . . 161
5.4. Semáforos . . . 162
5.5. Monitores . . . 162
5.5.1. Sintaxis de un monitor . . . 163
5.6. Sincronización en pase de mensajes . . . 165
5.6.1. Características de la sincronización . . . 165
5.6.2. Memoria intermedia o *buffer* . . . 166
5.6.2.1. Buffer nulo . . . 167
5.6.2.2. Buffer de mensaje único . . . 167
5.6.2.3. Buffer de capacidad ilimitada . . . 168
5.6.2.4. Buffer de capacidad limitada . . . 169
5.6.3. Sincronización en tuberías . . . 169
5.6.4. Sincronización mediante señales . . . 170
5.7. Bloqueo mutuo . . . 170
5.7.1. Condiciones para que se cumpla el interbloqueo . . . 171
5.7.2. Grafos de asignación de recursos . . . 171
5.7.3. Métodos para el tratamiento del bloqueo mutuo . . . 171
5.7.4. Prevención . . . 172
5.7.5. Cómo evitar el bloqueo mutuo . . . 173
5.7.5.1. El algoritmo del banquero . . . 173
5.7.6. Detección del interbloqueo . . . 174
5.7.7. Recuperación del interbloqueo . . . 175
5.7.8. Algoritmo del avestruz . . . 176
5.8. Sincronización en POSIX . . . 176
5.8.1. Señales . . . 177
5.8.2. Mutex . . . 178
5.8.2.1. Mutex no bloqueante . . . 178
5.8.3. Semáforos para hilos . . . 179
5.8.4. Semáforos para procesos . . . 179
5.8.5. Variables de condición . . . 180
5.8.6. Memoria compartida . . . 181
5.8.6.1. Proyectar un archivo compartido . . . 182
5.8.6.2. Acceso compartido a un archivo . . . 183
5.9. Sincronización en Linux . . . 183

5.9.1. Operaciones atómicas
5.9.2. Spinlock . . . 183
5.9.2.1. Full spinlock . . . 184
5.9.2.2. Lectores y escritores . . . 184
5.9.3. Mutex . . . 184
5.10. Sincronización en Windows . . . 185
5.10.1. Sincronización con IRQL alto . . . 185
5.10.1.1. Spinlocks . . . 185
5.10.2. Sincronización con IRQL bajo . . . 186
5.10.3. Llamada a Procedimiento Local . . . 187
5.10.4. Mailslots . . . 188
5.11. Trabajos prácticos . . . 190
5.11.1. Tuberías . . . 191
5.11.2. Otro ejemplo: . . . 191
5.12. Autoevaluación . . . 192
. . . 193

6. Almacenamiento **195**
6.1. Conceptos . . . 196
6.1.1. Archivos . . . 197
6.1.1.1. Tipos . . . 197
6.1.1.2. Atributos . . . 198
6.1.1.3. Operaciones . . . 199
6.1.2. Particiones . . . 199
6.1.3. Nombres . . . 200
6.1.4. Sistema de archivos . . . 201
6.1.4.1. Directorio . . . 201
6.1.4.1.1. Directorio de un nivel . . . 202
6.1.4.1.2. Directorio de dos niveles . . . 202
6.1.4.1.3. Estructura de árbol . . . 203
6.1.4.1.4. Estructura de grafo . . . 203
6.1.4.1.5. Entradas de directorio . . . 204
6.1.4.2. Designación de un archivo . . . 206
6.1.4.2.1. Directorio raíz por proceso . . . 206
6.1.4.2.2. Otros tipos de objetos . . . 207
6.1.4.3. Journaling . . . 207
6.1.5. Arquitectura del software . . . 209
6.1.6. Estructuras asociadas . . . 211
6.1.7. Organización de los archivos . . . 212
6.1.7.1. Gestión del espacio libre en disco . . . 213
6.1.7.2. Técnicas de asignación . . . 216
. . . 216
6.2. Implementaciones . . . 216
6.2.1. Sistemas de archivos en Windows . . . 220
6.2.1.1. FAT - File Allocation Table . . . 222
6.2.1.2. NTFS
6.2.2. Sistema de archivos de UNIX

6.2.3. Sistemas de archivos en Linux 225
6.2.3.1. Second Extended File System 225
6.2.3.2. Virtual File System 226
6.2.3.3. Third Extended File System 227
6.2.3.4. ReiserFS 228
6.2.3.5. Fourth Extended File System 229
6.2.3.6. Sistemas de archivos sin almacenamiento 230
6.2.3.6.1. procfs 230
6.2.3.6.2. debugfs 231
6.2.3.6.3. sysfs 231
6.2.3.6.4. ramfs 232
6.3. Trabajos prácticos 232
6.3.1. Particiones y sistemas de archivos 234
6.3.2. Directorios, archivos y enlaces 236
6.3.3. Permisos 237

7. Sistemas de entrada y salida 237
7.1. Dispositivos de entrada y salida 238
7.2. Organización de las funciones de entrada y salida 241
7.3. Aspectos del diseño 241
7.3.1. Objetivos 241
7.3.2. Mecanismo y política 241
7.3.3. Estructura lógica de las funciones 242
7.4. Subsistema de entrada y salida del núcleo 243
7.4.1. Planificación de la entrada y salida 243
7.4.2. Almacenamiento intermedio 244
7.4.2.1. Memoria intermedia nula 244
7.4.2.2. Memoria intermedia sencilla 245
7.4.2.3. Memoria intermedia doble 245
7.4.2.4. Memoria intermedia circular 246
7.4.3. Caché 247
7.4.4. Spool 247
7.4.5. Manejo de errores 247
7.4.6. Estructuras de datos 248
7.5. Planificación del acceso a discos 248
7.5.1. Estructura de un disco 249
7.5.2. Planificación del acceso al disco 249
7.5.3. Distintos tipos de algoritmos de planificación 250
7.5.3.1. FCFS 250
7.5.3.2. SSTF 250
7.5.3.3. SCAN 251
7.5.3.4. C-SCAN 251
7.5.3.5. LOOK 251
7.5.3.6. El ascensor Linus 252
7.5.3.7. El planificador con fecha límite 252

7.5.3.8. El planificador de entrada y salida anticipatorio 253
7.5.3.9. CFQ 253
7.5.4. Manejadores de dispositivos 255
7.6. Ventanas al núcleo 255
7.6.1. Desventajas 257
7.6.2. devfs 260
7.6.3. Conexión en caliente 261
7.6.4. El nuevo modelo de dispositivos de Linux 262
7.6.4.1. Colectores, dispositivos y clases 263
7.6.5. sysfs 263
7.6.6. udev 266
7.6.7. HAL en Linux 267
7.6.7.1. DeviceKit 268
7.6.8. D-Bus 268
7.6.8.1. Netlink sockets 268
7.6.9. Cómo funciona todo junto 269
7.7. Trabajos prácticos 271
7.7.1. ¿Cómo creamos archivos especiales? 273
7.8. Autoevaluación 274

8. Multiprocesamiento **275**
8.1. Múltiples procesadores 275
8.2. Aplicaciones paralelas 277
8.3. Ley de Amdahl 279
8.4. Taxonomía de Flynn 280
8.5. Arquitecturas de múltiples procesadores 283
8.5.1. Procesamiento simétrico y asimétrico 285
8.6. Multiprocesadores simétricos 286
8.6.1. Ventajas 286
8.6.2. Organizaciones 287
8.6.3. Afinidad 289
8.6.4. Granularidad 289
8.6.5. Paralelismo 290
8.6.5.1. Independiente 290
8.6.5.2. De grano grueso y muy grueso 290
8.6.5.3. De grano medio 291
8.6.5.4. De grano fino 291
8.6.6. Serialización de datos 291
8.7. Diseño 292
8.7.1. Procesos o hilos simultáneos y concurrentes 293
8.7.2. Planificación 293
8.7.2.1. Asignación de hilos a procesadores 294
8.7.2.2. Uso de la multiprogramación en procesadores individuales 294
8.7.2.3. Expedición de un proceso 295
8.7.2.4. Planificación de procesos 295

8.7.2.5. Planificación de multiprocesadores 295
8.7.3. Sincronización 299
8.7.3.1. Estructuras de datos locales a la CPU 299
8.7.3.2. Spinlocks 299
8.7.3.3. Semáforos - mutexes 300
8.7.3.4. Traba de lectura rápida 300
8.7.3.5. Leer-Copiar-Actualizar 301
8.7.3.6. Barrera 301
8.7.4. Gestión de la memoria 301
8.7.4.1. Arquitectura de Acceso Uniforme a Memoria (UMA) . . . 302
8.7.4.2. Arquitectura de Acceso No Uniforme a Memoria (NUMA) 302
8.7.4.3. Arquitectura de Memoria de Sólo Caché (COMA) 303
8.7.5. Coherencia y consistencia 303
8.7.5.1. Ordenamiento y consistencia de la memoria 305
8.7.5.2. Modelos de consistencia 306
8.7.5.3. Modelo de consistencia estricta 307
8.7.5.4. Modelo de consistencia secuencial 308
8.7.5.5. Modelo de consistencia causal 309
8.7.5.6. Modelo de consistencia PRAM 309
8.7.5.7. Modelo de consistencia de procesador 310
8.7.5.8. Modelo de orden de almacenamiento total 310
8.7.5.9. Modelo de consistencia débil 311
8.7.5.10. Modelo de consistencia de liberación 312
8.7.5.11. Modelo relajado de consistencia de liberación 313
8.7.5.12. Ordenamiento explícito en memoria 313
8.7.5.13. Las barreras de memoria del hardware 314
8.8. Multiprocesamiento en Windows 314
8.8.1. La base de datos del activador 315
8.8.2. Sincronización en el núcleo 316
8.8.3. Gestión de la memoria 316
8.9. Multiprocesamiento en Linux 318
8.9.1. Gestión de la memoria 319

Bibliografía 321

Introducción

1

El presente libro acerca de los sistemas operativos nació como una breve guía de estudios, para orientar a los alumnos en la lectura de los temas en la abundante bibliografía existente. Por ello, notará que tiene numerosas referencias y, aunque los temas le parezcan recurrentes, es bueno que los lea desde distintos enfoques. Es muy importante que no intente memorizarlos, sino que lea con espíritu crítico, analizando lo que lee y discutiendo los temas con sus compañeros y en clase.

Los sistemas operativos están en constante evolución, como también lo está el hardware; de manera que tal vez algunos conceptos desarrollados ahora caerán en desuso dentro de un tiempo. Si usted memoriza los conceptos actuales como algo definitivo, tal vez dentro de unos años, cuando esté recibido, lo que aprendió le parecerá obsoleto, pero si aprendió razonando sabrá que tiene las bases y verá en esos sistemas operativos una evolución de lo que aprendió. Y, por otra parte, cuando conozca la historia de los sistemas operativos, verá que muchas soluciones ingeniosas aportadas hace muchos años, aún hoy se utilizan.

Los usuarios no interactuamos de forma directa con el sistema operativo, sino con las aplicaciones que ejecutan sobre él. De hecho, es deseable que los usuarios no tengamos acceso a lo que denominamos el «núcleo» del sistema operativo. Entonces nos surge la pregunta ¿cómo puede explicarse a un alumno el funcionamiento de algo a lo que no tiene acceso?. Es como intentar entender el funcionamiento de la mente sin poder tener acceso al cerebro de un ser humano vivo. A través de las distintas Prácticas se intenta mostrar cuál es el estado en un instante dado y cómo reacciona el sistema operativo, porque es una entidad de un dinamismo inimaginable, aún cuando nos da la impresión de estar ocioso.

1.1 Funciones y objetivos de los sistemas operativos

¿Qué es un sistema operativo? Genéricamente hablando, entendemos que un sistema es un conjunto de elementos relacionados entre sí, que funcionan con un fin determinado y, al

ser operativo, entendemos que su fin es el de la operación o conducción del hardware. La evolución del hardware ha sido constante y parece no detenerse: del monoprocesador a los multiprocesadores, memorias cada vez más rápidas, incorporación de nuevos dispositivos, interconexión con otros sistemas, pero ¿qué ha pasado con los sistemas operativos? Los sistemas operativos deberían acompañar esta evolución y ya veremos más adelante que no siempre ha sido así.

Algunos conceptos comunes acerca de qué es un sistema operativo son los de manejador de recursos, programa de control o programa que se ejecuta constantemente en una computadora.

Tanenbaum y Woodhull (1998) lo expresan desde dos puntos de vista complementarios. Por una parte, como máquina extendida o virtual más fácil de entender y programar que el hardware puro[1] y, por otra, como administrador de recursos, entendiendo por «recursos» a los componentes tanto físicos como lógicos: el procesador, la memoria, los discos, los ratones o los archivos.

Un sistema operativo es un programa, o más bien un conjunto de programas o algoritmos, que actúa como intermediario entre el usuario (en su sentido amplio) de un computador y el hardware del computador. Según Silberschatz y Galvin (1997), el propósito de un sistema operativo es crear un entorno en el que el usuario pueda ejecutar programas de forma cómoda y eficiente.

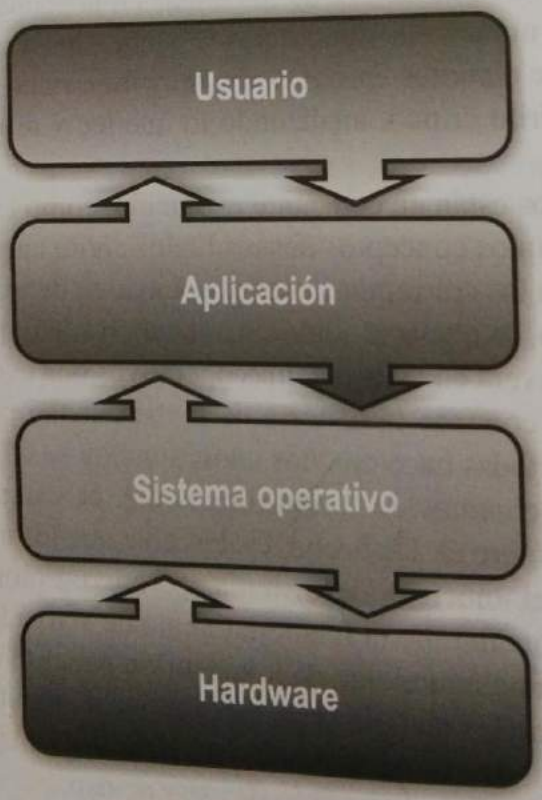


Fig. 1.1 – Ubicación del Sistema Operativo.

Tal vez le sorprenda saber que lo que usted está usando en su computador personal no es un sistema operativo, sino las aplicaciones que sobre él ejecutan. En realidad, usted no

1. Un enfoque desde el punto de vista del programador de sistemas.

usa al sistema operativo en forma directa; es más, en los sistemas modernos se podría llegar a modificar tanto el aspecto que tienen los escritorios y demás aplicaciones –hasta hacerlo parecer tanto a otro–, que sería difícil reconocer cuál es el sistema operativo subyacente.

1.1.1 Funciones

De modo que las funciones primordiales de los sistemas operativos son las de gestionar el hardware, brindando una interfaz con el usuario.

Pero con la evolución de los sistemas de cómputo y la de los sistemas operativos han surgido nuevos requerimientos. Dado que existe una gran diversidad de hardware, es necesario que brinden la capacidad de poder ejecutar sobre esta diversidad, lo que llamamos **portabilidad**. Pero también interoperabilidad e interconectividad (por el mismo motivo). Por otra parte, al haberse extendido tanto entre diversos tipos de usuarios, requerimos que nos provea de un ambiente multitareas y que sea multiusuario, pero esto conlleva requerimientos de seguridad y protección (entre usuarios y desde el exterior), fácil administración, independencia de dispositivo y abstracción del hardware.

Piense por un momento en la siguiente analogía: el sistema operativo es «El Gran Chofer» de un automóvil (hardware) que recibe las indicaciones del pasajero (aplicaciones). Un chofer puede manejar distintos tipos de vehículos (portabilidad) y un mismo vehículo puede ser manejado por distintos choferes, contrariamente a la creencia popular de que el sistema operativo «viene» con la computadora.

1.2 Estructuras de los sistemas operativos

El sistema operativo provee una **máquina virtual** a las aplicaciones, es decir, las aplicaciones no interactúan con el hardware real sino con una abstracción que les presenta el sistema operativo. Las **llamadas a sistema** (*system call*) son el mecanismo utilizado por las aplicaciones para solicitarle un servicio al sistema operativo, son la interfaz entre un proceso y el sistema operativo. La llamadas a sistema implementadas por el sistema operativo expanden las capacidades del conjunto de instrucciones provistas por el hardware puro. Las llamadas a sistema soportan nuevas abstracciones tales como «procesos» y «sistema de archivos».

Además de proveer la interfaz de llamadas al sistema, el sistema operativo tiene la responsabilidad de administrar los recursos de hardware. Las aplicaciones no pueden acceder a los dispositivos de entrada y salida o ejecutar instrucciones privilegiadas del procesador. Es el sistema operativo el que realiza estas operaciones en nombre de los programas de aplicación. Para hacerlo, intenta utilizar de forma eficiente los recursos que tiene disponibles y proteger la integridad de las aplicaciones que deben compartir estos recursos. Ampliaremos este tema en el apartado 1.3.1.7.

Las tareas realizadas por un sistema operativo se pueden dividir en distintas áreas que veremos a continuación.

1.2.1 Gestión de procesos

Un proceso es un programa en ejecución. Asociado con un proceso está su código, sus datos, un conjunto de recursos asignados a él y uno o más «flujos» de ejecución de su código.

El sistema operativo provee llamadas a sistema para la gestión de procesos y debe administrar la asignación de recursos a ellos. Si pueden existir múltiples procesos simultáneamente, el sistema operativo debe ser capaz de proveer a cada uno un ambiente de máquina virtual en el cual pueda ejecutar.

1.2.2 Gestión de la memoria principal

La memoria principal es crucial para el funcionamiento de un sistema de computación moderno; es una matriz grande de palabras o bytes, cuyo tamaño va desde cientos de miles hasta miles de millones. Cada palabra o byte tiene su propia dirección. La memoria principal es un depósito de datos a los que se puede acceder rápidamente y que son compartidos por la CPU y los dispositivos de entrada y salida. El procesador central lee instrucciones de la memoria principal, durante el ciclo de obtención de instrucciones, y lee y escribe datos de la misma durante el ciclo de obtención de datos. Como mínimo la memoria debe ser compartida por un programa de aplicación y por el sistema operativo. En sistemas más sofisticados, la memoria puede estar compartida por varios procesos. El sistema operativo debe gestionar la asignación de memoria a los procesos y controlar el hardware de administración de esta, que determina a qué ubicaciones de memoria puede tener acceso un proceso.

> Según M. E. Russinovich y D. A. Solomon (2004), Windows implementa un sistema de memoria virtual basado en un espacio de direcciones plano, es decir lineal, que le da a cada proceso la ilusión de tener su propio espacio de direcciones, grande y privado. La memoria virtual provee una visión lógica de la memoria que puede no corresponderse con su distribución física. Al momento de la ejecución, el administrador de la memoria con ayuda del hardware se encarga de traducir o *mapear* las direcciones virtuales a direcciones físicas, donde están almacenados realmente los datos. Mediante el control de la protección y el mapeo el sistema operativo se asegura que los procesos no se estorben unos a otros o sobreescriban datos del sistema operativo. Dado que la mayoría de los sistemas tienen mucho menos memoria física que el total de memoria virtual en uso por el proceso en ejecución, el administrador de memoria transfiere o *pagina* parte del contenido de la memoria a disco. La paginación de datos al disco libera la memoria física de manera tal que pueda ser usada por otros procesos o por el propio sistema operativo.

1.2.3 Gestión de archivos

Un archivo es una colección de información relacionada entre sí, definida por su creador. Por lo general, los archivos representan programas (tanto fuentes como objetos) y datos. Los archivos de datos pueden ser numéricos, alfabéticos o alfanuméricos. Los archivos pueden ser de forma libre, como los de texto, o tener un formato rígido. Un archivo consiste en una secuencia de bits, líneas o registros, cuyos significados han sido definidos por su creador. El concepto de archivo es muy general. Las computadoras procesan información, la cual debe ser transmitida, procesada y almacenada. El objeto en un «sistema de archivos» es una entidad abstracta para el almacenamiento o la transmisión de un conjunto de información.

El sistema de archivos propiamente dicho es una colección organizada de tales objetos y el sistema operativo debe proveer primitivas para manipularlos.

1.2.4 Gestión de dispositivos de entrada y salida

Uno de los objetivos de un sistema operativo es ocultar las peculiaridades de dispositivos de hardware específicos, de modo que el usuario no las perciba. Por ejemplo, en UNIX, el subsistema de entrada y salida oculta las peculiaridades de los dispositivos del resto del sistema operativo. Una computadora comunica información a través de sus dispositivos de entrada y salida. Los procesos acceden a estos dispositivos a través de las llamadas a sistema del sistema operativo, provistas para este propósito. El sistema operativo intenta manejar estos dispositivos de una manera tal que puedan ser compartidos de forma eficiente por los procesos que los solicitan.

1.2.5 Servicios

Un sistema operativo crea un entorno para la ejecución de programas; proporciona ciertos servicios a los programas y a sus usuarios. Desde luego, los servicios específicos varían de un sistema operativo a otro, pero podemos identificar algunas clases comunes. Es posible describir un sistema operativo a partir de los servicios que presta, tales como:

- Controlar la ejecución de procesos (creación, terminación, suspensión y comunicación).
- Planificar el uso de la CPU.
- Asignar memoria principal en ejecución.
- Asignar memoria secundaria.
- Permitir acceso controlado desde los procesos a los dispositivos.
- Todo esto de manera eficiente y «transparente» al usuario.

1.2.6 Tipos de sistemas operativos

Dada la diversidad surgida en esta evolución, se han hecho distintos esfuerzos por clasificarlos de alguna manera. Podemos hablar de sistemas por lotes o *batch*, interactivos, monousuarios, de tiempo compartido (*time sharing*), multiusuarios, paralelos, distribuidos, de red, de tiempo real y cliente-servidor.

El término lote o *batch* proviene de la época de los sistemas que utilizaban tarjetas perforadas. En esa época se preparaba un conjunto o lote de tarjetas que contenían tanto las instrucciones para lanzar la ejecución del programa, como los datos de entrada que precisaba. Esta tarea normalmente la realizaba un operador, quien no necesariamente sabía lo que el programa hacía, simplemente disponía el trabajo (*job*) y verificaba que se ejecutara sin problemas y, en caso de haberlos, notificaba al usuario quien, a su vez, se había desentendido de la ejecución. Es decir, el usuario no interactuaba directamente con su programa. Generalmente se trataba de procesos que demoraban varios minutos, tal vez horas, y leían sus datos de entrada a partir de archivos –generalmente almacenados en cinta magnética–, volcando su

salida a impresoras o a archivos. En la actualidad hay más procesos interactivos, no obstante todavía existen procesos no interactivos o *batch*[2].

En un comienzo, esos sistemas sólo ejecutaban un trabajo por vez, luego se desarrolló la técnica que permitió que dos o más procesos ocuparan la misma unidad de memoria principal a la vez, y se la llamó multiprogramación.

> La **multiprogramación** es la técnica que permite que dos o más procesos ocupen la misma unidad de memoria principal a la vez. El grado de multiprogramación es la cantidad de procesos que se tienen en memoria. En un sistema con un único procesador, habrá varios programas en memoria pero sólo uno en ejecución, en un momento específico[3]. Los sistemas de tiempo compartido son una consecuencia lógica de la multiprogramación.

En los sistemas por lotes multiprogramados (*multiprogrammed batch*) los trabajos residían en un grupo almacenado en disco; cuando un trabajo era incapaz de ejecutar, porque estaba esperando la finalización de una operación de entrada o salida, otro trabajo comenzaba a hacerlo. El compartir los recursos de computación entre los procesos que ejecutaban de manera concurrente[4] incrementó significativamente la complejidad del sistema operativo.

Los sistemas de tiempo compartido se desarrollaron con el objetivo de compartir los recursos computacionales –principalmente la CPU– entre muchos usuarios, de manera interactiva, cada uno de ellos trabajando desde su terminal. Para estos sistemas se debe proveer multiprogramación. Dado que las computadoras actuales no se usan de la misma manera, el término en cierta medida ha caído en desuso.

Los sistemas de tiempo real tienen restricciones de tiempo bien definidas; se usan para una aplicación dedicada, tienen una memoria primaria amplia y el almacenamiento secundario es limitado. En los sistemas de tiempo compartido, los usuarios esperan que su proceso termine rápido, es decir, esperan una respuesta rápida del sistema. Los sistemas de tiempo real, en cambio, se caracterizan por el hecho de que habrá consecuencias severas si no se alcanzan propiedades de exactitud, tanto lógicas como de tiempo. Hay dos tipos de sistemas de tiempo real: blando y duro. En un sistema de tiempo real blando, las tareas son realizadas por el sistema tan rápido como sea posible (tienen requerimientos estrictos de tiempo), pero no tienen que finalizar en tiempos específicos –por ejemplo, en la grabación de un CD o en la reproducción de audio y video en los sistemas operativos modernos. En los sistemas de tiempo real duros, las tareas se deben realizar no sólo de manera correcta sino también deben **garantizar** que se harán en un tiempo determinado –por ejemplo, un sistema de control de vuelo dentro de un avión de pasajeros. Las aplicaciones de tiempo real cubren un rango amplio. Muchas de estas aplicaciones son «embutidas» o «embebidas» (*embedded*). Esto significa que la computadora que se está usando está metida dentro de un sistema de manera tal que el usuario no la percibe como tal. Por ejemplo, ciertas impresoras, llamadas «impresoras de red», para grandes volúmenes de impresión –que sean embebidas no implica que sean de tiempo real– (cf. Labrose 1992).

2. Piense en los procesos Windows que figuran en la bandeja de sistema o *systray*, tales como antivirus o programador de tareas.
3. Esto cambia ligeramente en la actualidad con los procesadores multi núcleo.
4. Véase subsección 2.3.3.

Los sistemas paralelos son sistemas multiprocesadores; los procesadores comparten el colector o *bus* y el reloj (*clock*). Si comparten memoria y periféricos, son fuertemente acoplados (*tightly coupled*). El multiprocesamiento puede ser simétrico o asimétrico. Las ventajas de los sistemas paralelos es que mejoran el *throughput* o rendimiento (resolución de mayor cantidad de procesos en un momento dado), permiten compartir periféricos (puede haber varios procesadores accediendo al mismo disco) y suelen tener la capacidad para seguir dando un servicio proporcional al nivel de hardware que sobrevive a fallos, lo que se denomina degradación gradual (*graceful degradation*). Los sistemas diseñados para degradarse gradualmente también se conocen como tolerantes a fallas (*fault tolerant*); esto significa que ante la caída de un procesador, continúa trabajando el resto, absorbiendo su trabajo.

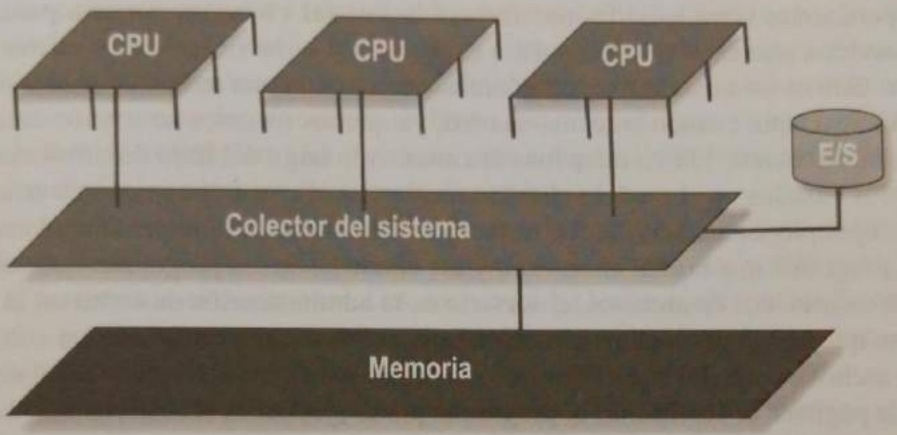


Fig. 1.2 – Sistema SMP.

Hay distintos tipos de sistemas paralelos. Por una parte están los simétricos (SMP, de *Symmetric MultiProcessing*, por sus siglas en inglés), como vemos en la figura 1.2, en los que todos los procesadores comparten un único espacio de direcciones y acceso a todos los recursos, cada procesador tiene una copia idéntica del sistema operativo y esas copias se comunican entre sí, si es necesario. Por otra parte están los asimétricos, en los que uno de los procesadores distribuye y dirige la actividad de los otros. También existen los llamados equipos tándem, que duplican el hardware y el software para asegurar continuidad ante fallas.

De manera similar, existen los sistemas de múltiples hilos simétricos (SMT, de *Symmetric Multi-Threading*, por sus siglas en inglés), por ejemplo los procesadores con tecnología *Hyper-Threading* (HT) de Intel. Se caracterizan porque cada *chip* SMT físico puede tener más de un procesador virtual, con la salvedad de que los procesadores virtuales comparten ciertos recursos (algunos tipos de caché). Y dado que comparten ciertos recursos, los procesadores virtuales no deberían ser tratados de la misma forma que los procesadores corrientes.

En los sistemas distribuidos se dispone el trabajo entre varios procesadores, pero estos no comparten memoria (física) ni reloj. Las ventajas de un sistema distribuido son varias: compartir recursos, mejorar el procesamiento (por división de un proceso en subprocesos), confiabilidad y comunicación.

En los sistemas operativos de red, los usuarios saben que hay varias máquinas y necesitan acceder a los recursos iniciando una sesión en la máquina remota apropiada, o bien transfi-

riendo datos de la máquina remota a su propia máquina. El sistema operativo de red es una capa que se agrega a cada sistema operativo para interactuar con la máquina de servicios. En Windows 95 ó 98, usted accede a «Entorno de red» para ver a sus vecinos y va haciendo clic para acceder a la computadora y luego al recurso compartido (por ejemplo, el disco). Es decir, el acceso a los recursos es explícito. En los sistemas operativos distribuidos (por ejemplo, Amoeba), los usuarios no tienen que saber que hay varias máquinas; acceden a los recursos remotos de la misma manera que a los locales. Hay transparencia de acceso a los recursos, no saben si el recurso al cual están accediendo es local o remoto.

En los sistemas cliente-servidor se utiliza el modelo del mismo nombre. Este modelo se caracteriza porque existen dos entidades de software que requieren de un **servicio**: por una parte está el software cliente, que es un proceso que solicita del servicio a través de un conjunto de operaciones bien definidas que forman la interfaz cliente y, por otra parte, está el software servidor, que es el que administra el servicio y se lo entrega a los clientes que se lo soliciten. Este es un modelo muy comúnmente usado con una red de comunicaciones; es el que mejor se adapta cuando tenemos una red, ya que los recursos no son locales, pero no necesariamente deben residir en máquinas distintas. A lo largo del libro desarrollaremos muchos ejemplos basados en el modelo cliente-servidor, en el que ambos ejecutan en el mismo equipo. Por ejemplo, en un servidor de impresión, el servicio es la impresión misma, el servidor es el programa que brinda el servicio y los clientes son los programas que solicitan el servicio. En un servidor de archivos, el servicio es la administración de archivos, el servidor es el proceso que los gestiona y los clientes solicitan el servicio –que puede ser «crear archivo», «abrir archivo», etc. Tal vez el ejemplo más claro sea el del servidor Web: el servicio es la entrega de páginas, el servidor es el programa que se ejecuta en el sitio remoto al cual está accediendo y los clientes son programas tales como el Internet Explorer.

Un error muy común es llamar servidor a la máquina donde reside el software servidor. El concepto de servidor es un concepto de software, no de hardware.

1.2.6.1 El núcleo del sistema operativo

El núcleo del sistema operativo es el código diseñado para que sea ejecutado mientras el procesador está en modo núcleo. El núcleo no debería ser considerado un programa, la forma más apropiada de describirlo sería la de una rutina de biblioteca, en la que uno o más procedimientos en ella ejecutan siguiendo una trampa, una interrupción o una llamada a sistema. En algún punto el procesador conmuta a modo usuario y el control retorna a un programa de usuario, a pesar de que puede que no sea el mismo programa que estaba ejecutando cuando se ingresó a modo núcleo.

Una **interrupción** (hardware) es una señal asincrónica que indica la necesidad de atención (mediata o inmediata), no depende del programa en ejecución, es externa a él.

Una **trampa** es un tipo de interrupción sincrónica provocada típicamente por una condición excepcional, tal como una división por cero o un acceso a memoria inválido, en un proceso o hilo en modo usuario. Esta trampa provoca un cambio a modo núcleo, en la que el sistema operativo realiza alguna acción antes de devolver el control al proceso que la originó.

Para algunos autores, sólo al núcleo se lo debería considerar como el sistema operativo, dado que es la capa de software que interactúa directamente con el hardware. Cada una de las cuatro áreas de la administración del sistema operativo está implementada en el núcleo, con llamadas a sistema que disparan la ejecución de funciones en esa área. Además, las interrupciones dan como resultado la ejecución de funciones de administración del dispositivo y las trampas generadas por el hardware de administración de la memoria activan el código de la misma. Cada vez que se ingresa al núcleo puede llamarse al administrador de procesos como el último paso antes de que se devuelva el control a un programa de usuario.

Si bien son comunes los sistemas que se caracterizan por tener grandes núcleos **monolíticos**, también hay mucho interés en los sistemas operativos con **micronúcleo** (*microkernel*, en inglés). En estos sistemas, el núcleo tiene mínima funcionalidad, sólo provee funciones esenciales, tales como el control del hardware de administración de la memoria, facilidades para la comunicación entre procesos y manejo de las interrupciones y trampas, así como planificación básica. Los procesos servidores se encargan de proveer las funciones de mayor nivel, tales como los servicios del sistema de archivos. Un programa que requiera de servicios del sistema de archivos usará llamadas a sistema para comunicación entre procesos, para enviar los pedidos al proceso servidor del sistema de archivos.

El uso de procesos servidores mejora la flexibilidad; se pueden activar servidores nuevos sin siquiera reiniciar el sistema. El tamaño pequeño del micronúcleo simplifica la implementación y el portado a nuevos sistemas. Los micronúcleos son más apropiados para los sistemas distribuidos, ya que cuando se usan facilidades de comunicación entre procesos existe poca diferencia si el proceso es local o remoto.

Si la definición de un sistema operativo se expande para incluir a algunos servicios provistos por procesos en capa de usuario, la distinción entre programa de sistema y sistema operativo se torna difusa. Los servicios tales como un sistema de impresión con *spooling* o un gestor de ventanas son difíciles de clasificar. Según Harris (2002), se los podría considerar parte de las capacidades de administración de dispositivo del sistema.

1.3 Estructura y funcionamiento de la computadora

La computadora es una máquina destinada a procesar datos. En una visión esquemática, como la que muestra la figura 1.3, este procesamiento involucra dos flujos de información: el de datos y el de instrucciones. Se parte del flujo de datos que ha de ser procesado. Este es tratado mediante un flujo de instrucciones de máquina, generado por la ejecución de un programa, y produce el flujo de datos resultado.

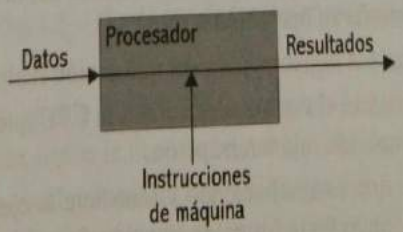


Fig. 1.3 – Flujos de información.

1.3.1 Componentes de una computadora

Para llevar a cabo la función de procesamiento, una computadora con arquitectura de von Neumann (1945) está compuesta por los cuatro componentes básicos representados en la figura 1.4:

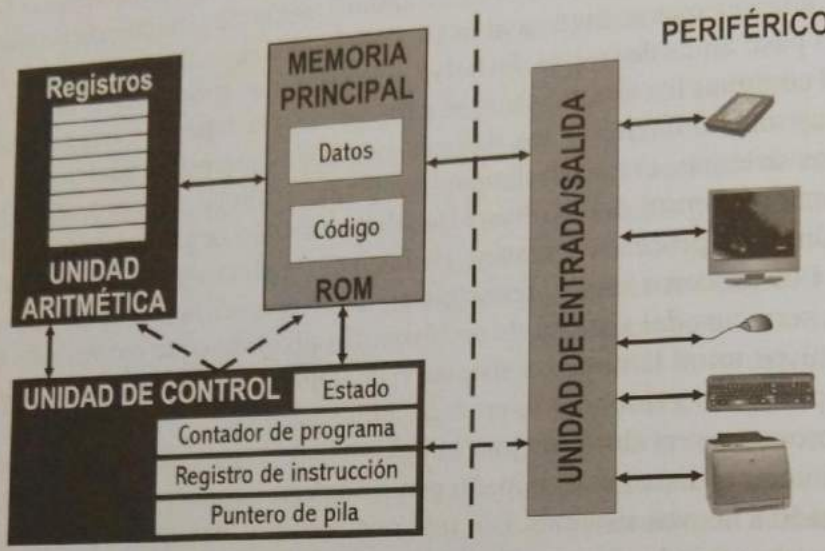


Fig. 1.4 – Componentes de una computadora.

Para lograr sincronización entre CPU y dispositivos de entrada o salida hay dos métodos:

- Transferencia de datos por interrupciones (figura 1.5).
- Transferencia de datos por acceso directo a memoria (DMA).

Antes se realizaba por espera activa (la CPU esperaba que se realizara la entrada o salida).

En un sistema basado en interrupciones hay un controlador por cada tipo de dispositivo, que cuenta con almacenamiento local en *buffer* y registros de propósito general.

El controlador del dispositivo es el responsable de transferir los datos entre el dispositivo que controla y su *buffer* local.

1.3.1.1 Dinámica de una operación de entrada y salida

- La CPU carga los registros apropiados del controlador del dispositivo con la acción que se ha de tomar y continúa su operación normal.
- El controlador analiza los registros para ver qué acción realizar.
- Concluida la transferencia el controlador avisa a la CPU que la transferencia ya terminó. Esto se realiza causando una interrupción.
- La CPU suspende lo que estaba haciendo y transfiere la ejecución a un lugar fijo que tiene la dirección de inicio de la rutina de atención de esa interrupción.
- Dicha rutina transfiere los datos desde el *buffer* local del controlador a la memoria principal.

- Una vez realizada la transferencia, la CPU puede continuar con los procesos interrumpidos.

Cada diseño de computadora tiene su esquema de interrupciones.

Es preciso distinguir las interrupciones de las excepciones. Las excepciones o *traps* son las llamadas interrupciones por software. Las interrupciones son externas al proceso; las excepciones son internas a él.

1.3.1.2 El vector de interrupciones

Es un conjunto de posiciones ubicada normalmente en la parte baja de la memoria, donde se almacenan las direcciones de las rutinas de atención de las interrupciones. Se indexa a través de un número único que se asocia a cada dispositivo. La estructura de interrupciones debe guardar la dirección de la instrucción interrumpida; normalmente se guarda en la pila del sistema. Por lo general, cuando se está atendiendo una interrupción se desactivan las otras. En la actualidad existen arquitecturas que permiten procesar una interrupción mientras se atiende otra, asignando un esquema de prioridades. Se aceptará una interrupción de mayor prioridad a la que se esté atendiendo; las de prioridad menor o igual se enmascaran o desactivan.

1.3.1.3 Hardware de las interrupciones

Según vemos en la figura 1.5 los circuitos electrónicos externos relacionados con las interrupciones en las actuales placas base emula el funcionamiento de los originales PC-AT de IBM (familia de procesadores x86).

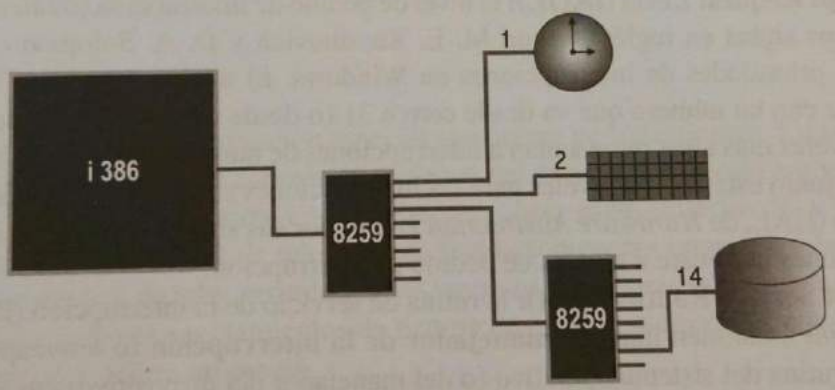


Fig. 1.5 – Hardware de las interrupciones.

- Llega una petición a un chip 8259A. Cada vez que se pulsa una tecla, por ejemplo el controlador de teclado, activa la línea de interrupción conectada al primer chip 8259A.
- El 8259A maestro envía al procesador la petición. Cuando el procesador puede atenderla (inmediatamente si las interrupciones están habilitadas) el chip 8259A envía por el bus de datos el número de interrupción. En nuestro caso, el número enviado es el hexadecimal 0x21. 0x20 es valor programado como base en el 8259A durante el arranque, más 1 que es la línea por la que llegó la interrupción.

- El procesador utiliza este valor como índice en la tabla IDT (*Interrupt Descriptor Table*) para localizar la rutina que sirve la interrupción.

1.3.1.4 Tratamiento de la interrupción

Observe que tanto la tabla de interrupciones como la rutina de tratamiento de la interrupción se han considerado parte del sistema operativo. Esto suele ser así por razones de seguridad, para evitar que los programas que ejecuta un usuario puedan perjudicar a los datos o programas de otros usuarios.

Para entender cómo el núcleo del sistema operativo gestiona las interrupciones tendremos que explicar ciertos términos:

Interrupt Request (IRQ): todos los dispositivos que deseen comunicarse con el procesador por medio de interrupciones deben tener asignada una línea única capaz de avisar al CPU cuando le requiere para realizar una operación. Esta línea se denomina IRQ (de *Interrupt Request*, o pedido de interrupción, por sus siglas en inglés). Es el valor que identifica a una interrupción. En los sistemas de arquitectura x86, las interrupciones externas de entrada y salida llegan a una de estas líneas de un controlador de interrupciones. El controlador, a su vez, interrumpe al procesador en una línea única. Una vez que se interrumpe al procesador, este consulta al controlador para obtener el pedido de interrupción. El controlador de interrupciones traduce el IRQ a un número de interrupción. En Windows se usa este número como índice en la tabla de despacho de la interrupción (IDT, de *Interrupt Dispatch Table*, por sus siglas en inglés) y transfiere el control a la rutina apropiada de despacho de la interrupción. Al momento del arranque, Windows llena la IDT con punteros a las rutinas del núcleo que manejan cada interrupción y excepción.

Interrupt Request Level (IRQL): el nivel de pedido de interrupción (o *Interrupt Request Level*, por sus siglas en inglés), según M. E. Russinovich y D. A. Solomon (2004), es un esquema de prioridades de interrupciones en Windows. El núcleo representa a los niveles internamente con un número que va desde cero a 31 (o desde cero a 15 en Windows para 64 bits). Los niveles más altos representan a interrupciones de mayor prioridad. Si bien el núcleo define el conjunto estándar de niveles para las interrupciones software, la capa de abstracción del hardware (HAL, de *Hardware Abstraction Layer*, por sus siglas en inglés) asigna números de interrupciones hardware a niveles de pedido de interrupción.

Interrupt Service Routine (ISR): la rutina de servicio de la interrupción (ISR, *Interrupt Service Routine*), también llamado **manejador de la interrupción** (o *interrupt handler*, en inglés) es la rutina del sistema operativo (o del manejador del dispositivo) cuya ejecución se dispara por la recepción de una interrupción.

Interrupt Priority Level (IPL): es el nivel de prioridad de la interrupción y se refiere al estado actual de las interrupciones en el sistema, si serán o no aceptados los pedidos de interrupción, es decir, si están o no habilitadas las interrupciones. Pueden indicarse en el hardware (mediante registros en el PIC) o en el software mediante una máscara de bits o un valor entero.

1.3.1.5 Tratamiento diferido

Si el tratamiento de una interrupción conlleva mucho trabajo es recomendable realizar parte de este tratamiento de forma inmediata y parte de forma diferida, ya que de otra manera puede percibirse una degradación en el nivel de respuesta del sistema. Es decir, en un sistema operativo para uso en computadoras portátiles o de escritorio, lo notaremos como «pesado» o «lento» en cuanto a su interactividad. Suelen recibir distintos nombres: manejador de interrupciones de primer y segundo nivel (FLIH, o *First-Level Interrupt Handler*, y *Second-Level Interrupt Handler*, SLIH, respectivamente); también se les suele llamar manejadores de interrupción «duros» (*hard interrupt handler*) o «rápidos» (*fast interrupt handler*) a los primeros y «lentos» (*slow interrupt handler*) o «blandos» (*soft interrupt handler*) a los SLIH.

Tanto Windows como Linux tienen estos mecanismos para el tratamiento diferido de partes del servicio de procesamiento de las interrupciones.

Deferred Procedure Call (DPC): Windows cuenta con una rutina –que realiza la mayoría del trabajo involucrado en el manejo de una interrupción de dispositivo después de que ejecuta la rutina de servicio a la interrupción– llamada DPC (*Deferred Procedure Call*). La rutina ejecuta a un nivel de IRQL que es menor que el de la ISR para evitar bloquear otras interrupciones innecesariamente, según M. E. Russinovich y D. A. Solomon (2004).

Top and Bottom half: por su parte, Linux utilizó un mecanismo hasta la versión 2.3.42 que partía la tarea en «la mitad de arriba» y «la mitad de abajo» (o *top half* y *bottom half*, en inglés); la primera recibía la interrupción de hardware y la segunda procesaba la parte más larga. A partir de la versión 2.5 del núcleo Linux todo el código fue transformado a *softirqs* o *tasklets*, según Wilcox (2003).

Lo que se busca con estos mecanismos es, en definitiva, no perder interrupciones que puedan llegar mientras se está tratando a otra.

1.3.1.6 Clock tick

Un *tick de reloj* es una interrupción especial que ocurre periódicamente. Esta interrupción puede verse como el latido del corazón (*heartbeat*) del sistema. El tiempo entre interrupciones es específico de la aplicación y, generalmente, suele estar entre los 10 y los 200 mS. Mientras más rápido sea el *tick* de reloj, mayor será la sobrecarga impuesta sobre el sistema. La interrupción del *tick* de reloj permite que las tareas sean demoradas durante un número entero del mismo y proveer «agotamientos de tiempo de espera», o *timeouts*, cuando las tareas están esperando que ocurra un evento.

> Un **evento** es un mensaje de software que indica que algo ha pasado y que requiere la respuesta del programa, por ejemplo al presionar una tecla o al hacer clic con el *mouse*, cuando llega un bloque de datos desde el disco, llega un paquete desde la red, o finaliza una operación pendiente. Es una acción que puede ocurrir en cualquier momento, se inicia fuera del ámbito de un programa pero que es manejado por una pieza de código dentro del mismo.

1.3.1.7 Llamadas al sistema

Son la interfaz entre el sistema operativo y un programa en ejecución. Pueden ser instrucciones en lenguaje ensamblador (MS-DOS) o pueden hacerse desde lenguajes de alto nivel como C (UNIX, Minix, Linux y Windows NT). Para ello, el sistema se vale del modo dual de operación de los microprocesadores: **modo núcleo** y **modo usuario**, como se aprecia en la figura 1.6.

> **Ejecución multimodo**: para proveer al sistema operativo con privilegios no otorgados a los programas de aplicación, el hardware debe soportar múltiples modos de ejecución. Comúnmente, se soportan dos modos de ejecución en el procesador, el **modo núcleo** o supervisor (*kernel or supervisor mode*) y el **modo usuario** o protegido (*user or protected mode*). Un único bit en el PSW (*Program Status Word*) registra el modo de ejecución del sistema. Los intentos por realizar ciertas actividades mientras se está en modo usuario dan como resultado una trampa (*trap*). Las actividades restringidas son aquellas cosas reservadas normalmente al sistema operativo, tales como la ejecución de ciertas instrucciones –llamadas «instrucciones privilegiadas»–, el acceso a ciertos registros y el acceso a los dispositivos de entrada o salida.

El control pasa al vector de interrupciones para que una rutina lo atienda. El bit de modo se pone en modo monitor. El monitor analiza la instrucción que provocó la interrupción. Así se ejecuta la solicitud y vuelve el control a la instrucción siguiente a la llamada al sistema. Los parámetros asociados a las llamadas pueden pasarse de varias maneras: por registros, bloques o tablas en memoria o pilas. Hay varias categorías de llamadas al sistema:

- Control de procesos: para finalizar, abortar, cargar, ejecutar, crear, terminar, establecer y obtener atributos del proceso, esperar un tiempo, señalar y esperar evento, asignar y liberar memoria.
- Manipulación de archivos: para crear, eliminar, abrir, cerrar, leer, escribir, volver a posicionar, obtener y establecer atributos de archivo.
- Manipulación de dispositivos: para solicitar, liberar, leer, escribir, volver a posicionar, obtener y establecer atributos de dispositivo.
- Mantenimiento de información: para obtener fecha y hora, datos del sistema y atributos.
- Comunicaciones: para crear, eliminar conexión de comunicación, enviar y recibir mensajes, transferir información de estado, etc.

Podríamos decir que comandamos al núcleo del sistema operativo mediante las llamadas a sistema. Un sistema operativo está definido por los servicios que ofrece en relación con el acceso a entrada/salida, procesos, manejo de memoria, envío de señales, tiempo, etc.

- A los servicios del sistema operativo se accede a través de las llamadas al sistema: `open`, `read`, `fork`, `mmap`, `kill`, `time`, etc.
- La llamada al sistema la invoca un proceso de usuario (o mejor dicho, un proceso en modo usuario) y es servida por el núcleo.

- Una llamada al sistema implica pasar o saltar del código del usuario al código del núcleo. Este salto conlleva un cambio en el modo del funcionamiento del procesador[5].

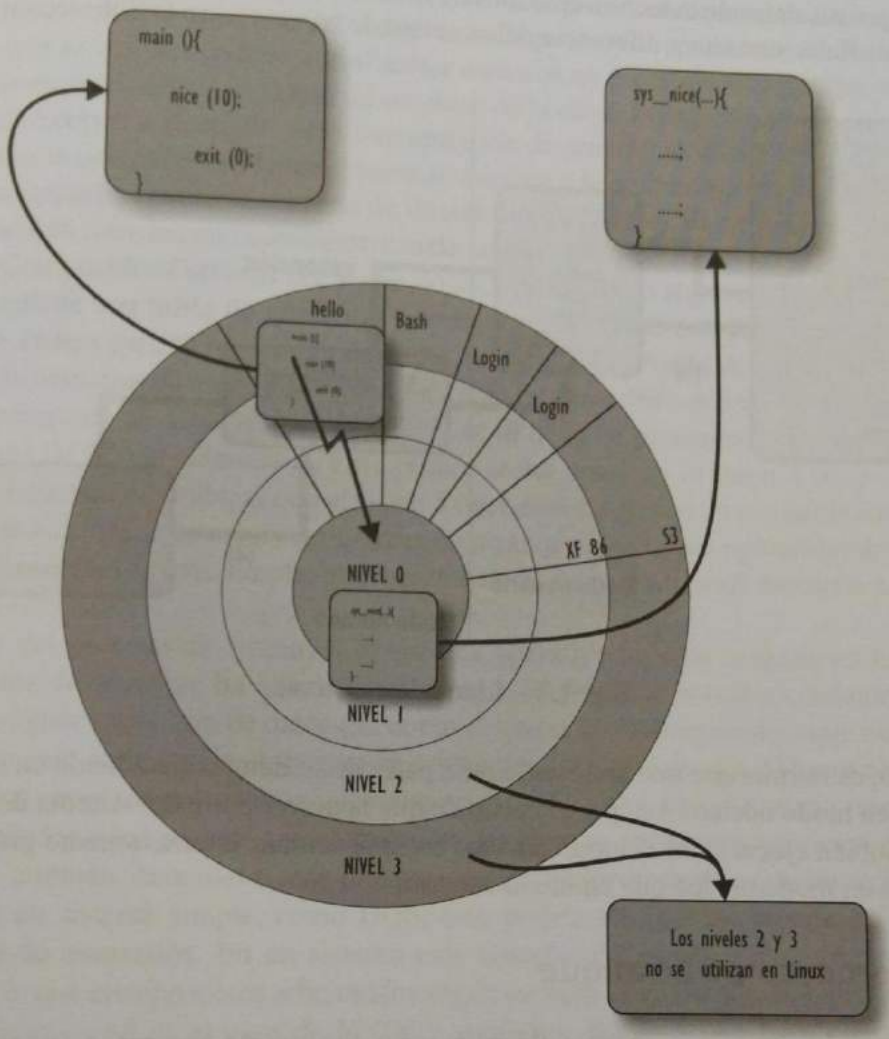


Fig. 1.6 – Modo dual de operación de los microprocesadores.

Las aplicaciones del usuario conmutan del modo usuario al modo núcleo cuando hacen una llamada a servicios del sistema. Por ejemplo, una función `ReadFile` de Windows necesita llamar a la rutina de Windows interna que efectivamente realizará la lectura de datos del archivo, como se muestra en (1) en la figura 1.7. Esa rutina, por el hecho de que accede a estructuras de datos internas del sistema, debe ejecutar en modo núcleo. La transición desde el modo usuario al modo núcleo se realiza mediante el uso de una instrucción especial del procesador que provoca que el procesador permute a modo núcleo (2). El sistema operativo atrapa esta instrucción, notifica que se está solicitando un servicio del sistema, valida los argumentos que la aplicación le pasó a la función del sistema (3) y luego ejecuta la función

5. El procesador debe pasar de modo usuario (acceso restringido a los recursos) a modo supervisor o privilegiado (recordemos que no tiene nada que ver con el superusuario `root`).

interna (4). Antes de regresar el control al código de usuario (6), el modo del procesador se conmuta nuevamente a modo usuario (5). De esta manera, el sistema operativo se protege a sí mismo y a sus datos de la lectura cuidadosa y la modificación por parte de las aplicaciones del usuario. Relacione con y diferencie del «cambio de contexto» en la Subsección 2.1.4.

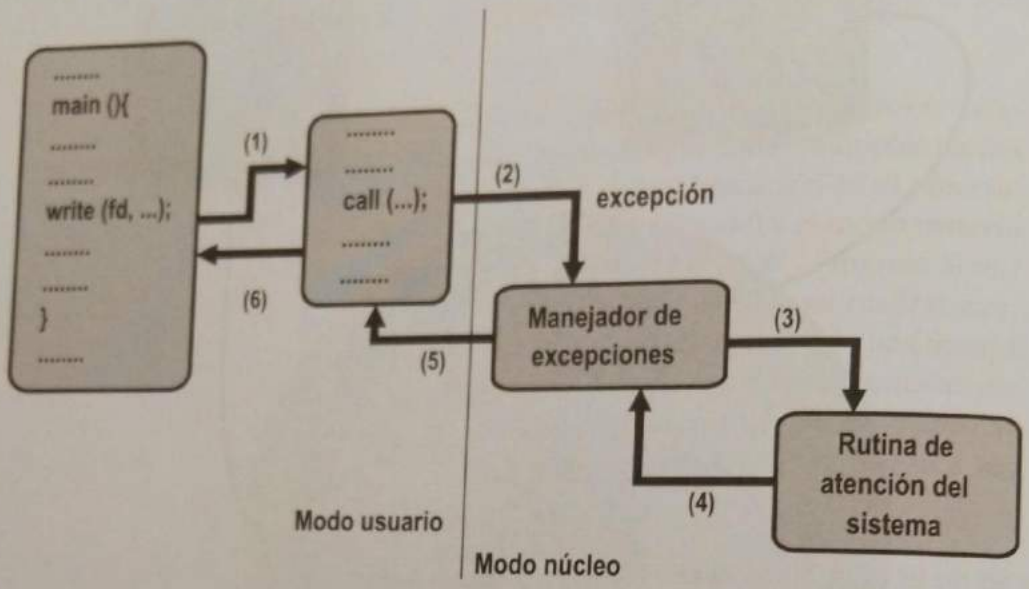


Fig. 1.7 – Llamada a sistema.

Por ello, es normal que una aplicación pase parte de su tiempo ejecutando en modo usuario y parte en modo núcleo. Así, por el hecho de que la mayor parte del sistema de ventanas y gráficos también ejecutan en el modo núcleo, las aplicaciones intensivamente gráficas pasan más tiempo en modo núcleo que en modo usuario.

1.3.2 El proceso de arranque

Cuando se le suministra energía eléctrica a un sistema de computación, se ejecuta un programa en memoria de sólo lectura (*Read Only Memory o ROM*). Luego de ejecutar algunas comprobaciones de diagnóstico, se ejecuta un programa de arranque (*boot*[6]). El programa comprueba la presencia de uno o más dispositivos de arranque (etapa 0). En una PC, un dispositivo de arranque puede ser una unidad de discos flexibles, de CD-ROM o un disco rígido. Una vez hallado el dispositivo, el programa lee el primer sector del dispositivo de arranque y lo carga en memoria principal. Al primer sector de este dispositivo de arranque se le llama «sector de arranque» y debería contener la siguiente etapa (etapa 1) del proceso de arranque –a la que una vez cargada en memoria principal, se le transfiere control para que continúe ejecutando.

6. El término *bootstrap* surge de la asociación de dos palabras inglesas: *boot+strap*. Este término, así expresado, hace referencia a las orejas o agarraderas que tienen algunas botas y que facilitan el calzado. En nuestro caso, el término data de 1950 y se presenta de forma metafórica como verbo, indicando que el *bootstrap* ayuda al arranque sin necesidad del auxilio de otros, de allí que se use también para indicar la compilación de las herramientas que serán usadas para compilar el resto del sistema o programa.

Puede que el sector de arranque no contenga un programa de arranque. Si este sector nunca fue inicializado, se copiarán a la memoria valores indefinidos. Antes de transferir el control a la próxima etapa en el proceso de arranque, el programa de arranque actual revisará que exista uno o más bytes al final del sector de arranque, cuyo contenido es un patrón de bits «mágico» que se usa para indicar que el sector contiene un programa de arranque válido.[7]

En algunos sistemas, el programa de arranque de la etapa 1 carga el sistema operativo. En otros, se debe copiar a memoria y ejecutar una serie de programas de arranque. En cualquier caso, el sistema operativo de alguna manera se copiará a la memoria y el programa de arranque final bifurcará al punto de entrada de inicialización del sistema operativo. Este proceso de arranque será tratado con detenimiento en la subsección ??.

En un Computador Personal (PC), que arranca desde disco rígido, el sector de arranque también contiene una **tabla de partición** (*partition table*). El disco puede ser dividido como máximo en cuatro particiones de las llamadas **primarias**. La tabla de partición contiene la ubicación de comienzo y fin de cada partición. Esta tabla también define si una partición está activa. El programa de arranque de la etapa 1 lee la tabla de partición y copia a memoria el primer sector de la partición activa. Este primer sector contiene la etapa 2 del programa de arranque, diseñado para trabajar con el sistema operativo contenido en esa partición. También puede tener aptitudes para cargar programas de arranque o sistemas operativos de otras particiones. En su aspecto más simple, el programa de la etapa 2 carga en memoria la partición del sistema operativo.

Al final del proceso de arranque, el sistema operativo ha sido cargado en la memoria y el programa de arranque ha bifurcado al mismo. El procesador está ejecutando en modo núcleo. Cualquier estructura de datos (tal como el vector de interrupciones) que no haya sido preparada cuando fue cargado el sistema operativo debe ser inicializada. Si fuera necesario, se inicializan los registros del sistema y los dispositivos. Una vez que el sistema operativo está listo para atender a los procesos, crea todos los procesos que proveen servicios del sistema operativo y, también, crea uno o más procesos para ejecutar programas de inicialización del sistema. En un sistema simple, como DOS, esto podría sólo ser un proceso para ejecutar al intérprete de comandos. En un sistema más complicado, puede existir un programa de sistema `init`, que crea procesos adicionales según se haya especificado en una base de datos de configuración –tal es el caso de UNIX y similares. En cualquier caso, una vez que el sistema es conmutado a modo usuario, y el control conmuta a los procesos en modo usuario, el sistema operativo ha completado su papel en el arranque y el sistema ya está corriendo (cf. Harris 2002).

1.4 Evolución de los sistemas operativos

En Tanenbaum y Woodhull (1998) está contada la historia de los sistemas operativos y su evolución a través de las llamadas «generaciones». Esos conceptos están muy asociados a los tipos de sistemas que acabamos de ver. También puede ver los distintos tipos de sistemas en la obra de Silberschatz y Galvin (1997).

Los primeros sistemas de cómputo electrónico datan de los años 1945-1950, pero estas computadoras no tenían sistema operativo y el usuario que las operaba estaba muy altamente

7. No es «mágico» en un sentido literal. Es una convención.

capacitado y cumplía las funciones de usuario, operador, programador y administrador. Era necesario conocer exactamente el hardware, porque a cada programa había que darle todas las especificaciones físicas, mediante clavijas y paneles, para que ejecutara correctamente, es decir, especificar posiciones de memoria donde almacenar cada dato y además encargarse de la entrada y salida en los periféricos de la época. A este tipo de máquinas se las conoce como «computadoras controladas por programa» (*program-controlled computers*). La «ventaja» es que disponía de toda la máquina para sí de manera real y efectiva; en nuestros tiempos, es tan sólo una ilusión que nos muestran los sistemas operativos modernos. Pero sigue siendo una realidad para quienes usan juegos de consola. Como sabemos que muy probablemente el lector es uno de ellos, utilizamos este modelo para que tenga una analogía con aquellas computadoras.

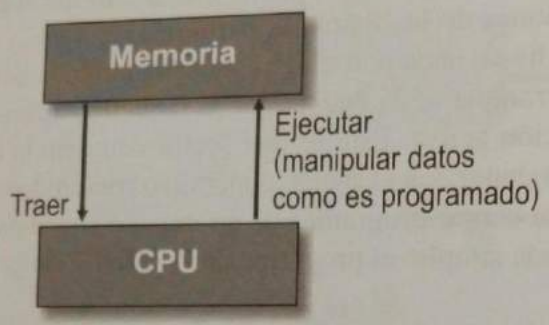


Fig. 1.8 – Arquitectura Von Neumann.

Luego de los trabajos de Turing (1936), acerca de la máquina de cómputo universal, y de von Neumann (1945) (ver Burks, Goldstine y von Neumann 1946), y su modelo de computadora con programa almacenado (*stored-program computer*), según vemos en la figura 1.8, debemos reconocer el trabajo de Tom Kilburn y Frederic William al desarrollar en la Universidad de Manchester la *Small-Scale Experimental Machine* (SSEM), en junio de 1948, que fue la primera computadora con programa almacenado. El primer programa que ejecutó fue el de encontrar a todos los divisores de 2^{18} (262144), simplemente probando dividir 2^{18} entre todos los números enteros menores que él, hasta el 1. Claro está que aquellas divisiones que tuvieran resto cero, indicaban cuáles eran los divisores. En el programa la división se hizo por el método de restas sucesivas.

Poco tiempo después, en agosto de 1948, se comenzaba la construcción de la Manchester Mark 1, en la misma Universidad de Manchester, quedando operativa en abril de 1949 y ejecutó un programa para encontrar números primos de Marsenne entre el 16 y 17 de junio de 1949. La Mark 1 no tenía aún lenguaje ensamblador (los programas se debían escribir en binario), ni sistema operativo. A esta máquina le siguió la Ferranti Mark 1, reconocida como la primera computadora de propósito general y disponible comercialmente del mundo.

En aquellas épocas comenzaban a desarrollarse «bibliotecas» (*libraries*) para ayudar a la programación con la entrada y salida. Estas bibliotecas fueron las semillas germinales del sistema operativo.

Las computadoras se expandieron del campo militar al de los negocios, entonces el dinero entró en las ecuaciones. A medida que se fueron haciendo más potentes, el tiempo de cómputo fue disminuyendo y esto permitió más usuarios. Aunque al principio, como las

máquinas seguían ejecutando un solo programa por vez, los programas (y sus respectivos usuarios) pasaban por la máquina de a uno a la vez. En estos momentos el lector debería estar imaginándose cediéndole la consola de juegos a su hermano menor.

Las computadoras eran mucho más costosas que en estos tiempos, de manera que había que encontrar formas de hacerlas rendir más. También era necesario llevar la contabilidad del tiempo consumido por los programas, del uso de la CPU, la memoria, el espacio de almacenamiento y el papel impreso. Todas estas funciones se fueron incorporando paulatinamente al sistema operativo. De a poco también fue delineándose la figura del operador, distinguiéndola de la del usuario/programador, que todavía seguía siendo el mismo.

De forma paralela al desarrollo del hardware y de los sistemas operativos, está también el desarrollo de los lenguajes de programación. Grace Murray Hopper fue una científica de la computación, programadora de la Harvard Mark I (*Automatic Sequence Controlled Calculator* de IBM), y se le reconoce haber desarrollado el primer compilador para un lenguaje de programación –al que llamó el «compilador A»– en el año 1951 para la UNIVAC I, cuya primera versión fue A-0. Posteriores mejoras llevaron al compilador a la versión B-0 (conocido como FLOW-MATIC), que fue tomado como base para COBOL (*COmmon Business-Oriented Language*). La idea que logró llevar adelante Grace Hopper fue la de desarrollar un lenguaje de programación cercano al idioma inglés en lugar del código de máquina, por esto se le llama «la madre del lenguaje COBOL»'. A partir de una anécdota que narra que al encontrar una mariposa de la luz pegada a un *relay*, que impedía su funcionamiento, se le atribuye, quizá incorrectamente, haber acuñado el término *computer bug* (literalmente, «bicho de computadora») y el de *debugging*. Lo que sí es seguro es que ha difundido los términos porque siempre contaba esta anécdota en sus conferencias.

Se acepta comúnmente que el GM-NAA I/O haya sido el primer sistema operativo de la historia (1956) que ejecutaba en una IBM 704. Se los suele denominar más propiamente «monitores» o «programas monitores» del computador. Piense que en estos primeros tiempos los propios usuarios/programadores eran los que desarrollaban estos programas monitores, para poder trabajar más cómodamente sobre el equipo, y que estos programas monitores eran específicos para la máquina sobre la cual ejecutaban. Obviamente, también era algo común y aceptado el compartir el software. El concepto de software propietario apareció mucho después.

Luego, a partir de este sistema operativo se desarrolló el SOS (SHARE Operating System), al que luego IBM introduciría en las máquinas IBM 7090 (en noviembre de 1959) y IBM 7094 (en septiembre de 1962), de la segunda generación de computadoras, con el nombre de IBSYS.

La empresa IBM (*International Business Machines*) tuvo un gran protagonismo en esta época. Desarrollaba el hardware y el software para el equipo, tanto el sistema operativo como el software de soporte. John Backus trabajando en IBM desarrolló el lenguaje y el compilador FORTRAN (una palabra compuesta a partir de *The IBM FORmula TRANslating System*), en el año 1957, para el equipo IBM 704.

En la década de 1960 IBM se abocó al desarrollo de la serie *System/360*. Todos los modelos de la serie usaban la misma arquitectura de instrucciones, para los cuales IBM intentó desarrollar el mismo sistema operativo, el *OS/360*. La idea del *System/360* era la de hacer una clara distinción entre arquitectura e implementación, permitiéndole a IBM sacar al mercado un conjunto de diseños compatibles a diferentes precios. Tuvo mucho éxito porque los clien-

tes podrían comprar un sistema pequeño y migrarlo a uno más grande, si sus requerimientos crecían. El *OS/360* apareció en 1964 y fue de los primeros sistemas operativos en tener dispositivos de almacenamiento de acceso directo; era un sistema operativo de procesamiento por lotes. En 1967 IBM presentó a *TSS/360* (*Time Sharing System*) uno de los primeros sistemas operativos de tiempo compartido, que corría sobre el Modelo 67 de la serie. TSS/360 fue la primera implementación de multiprocesamiento simétrico fuertemente acoplado en *mainframes*.

Pero ya en el año 1961 había aparecido CTSS (*Compatible Time-Sharing System*), desarrollado por el MIT para una IBM 7094. Entre sus características podemos destacar que traía un programa para procesar texto en modo *batch*, insertándole directivas dentro del mismo texto, que se llamaba RUNOFF; además, traía un programa para intercambiar mensajes entre usuarios de la misma máquina. En Corbató, Merwin-Daggett y Daley (1962) se discute brevemente la necesidad del tiempo compartido, algunos de los problemas de su implementación y un algoritmo de planificación. Fue un sistema que influenció mucho a sus sucesores, entre ellos a Multics. Esta máquina estuvo encendida hasta el 20 de julio de 1973.

> Hubo dos usos del término, *time sharing*, sin el guión y antes de 1960, un método de implementar multiprogramación y *time-sharing*, con el guión y luego de 1960, una técnica para desarrollar computación interactiva (ver Lee 1992). Acerca del primer uso del término se puede ver el trabajo de Strachey (1959).

El sistema de multiprogramación THE aparece en 1968. El líder del proyecto de diseño fue Edsjer Dijkstra (ver Dijkstra 1968b), quien introdujo la primera forma de segmentación de memoria basada en software, porque el equipo no tenía administración de memoria basada en hardware.

El próximo hito importante en la historia de los sistemas operativos es la aparición de Multics (*Multiplexed Information and Computing Service*) en el año 1969, luego de varios años de desarrollo. Fue un proyecto cooperativo en el que trabajaron muchas personas (de distintas empresas) lideradas por Fernando Corbató (ver Corbató y Vyssotsky 1965). Era un proyecto muy ambicioso que no tuvo el éxito esperado pero tuvo mucha influencia en la historia. Se pretendía que estuviera encendido y operativo durante todos los días de la semana (alta disponibilidad). Implementó el enlace dinámico (*dynamic linking*) y el almacenamiento en un único nivel (*Single Level Store o SLS*), en el que todo el almacenamiento del que dispone un proceso en la computadora se ve como un único espacio bidimensional de direcciones que apuntan a páginas. Estas páginas las gestiona el sistema operativo y las hace residir en memoria principal o archivos en almacenamiento secundario.

Los Laboratorios Bell (*Bell Labs*), de AT&T, participaron del proyecto Multics pero se retiraron para utilizar sus recursos en otros proyectos. Ken Thompson trabajaba en *Bell Labs* y junto con Dennis Ritchie comenzaron a desarrollar en 1969 otro sistema en una PDP-7, al que Brian Kernighan jocosamente llamó Unics –en contraste con Multics–, pero luego cambió el nombre a UNIX. Luego, el sistema se implementó en una PDP-11/20.

Contrariamente a lo que se decía en aquella época acerca de que un sistema operativo, dada su complejidad y dependencia con el hardware, debía escribirse en *assembler* por razones de desempeño, ellos lo escribieron en lenguaje C (ver Kernighan y D. M. Ritchie 1988), logrando que UNIX fuera mucho más portable que otros sistemas operativos que le precedieron y sin problemas de desempeño. UNIX se distribuía originalmente con su código fuente.

principalmente en universidades, por lo que se lo utilizó mucho como material didáctico para la enseñanza del sistema operativo (ver Lions 1996). Un ejemplo notable de esto es el *Berkeley Software Distribution*, BSD o Berkeley UNIX, cuya primera versión fue en 1977. La última versión de UNIX con su código fuente fue la sexta aparecida en mayo de 1975.

Esta portabilidad le permitió volverse la elección para una segunda generación de minicomputadoras y la primera generación de estaciones de trabajo o *workstations*. Como seguía siendo propiedad de AT&T, sólo tenían el dinero para la licencia las grandes compañías y universidades de Estados Unidos. Esta situación daría lugar al nacimiento del movimiento por el software libre.

El 25 de Octubre de 1977 aparece la primera versión del sistema operativo VMS (*Virtual Memory System*) de la empresa DEC (*Digital Equipment Corporation*), que se distribuye con la VAX-11/780. El nombre completo de este sistema operativo es VAX-11/VMS y su última versión fue la 5 en el año 1991. Era un sistema operativo de 32 bits; su diseñador fue David Cutler, quien fue luego diseñador del Windows NT. El VMS tenía un sistema de archivos que implementaba entrada y salida orientada al flujo (*stream*) o al registro, listas de control de acceso (*Access Control List o ACL*) y número de versión en el nombre del archivo.

Poco antes de 1980 aparecen las computadoras hogareñas; sus procesadores eran de 8 bits, a diferencia de los 32 que ya estaban usando la VAX y otros. El *Control Program for Microcomputers* – o CP/M –, del año 1973, fue un sistema operativo creado por Gary Kildall, originalmente para las microcomputadoras basadas en el procesador 8080/85 de Intel. Como Gary Kildall trabajaba sobre una DEC PDP-10 con VMS como sistema operativo, imitó los comandos y las convenciones para nombrar archivos. Gordon Eubanks, alumno de Gary Kildall, creó un intérprete BASIC para el sistema. Las primeras versiones de CP/M y de este intérprete BASIC estaban en el dominio público[8].

Gary Kildall fundó la empresa Galactic Digital Research en 1976 (luego cambió su nombre a Digital Research Inc.). Fue pionero en el desarrollo de la BIOS (*Basic Input Output System*), porque tenía que soportar un creciente número de fabricantes de hardware que requerían CP/M. Los fabricantes le colocaban a las microcomputadoras un chip ROM-BIOS, de manera que CP/M se cargaba inmediatamente al encenderla. Kildall reescribió completamente CP/M para poder hacer uso de las ROM-BIOS. Para 1978 CP/M ya corría en Apple, Radio Shack, Commodore, Zenith y Sharp, entre otras, y se entregaba con Microsoft BASIC y FORTRAN, UCSD Pascal, WordStar de MicroPro y dBase de Ashton-Tate.

Inicialmente, el CP/M estaba confinado a un único hilo de ejecución y no más de 64 kilobytes (64K) de RAM; las versiones posteriores (MP/M) agregaron variaciones para múltiples usuarios y CP/M fue migrado a procesadores de 16 bits, como el Intel 80286. MP/M fue desarrollado por Digital Research Inc. (DRI) en 1979 y permitía que múltiples usuarios simultáneos compartieran esos pequeños microcomputadores. Su núcleo poseía múltiples hilos de ejecución y tenía un planificador basado en prioridades; ofrecía protección de memoria, entrada y salida concurrente (ver subsección 2.3.3) y soporte para *spooling*,[9] además de permitir que cada usuario pudiera ejecutar más de un programa y permutar entre ellos. Necesitaba un mínimo de 32K de RAM para ejecutar. Ambos eran un gran avance para su época, pero se vieron desplazados comercialmente por DOS de IBM y Microsoft. Tanto CP/M como

8. Es decir, que no está alcanzado por los derechos de autor y propiedad intelectual (copyright), no se lo debe confundir con las licencias libres como la Licencia Pública de GNU (GPL).

9. *spool: Simultaneous Peripheral Operation On-line*

PC/DOS y MS-DOS usaban una interfaz de línea de comandos (*Command Line Interface o CLI*).

Gary Kildall también desarrolló una interfaz gráfica de usuario (*Graphical User Interface o GUI*), a la que llamó GEM (*Graphical Environment Manager*), y que fue presentada en noviembre de 1983 en COMDEX (*Computer Dealer's Exhibition*), mientras que la GUI de Microsoft –Windows 1.0– fue presentada en la primavera de 1985 en el mismo lugar.

En abril de 1980, Tim Paterson comenzó a trabajar en QDOS (*Quick and Dirty Operating System*), copiando las llamadas a sistema de CP/M y simplificándolo. El sistema debía correr sobre un 8086 de Seattle Computer Products (SCP), que tenía originalmente sólo al BASIC de Microsoft. Este sistema operativo pronto cambió de nombre a 86-DOS y fue posteriormente comprado por Microsoft para comercializarlo bajo el nombre de MS-DOS. SCP le había solicitado esto a Paterson porque sus ventas languidecían y DRI tardaría un tiempo en lograr una versión de CP/M para él mismo. Observe el lector que, más allá de la competencia comercial, tenemos acá otro ejemplo de que el software debe acompañar al desarrollo del hardware.

Microsoft, en diciembre de 1980, le compró una licencia no exclusiva a SCP del sistema 86-DOS y le pidió a Tim Paterson que lo portara al IBM-PC, que usaba un procesador 8088 de Intel. Luego, Microsoft le compró a SCP la totalidad de los derechos del 86-DOS y, a su vez, le otorgó una licencia a IBM para ese computador. IBM le encontró unos 300 errores, escribió los manuales y llamó a ese sistema operativo *IBM Personal Computer Disk Operation System 1.0*, o PC-DOS 1.0, y lo lanzaron en 1981.

A partir de la misma fecha, Microsoft comenzó a comercializar el mismo sistema operativo bajo el nombre de MS-DOS, pero conservando la totalidad de los derechos y el código fuente.

Mientras tanto, en el terreno de las estaciones de trabajo y *mainframes* seguía creciendo UNIX en sus distintas formas, sabores o *flavors*. En enero de 1984, Richard Stallman renunciaba a su trabajo en el Laboratorio de Inteligencia Artificial del MIT para comenzar a desarrollar software para el Proyecto GNU, cuyo anuncio se había efectuado el 27 de septiembre de 1983. El objetivo era desarrollar software libre, que pudiera ser usado con cualquier fin, distribuido, modificado y redistribuido, suministrando siempre el código fuente pero con la expresa prohibición de quitar alguna de estas atribuciones o libertades –como se las llama en la licencia de uso (*GNU Public License o GPL*).[10] Los primeros desarrollos, tales como el compilador de lenguaje C, el editor de textos Emacs, la biblioteca de C y el intérprete de comandos *bash*, se efectuaron sobre estaciones de trabajo UNIX; el desarrollo del núcleo *GNU Hurd* comenzó más tarde, según DiBona, Ockman y Stone (1999).

Por otra parte, el 24 de enero de 1984, Apple Computer Inc. presentaba el computador personal Macintosh conjuntamente con el sistema operativo Mac OS. Al Macintosh se le reconoce el haber popularizado la interfaz gráfica de usuario. El sistema operativo estaba parcialmente basado en el Lisa OS, que traía pre instalado la computadora Lisa de Apple en el año 1983; también, tomaba conceptos de la Xerox Alto producida por Xerox PARC.

Andy Hertzfeld, George Crow, Joanna Hoffman, Burrell Smith, Bill Atkinson y Jerry Manock eran integrantes del equipo de diseño del Macintosh. En particular, Andy Hertzfeld escribió una buena parte del software del sistema original del Macintosh incluyendo al que

10. Note la diferencia con dominio público.

estaba grabado en la ROM,[11] así como también el *Control Panel* (Panel de Control) y el *Scrapbook* (Portapapeles), que en años posteriores vimos en Windows (ver Hertzfeld 2004).

Las primeras versiones del sistema operativo Mac OS sólo permitían ejecutar una aplicación por vez y en la administración de la memoria se producía mucha fragmentación, que luego solucionaron gracias a un *handle* reubicable (un puntero a una tabla no reubicable de otros punteros). La primera versión del Mac OS (*System 1*) se distinguió fácilmente de otros sistemas operativos de su época, porque estaba enteramente basada en una GUI y no en una CLI. Traía una aplicación usada para la administración de archivos, llamada *Finder*, que también se encargaba de mostrar el *Desktop* (Escritorio), estas aplicaciones residían en un *folder* o carpeta llamado *System Folder* (es decir, Carpeta de Sistema) en la que también estaba el *Printer Driver* o manejador de impresora, que interactuaban con el sistema.

El 20 de noviembre de 1985, Microsoft presentó a Windows 1.0, que era un ambiente de operación gráfico de 16 bits, es decir que no era un sistema operativo en sí mismo, era una aplicación que ejecutaba sobre sistema operativo DOS.

En el año 1987 aparece el sistema operativo Minix, creado por Andrew S. Tanenbaum en la Vrije Universiteit de Amsterdam para poder ejemplificar los principios de un sistema operativo (ver Tanenbaum y Woodhull 2005). Es un sistema operativo tipo UNIX (*UNIX-like*), que está basado en una arquitectura micronúcleo (*microkernel*) y su intención es netamente educativa. Minix 1 fue compatible a nivel de sus llamadas a sistema con la versión 7 de UNIX. Originalmente ejecutaba sobre microcomputadoras IBM PC, pero luego fue portado a Motorola 68000 y SPARC; además, ejecuta en Atari ST, Commodore Amiga y Macintosh. Si bien el código fuente está incluído en el libro y en un disco compacto, sus primeras licencias restringían sus modificaciones y redistribuciones. La versión 3 de Minix tiene licencia BSD, mucho más permisiva.

Aunque Microsoft había participado al comienzo, en abril de 1987, IBM presenta a OS/2 (*Operating System/2*). Era parte del *IBM Personal System/2* y les permitiría a los programadores desarrollar sus aplicaciones utilizando el modo protegido del procesador, ya que hasta ese momento las aplicaciones hacían mucho uso del acceso directo al hardware o a la BIOS porque el sistema operativo se los permitía. Para ello, estaba provisto por una extensa API (*Application Program Interface*) para controlar el dispositivo de video y manejar eventos del teclado y el *mouse*. Al comienzo no tenía GUI, esta apareció con OS/2 versión 1.1 en el año 1988 y tuvo un aspecto muy parecido al de Windows 2.1.

También en el año 1988 aparece Windows 2.0, como un ambiente de operación gráfico de 16 bits, que ejecutaba como una aplicación de MS-DOS. Sus aplicaciones eran la calculadora, el calendario, el reloj y el block de notas, entre otras.

El conjunto de normas estándares POSIX (*Portable Operating System Interface*) de la IEEE (*Institute of Electrical and Electronics Engineering*) también data del año 1988 y su objetivo es establecer una API para software compatible con el sistema operativo UNIX. El nombre fue sugerido por Richard Stallman.

Windows 3.0 aparece el 22 de mayo de 1990 con mejoras técnicas, para permitir un mejor uso de las capacidades de administración de la memoria de los procesadores 80286 y 80386 de Intel. Aparecen el Administrador de Programas (*Program Manager*), el Administrador de Archivos (*File Manager*) y el juego de cartas Solitario. Podía correr en tres modelos de me-

11. El Macintosh tenía más código grabado en ROM que el que tenían las BIOS en las PC.

moria diferentes: el modo real (*real mode*), para procesadores anteriores al Intel 80286; el modo estándar (*standard mode*), para computadoras con procesador Intel 80286, que comprendía con su modo protegido y el modo mejorado 386 (*386 enhanced mode*), para las computadoras más nuevas con un procesador Intel 80386 –este modo correspondía al modo protegido del procesador y al modo virtual 8086. El modo mejorado 386 tenía todos los beneficios: Direccionamiento y paginado de 32 bits, para tener acceso a la memoria más rápido, y el modo virtual 8086, que permitía ejecuciones más seguras de programas MS-DOS, porque ahora podían ejecutar dentro de una máquina virtual. En marzo de 1992 aparecería Windows 3.1. Windows 3.11 no fue una versión aislada de la serie sino que fue una actualización de Windows 3.1.

El 14 de mayo de 1991 sale a la luz el núcleo del sistema operativo Linux versión 0.01, desarrollado por Linus Torvalds. Era un sistema operativo tipo UNIX, de arquitectura monolítica, y ejecutaba en procesadores compatibles con el 80386 de Intel y hardware de PC; pero era extremadamente limitado el soporte de manejadores (*drivers*) de dispositivos. El único sistema de archivos que se reconocía era el de Minix. No fue sino hasta el 14 de marzo de 1994 que apareció la siguiente versión de hito, Linux 1.0. Tal vez la función individual nueva más importante fue el trabajo con redes porque la versión 1.0 incluía soporte de los protocolos de redes TCP/IP estándar de UNIX. El núcleo 1.0 también incluía un sistema de archivos nuevo muy mejorado, sin las limitaciones del sistema de archivos Minix original. En junio de 1996 apareció Linux 2.0, con la adición de dos nuevas capacidades importantes: soporte de múltiples arquitecturas, incluido un traslado a un sistema Alpha nativo de 64 bits, y soporte de arquitecturas multiprocesador.

Varios elementos se conjugaron en la historia de los sistemas operativos para que apareciera Linux. Por una parte, el Proyecto GNU, que había iniciado Richard Stallman, había ya creado una gran cantidad de herramientas y aplicaciones tales como el compilador GNU C, bibliotecas y editores de texto. El sistema de ventanas X Window no había sido desarrollado por el Proyecto GNU; aún así, tenía una licencia compatible con el mismo, pero el desarrollo del núcleo del sistema operativo GNU Hurd estaba atascado e incompleto. Por otra parte, estaba disponible el código fuente del sistema operativo Minix, pero no estaba permitida su modificación y redistribución. Tanenbaum se mostraba renuente a introducirle modificaciones porque quería mantenerlo sencillo para que siguiera teniendo utilidad a sus fines educativos.

Por ello, Linus Torvalds, siendo aún estudiante en la Universidad de Helsinki en Finlandia, se decide a desarrollar un sustituto de Minix pero con una licencia GNU (a partir de la versión 0.99) y arquitectura monolítica. Esto provocó que Tanenbaum publicara una nota en Usenet declarando que Linux era obsoleto, que estaba muy atado a la arquitectura x86 y que, por lo tanto, no era portable. La respuesta de Torvalds no se hizo esperar, lo que generó un famoso debate que puede leerse actualmente en Internet.

En definitiva, lo que Linus Torvalds desarrolló fue el núcleo, el cual en sí mismo no es usable, sino a través de sus aplicaciones, las cuales provienen en su mayoría del Proyecto GNU.

Y como tanto el núcleo como las bibliotecas y aplicaciones para GNU/Linux se ofrecen en forma de archivos fuente, no ejecutables, el sistema no sería usable si no fuera por las **distribuciones**, que son una gran cantidad de software ya compilado y que se entrega generalmente en algún medio de almacenamiento, por ejemplo DVD y que, no obstante esto, también está disponible en sitios de Internet a los que se les llama **repositorios**, que suelen

contener **paquetes** de software y actualizaciones de los mismos. Las primeras distribuciones comenzaron a aparecer en 1992, MCC Interim Linux en febrero, luego TAMU, SLS e Yggdrasil.

Windows NT 3.1 fue el primer sistema operativo de la línea NT (*New Technology*) para servidores y computadoras de escritorio y fue hecho público por Microsoft el 27 de julio de 1993. El número de versión fue elegido así para hacerlo coincidir con Windows 3.1. Podía correr sobre procesadores x86 de Intel (80386 y posteriores), Alpha de DEC y MIPS R4000. Para su desarrollo, Microsoft contrató a un grupo proveniente de *Digital Equipment Corporation* (DEC), liderados por Dave Cutler; por eso, muchos elementos de su diseño reflejan aspectos del VMS y el RSX-11. Fue un diseño y desarrollo nuevo, es decir que no se utilizó la base de código escrita para Windows 3.x. Al ser multiplataforma, la diversidad de dependencias del hardware se oculta mediante el módulo *Hardware Abstraction Layer o HAL*.

La siguiente versión de la línea NT fue Windows NT 3.5, la cual se conoció el 21 de septiembre de 1994. El objetivo principal era que el usuario lo percibiera más rápido que el anterior. Nueve meses después, el 30 de mayo de 1995, aparece Windows NT 3.51 al cual Microsoft le dió soporte hasta el 31 de diciembre de 2001.

El 24 de agosto de 1995, Microsoft presenta a Windows 95 como un sistema operativo basado en una GUI, a diferencia de sus predecesores que eran ambientes de operación gráfica, es decir, aplicaciones que ejecutaban sobre el sistema operativo MS-DOS. Pero al parecer se trataba de un MS-DOS 7.0 con un Windows 4.0, es decir, otra aplicación gráfica pero finamente enlazada con un sistema operativo MS-DOS, porque había una manera de instalarlo con otra versión de DOS, que era la de Digital Research Inc. –conocida como DR-DOS. Caldera, una distribución de Linux que en aquella época tenía los derechos de DR-DOS, demandó a Microsoft: ambas compañías llegaron a un acuerdo económico extra judicial y la demanda fue retirada. El sistema de archivos del Windows 95 cambió a VFAT, de 32 bits, que permitió nombres largos en los archivos –hasta 255 caracteres, mezclando mayúsculas y minúsculas. Esto representó un avance frente a FAT, que sólo permitía nombres de archivos de 8 caracteres de longitud en mayúsculas y tres caracteres más para indicar la extensión o tipo de archivo. Los requerimientos mínimos para instalarlo eran un procesador Intel 80386 DX, 4 MB de RAM y 50 MB de espacio en disco rígido, pero los óptimos eran un procesador Intel 80486 con 8 MB de RAM.

Windows NT 4.0 es un sistema operativo de núcleo híbrido, para procesadores de 32 bits, con desalojo, gráfico y orientado a los negocios. Está diseñado para trabajar ya sea con uniprocesadores como multiprocesadores simétricos e hizo su aparición el 29 de julio de 1996. Una diferencia significativa con las versiones previas de Windows NT es que el *Graphics Device Interface, o GDI*, está incorporado dentro del núcleo para acelerar la GUI, dando como resultado una mejora sustancial en el desempeño, comparado con Windows NT 3.51.

Microsoft también ha desarrollado núcleos de sistemas operativos para sistemas embebidos, que no son versiones recortadas ni modificadas de los otros núcleos, tal fue el caso de Windows CE, cuya primera versión fue presentada el 18 de noviembre de 1996, y que luego ha pasado a llamarse Windows Embedded Compact y ejecuta en procesadores Intel x86 y compatibles, MIPS y ARM.

Según M. E. Russinovich y D. A. Solomon (2004), el núcleo de Windows NT es similar a la mayoría de los sistemas UNIX en el sentido de que son sistemas operativos monolíticos ya que la mayoría del código del sistema operativo y los manejadores de dispositivos comparten el mismo espacio de memoria protegido en modo núcleo. Esto significa que cualquier componente del sistema operativo o manejador de dispositivo puede potencialmente corromper datos que están siendo usados por otros componentes del sistema operativo. A pesar de que muchos afirman que Windows NT es un sistema basado en micronúcleo, no lo es en la definición clásica de micronúcleos, en la que los componentes principales del sistema operativo, tales como el administrador de memoria, el administrador de procesos y el administrador de entrada y salida, ejecutan como procesos separados en sus propios espacios privados de direcciones, en capas sobre un conjunto de servicios que el núcleo posee.

El 25 de junio de 1998, Microsoft presentó a Windows 98 como un sistema operativo gráfico sucesor de Windows 95. Como tal, era un producto monolítico híbrido de 16 y 32 bits, basado en MS-DOS. Fue el primero en utilizar el *Windows Driver Model o WDM*, que se siguió utilizando en Windows 2000 y Windows XP.

Windows 2000 es una línea de sistemas operativos producidos por Microsoft para ser usados en computadoras de escritorio, *notebooks* y computadoras para servidores. Fue anunciado en diciembre de 1999, pero estuvo disponible el 17 de febrero del 2000 y fue el sucesor de Windows NT 4.0. Incorporaba la versión 3 del sistema de archivos NTFS, el Sistema de Archivos Encriptado (*Encrypting File System o EFS*) y el Sistema de Archivos Distribuido (*Distributed File System o DFS*). Por otra parte, el 14 de setiembre de 2000, Microsoft presentó a Windows Millennium Edition, más conocido como Windows ME, el cual fue el último de la línea de Windows monolíticos híbridos de 16 y 32 bits, basados en MS-DOS, es decir que fue el sucesor de Windows 98. Microsoft cesó su soporte el 11 de julio de 2006.

En agosto de 2001 se anuncia, pero recién el 25 de octubre de ese mismo año, Microsoft presenta a Windows XP (*eXPerience*), que fue el sucesor de Windows 2000 Professional y de Windows ME. Es otro sucesor de la línea de NT, por lo tanto, no tiene código proveniente de Windows 98.

Microsoft anunció la aparición de Windows Vista el 22 de julio de 2005, el desarrollo se completó el 8 de noviembre de 2006 y quedó completamente disponible a partir del 30 de enero de 2007. El objetivo primordial establecido por Microsoft fue el de mejorar la seguridad del sistema operativo. Se reestructuró completamente la arquitectura de los subsistemas de audio, la impresión, la pantalla y las redes en particular, ya que se le incorporó por completo IPv6 y mejor soporte para redes inalámbricas. Se implementó el Administrador de Transacciones del Núcleo (*Kernel Transaction Manager*), lo cual les permitió a las aplicaciones trabajar con el sistema de archivos, y con el Registro (o *Registry*), utilizando transacciones atómicas[12]. *User Access Control* o *UAC*, es una tecnología de seguridad que hizo posible que los usuarios utilizaran la computadora con menos privilegios. Cuando una acción requería de privilegios administrativos –tales como instalar o desinstalar software o hacer cambios en la configuración del sistema–, se le pedía al usuario una contraseña administrativa y, si el usuario era un administrador, se le pedía que confirmara la acción.

12. Es decir, una vez iniciada, no se interrumpe hasta que finalice.

Windows Server 2008 fue una versión de sistemas operativos de la línea para servidores de Microsoft Windows, anunciado para manufacturación el 4 de febrero de 2008 y oficialmente liberado el 27 de febrero de ese mismo año. Fue el sucesor de Windows Server 2003 y, como ocurrió con Windows Vista, estaba basado en el núcleo de Windows NT 6.0 Kernel Pack 1.

El desarrollo de Windows 7 se completó el 22 de julio de 2009 y se confirmó su fecha de venta oficial para el 22 de octubre de ese mismo año, junto a su equivalente para servidores Windows Server 2008 R2. Desarrollado para ser usado en PCs de escritorio y hogareñas, equipos portátiles, *tablet PC*, *netbooks* y *media centers*, fue concebido como una actualización incremental y focalizada de Vista y su núcleo NT 6.0 para mantener cierto grado de compatibilidad con aplicaciones y hardware en los que Vista ya era compatible. Es decir que no presentó el salto arquitectónico y de características que tuvo su predecesor Windows Vista respecto de Windows XP. Tiene capacidades multitáctiles[13] y una interfaz rediseñada junto con una nueva Barra de Tareas y un sistema de red doméstica y fácil de usar denominado «Grupo en el Hogar», además de mejoras en el rendimiento general del equipo.

1.4.1 La API de Windows

La interfaz de programación de aplicaciones (API o *Application Programming Interface*) de Windows es la interfaz de programación de sistemas de la familia de los sistemas operativos Microsoft Windows que incluye a Windows 2000, Windows XP, Windows Server 2003, Windows 95, Windows 98, Windows Millennium Edition (ME) y Windows CE. Cada sistema operativo implementa un subconjunto diferente de la API de Windows. Previo a la introducción de las versiones de 64-bit de Windows XP y Windows Server 2003 la interfaz de programación a la versión de 32 bits de los sistemas operativos Windows se llamaba Win32 API para distinguirla de la API de Windows original de 16 bits, que fue la interfaz de programación a la versión original de 16 bits de Windows. En este libro, el término «API de Windows» o *Windows API* se referirá a la interfaz de 32 bits de Windows 2000 y a las interfaces de programación de 32 y 64 bits a Windows XP y Windows Server 2003.

Según M. E. Russinovich y D. A. Solomon (2004), la API de Windows consiste de miles de funciones que están dividida en las siguientes categorías principales:

- Servicios Básicos
- Servicios de Componentes
- Servicios de Interfaz de Usuario
- Servicios de Gráficos y Multimedios
- Mensajería y Colaboración
- Redes
- Servicios Web (*Web Services*)

13. Pantalla táctil o *touchpad* que reconoce simultáneamente múltiples puntos de contacto.

1.5 Trabajos prácticos

1.5.1 Práctica con Linux

Para las prácticas en Linux, hemos elegido a la distribución Ubuntu[14]. Se trata de un sistema GNU/Linux[15] instalado en el CD-ROM. Es decir que no es necesario que se instale en el disco rígido de la computadora. Esto facilita enormemente las prácticas, ya que instalarlo en el disco rígido requiere mayor conocimiento acerca del uso del este sistema y esto se encuentra fuera del alcance de la materia. Existen varias distribuciones de GNU/Linux de este estilo (*Live CD*, en inglés o Disco Compacto Vivo) y esta es, tal vez, la más conocida y divulgada; además podemos agregar que tiene una muy buena detección del hardware, lo que la hace muy adecuada para utilizarla en una amplia gama de Computadoras Personales.

No obstante, si usted ya tiene una computadora con alguna distribución GNU/Linux instalada, úsela sin dudarlo, ya que la desventaja[16] que presenta esta distribución es que utiliza la memoria RAM como si fuera un disco rígido; por lo tanto, los cambios a los archivos de configuración y la creación de nuevos archivos son volátiles, es decir que no permanecerán después de un reinicio de la computadora.

Por otra parte, si no tiene experiencia en el uso y administración de un sistema GNU/Linux, con esta distribución corre pocos riesgos de arruinar su disco rígido o el sistema operativo que tenga en él (supuestamente Windows).

Ubuntu es un CD-ROM con arranque (*boot*), es decir que para cargarlo debe asegurarse de que su computadora es capaz de arrancar un sistema operativo desde la unidad de CD-ROM. Verifique esto en el *setup* de la BIOS y establezca como primer dispositivo de arranque a la unidad de CD-ROM (ver subsección 1.3.2).

Al arrancar Ubuntu verá una primera pantalla para la elección del lenguaje, que por omisión está en inglés. Tiene a su disposición una ayuda presionando la tecla F1; con la tecla F2 vuelve a la pantalla de elección del lenguaje; con la tecla F3 puede cambiar la distribución del teclado –que por omisión es la de Estados Unidos (US)–; con la tecla F4 puede cambiar los «modos» de arranque de «Normal» a «Modo gráfico seguro», en el que hace menos detección automática de hardware, «Use driver update CD», «OEM install (for manufacturers)».

Con la tecla F5 tiene opciones para lograr mejor accesibilidad. Estas opciones son: «Ninguna», «Alto contraste», «Lupa», «Lector de pantalla», «Terminal Braille», «Modificadores de teclado» o «Teclado en pantalla». Con la tecla F6 tiene otras opciones que son las que se le pueden pasar como parámetro al núcleo.

Al arrancar Ubuntu en «Modo Normal», aparece el logo de Ubuntu y una barra de desplazamiento mientras lee el CD; puede tardar unos minutos, dependiendo de la velocidad del lector de CD y la de su procesador. Finalmente aparece la pantalla de ingreso (*login*), en la que le pide ingresar el nombre de algún usuario; si no lo hace en unos pocos segundos, ingresa como usuario `ubuntu`, que es la opción por omisión y se trata de un usuario sin privilegios administrativos en el sistema.

14. `http://www.ubuntu.com`
15. Visite `http://www.gnu.org` y `http`
16. Su gran ventaja es

En la parte superior de su Escritorio tiene una barra de menúes descolgables y algunos íconos. Las opciones principales de estos menús son: «Aplicaciones», «Lugares» y «Sistema».

Tenga en cuenta que al ejecutarse desde el CD-ROM, que es mucho más lento que un disco rígido, muchas veces tendrá la sensación de que el sistema está «congelado» o «colgado» (como se dice en la jerga) y no es así. Por lo tanto, tenga paciencia y no comience a sacudir el *mouse* frenéticamente ni apretar teclas, tratando de obtener alguna respuesta.

La gran mayoría de las distribuciones Linux proveen varias terminales virtuales o «consolas», que logran imitar a las reales gracias a una combinación de teclas. Por tratarse Linux de un sistema operativo multiusuario, es posible que más de un usuario esté usando el sistema. Se provee de varias consolas desde las cuales puede iniciar sesión; presionando `Ctrl+Alt+F1` ... `F6` puede permutar entre las seis consolas e ingresar con distintos usuarios (o el mismo si lo desea, pero las sesiones serán tratadas como distintas). En el caso particular de Ubuntu, todas estas consolas ya están iniciadas con el usuario `ubuntu`. Luego, si presiona la combinación `Ctrl+Alt+F7`, retorna a la pantalla gráfica.

Necesitaremos tener acceso a una CLI, pero no es estrictamente necesario cambiarnos a las terminales virtuales. Podemos tener acceso a una de ellas dentro de una ventana gráfica. Para ello, en la barra de menúes descolgables vaya a «Aplicaciones», luego a «Accesorios» y de ahí a «Terminal».

1.5.1.1 procfs

En muchos sistemas operativos tipo UNIX se utiliza el sistema de archivos *proc* (*proc filesystem o procfs*), que es un sistema de archivos, generado dinámicamente, que muestra información de procesos. `procfs` es un sistema de archivos que no tiene relación con un dispositivo de almacenamiento por bloques. Está implementado en el núcleo del sistema operativo y permite llevar a «espacio de usuario»[17] datos del núcleo y de los procesos, que, sin la ayuda de `procfs`, sería casi imposible accederlos. Este sistema de archivos está montado (ver 6.3.1) normalmente en el directorio `/proc` y, dado que no contiene archivos reales, no consume espacio de almacenamiento. Este sistema tiene su origen en Killian (1984).

Entonces, podemos ver información acerca de nuestra CPU colocando el comando

```
cat /proc/cpuinfo
```

Dentro de nuestra ventana «Terminal», `cat` es un programa que se utiliza para con**cat**enar archivos, pero utilizado de este modo muestra el contenido de un archivo (de texto) por la pantalla.

Si tiene un procesador con doble núcleo (*hyperthreading* o posterior), seguramente verá información acerca de dos procesadores, nombre, frecuencia en MHz, tamaño de la caché y el valor de BogoMIPS (*Bogus Millions Instructions Per Second*), que es una medida no científica de medir la velocidad del procesador y que es típica del núcleo Linux utilizada para calibrar el *busy-loop* o espera activa (ver van Dorst 1996). Es decir que podríamos calificarlo como «el número de millones de veces por segundo en el que un procesador no puede hacer absolutamente nada». (ver Raymond 1990)

17. Posiciones de memoria accesibles a procesos de usuario, con el procesador en modo usuario.

Para ver información acerca de nuestra memoria, el comando

```
cat /proc/meminfo
```

nos mostrará el estado actual de la memoria del sistema, incluyendo información acerca de la memoria virtual y la caché. Es el principal método que utilizan los programas de aplicación (que operan con el procesador en modo usuario) para obtener información acerca del estado de la memoria. Mayormente, la utilizan los utilitarios que muestran el uso de los recursos del sistema.

Podemos ver información acerca de las IRQs con el comando

```
cat /proc/interrupts
```

que nos muestra el número de la interrupción usada y el nombre de quien la usa.

Los archivos en `/proc`, a excepción de unos pocos, tienen tamaño cero; esto es así porque en realidad el archivo no contiene datos, sino que actúa como un puntero hacia donde residen los datos. Es «una ventana al núcleo».

1.5.1.2 Llamadas a sistema

Podemos utilizar el programa `strace` (*system call trace*) que sigue la traza a las llamadas a sistema y las señales de un proceso, interceptándolas, registrándolas y mostrándolas por pantalla (*stderr, standard error*).

Cada línea mostrada contiene el nombre de la llamada a sistema, seguido por sus argumentos en paréntesis y sus valores retornados. Por ejemplo, si ejecutáramos el comando

```
cat /dev/null
```

debería mostrarnos el contenido de un archivo (de contenido nulo en este caso) pero que al seguirle el rastro a las llamadas a sistema que efectúa con el comando

```
strace cat /dev/null
```

nos muestra las sucesivas llamadas al sistema que va efectuando el proceso `cat`. Podemos destacar entre sus llamadas al sistema, la que carga el ejecutable en memoria `execve(«/bin /cat»...` y la que efectivamente abre el archivo para ver su contenido `open(«/dev/null»)`.

Podemos probar también con un programa simple como `pwd`, que muestra el directorio de trabajo (es decir, el directorio sobre el cual estamos posicionados actualmente) y que logra su objetivo primordial haciendo una llamada a la función `getcwd` (*get current working directory*).

También podemos trazar la ejecución de `ls`, que es el comando utilizado para listar los archivos que existen dentro de un directorio (o carpeta en la terminología de Windows) con el comando

```
strace ls
```

Podemos notar que para listar los archivos y directorios que existen en el directorio actual, abre el archivo «.» que contiene el listado de archivos en este directorio. Esto lo hace con `open(«.»)` y luego mediante sucesivas llamadas a `getdents`, obtiene las entradas del directorio actual y finalmente las muestra por pantalla con `write(1, «archivos»)`.

1.5.1.3 Apagando Linux

Para apagar o reiniciar la computadora haga clic con el *mouse* en el ícono superior derecho del Escritorio. Antes de que se apague o se reinicie por completo será expulsado el CD, para que la próxima vez que se inicie la computadora no vuelva a cargarse Ubuntu.

1.5.2 Práctica con Windows

Para muchas prácticas con Windows vamos a valernos de un grupo de herramientas desarrolladas por Mark Russinovich, quien fundó una empresa llamada Sysinternals y cuyo sitio en Internet es `http://www.sysinternals.com`.

Es cierto que Windows provee de ciertas herramientas para ver información acerca del procesador y de la memoria. Por ejemplo, en Windows XP la información acerca del procesador puede verla en el «Administrador de dispositivos», al cual se accede fácilmente desde la pestaña de «Hardware» en la «Propiedades del sistema», ícono «Mi PC» y «Ver información del sistema».

Pero con los utilitarios de Sysinternals puede ver información mucho más específica para la cual no siempre necesita privilegios de administrador, es suficiente con los privilegios de la cuenta Invitado. Varios de estos utilitarios deben ejecutarse desde una ventana de «Símbolo de sistema», es decir que los utilitarios no se basan en una GUI sino en una CLI.

También contamos con las Herramientas de Soporte de Windows o *Windows Support Tools*, que son unas 40 herramientas útiles para administrar y descubrir problemas en los sistemas Windows. Se las puede instalar ejecutando el programa `setup.exe` en la carpeta `\Support\Tools` en el CD de distribución de Windows.

Por otra parte, contamos con los Equipos de Recursos de Windows (*Windows Resource Kits*) que suplementan a las Herramientas de Soporte de Windows porque agregan herramientas adicionales para la administración del sistema. Se pueden descargar gratuitamente de `http://www.microsoft.com/` buscando *resource kit tools*.

Adicionalmente, necesitaremos para el mejor desempeño de ciertos utilitarios de Sysinternals, a las Herramientas de Depuración de Windows o *Windows Debugging Tools*, que podemos descargar gratuitamente de `http://www.microsoft.com/whdc/devtools/debugging/installx86.mspx` donde encontraremos la última versión para sistemas de 32 bits, o desde `http://www.microsoft.com/whdc/devtools/debugging/install64bit.mspx` si queremos la última versión para un sistema de 64 bits. Las primeras normalmente quedan instaladas en `\Archivos de programa \Debugging Tools for Windows (x86)`.

Además, en algunos casos, podemos compilar sencillos programas que se ofrecen tanto para Linux como para Windows. Como ya sabemos, la distribución Ubuntu que hemos elegido trae el compilador libre gcc. Para Windows también tenemos compiladores libres (o al menos gratuitos) del lenguaje C. Por ejemplo, el compilador Borland en Windows. He aquí las instrucciones para dejarlo operativo: `http://www.wikihow.com/Set-up-Borland\%27s-Free-C-Compiler-for-Windows`

1.5.2.1 ProcFeatures

\Por ejemplo, el utilitario `ProcFeatures` de `Sysinternals` (basado en CLI) utiliza la API `IsProcessorFeaturePresent` de Windows para determinar si el procesador y Windows soportan distintos rasgos, tales como páginas de no-ejecución (*No-Execute pages*), extensiones de direcciones físicas (*Physical Address Extensions o PAE*) y un contador de ciclo de tiempo real. Su propósito principal es identificar la versión de PAE del núcleo que ejecuta el sistema y si soporta la protección de desborde del *buffer* de no-ejecución. Además, nos muestra el tipo de procesador (marca y modelo), la frecuencia en MHz, si soporta el conjunto de instrucciones MMX[18] y SSE –o *Streaming SIMD Extensions*– de Intel.

1.5.2.1.1 Ejemplo 1

```
C:\Archivos de programa\Sysinternals>ProcFeatures
Process Feature v1.10
Copyright (C) 2005 Mark Russinovich
Sysinternals - www.sysinternals.com
             Intel(R) Pentium(R) 4 CPU 3.00GHz
x86 Family 15 Model 4 Stepping 1, GenuineIntel
No Execute Protection:              N
Physical Address Extensions (PAE):  N
Floating point emulation:           N
Pentium Floating point errata:      N
RDTSC (Cycle counter):              Y
MMX Instruction Set:                Y
3D Now Instruction Set:             N
SSE Instruction Set:                Y
SSE2 Instruction Set:               Y
C:\Archivos de programa\Sysinternals>
```

1.5.2.1.2 Ejemplo 2

```
C:\Archivos de programa\Sysinternals>ProcFeatures
Process Feature v1.10
Copyright (C) 2005 Mark Russinovich
Sysinternals - www.sysinternals.com
AMD Athlon(tm) 64 X2 Dual Core Processor 5200+
x86 Family 15 Model 107 Stepping 2, AuthenticAMD
No Execute Protection:              Y
Physical Address Extensions (PAE):  Y
Floating point emulation:           N
Pentium Floating point errata:      N
RDTSC (Cycle counter):              Y
MMX Instruction Set:                Y
3D Now Instruction Set:             Y
SSE Instruction Set:                Y
SSE2 Instruction Set:               Y
C:\Archivos de programa\SysInternals>
```

18. Conjunto de instrucciones SIMD (*single instruction, multiple data*) de Intel.

1.5.2.2 CoreInfo

`CoreInfo` es un utilitario de línea de comando que nos muestra la equivalencia entre procesadores lógicos y procesadores físicos, nodo NUMA y el *socket* en el cual residen, como así también la caché asignada a cada procesador. Utiliza la función de Windows `GetLogicalProcessorInformation`[19] para obtener esta información y mostrarla por pantalla. `CoreInfo` es útil para obtener una visión dentro del procesador y la topología de caché de su sistema.

1.5.2.2.1 Ejemplo 1

```
C:\Archivos de programa\Sysinternals>coreinfo
Coreinfo v1.00 -Dump information on CPU cores, NUMA nodes, sockets and caches
Copyright (C) 2008 Mark Russinovich
Sysinternals - www.sysinternals.com
Logical to Physical Processor Map:
**  Physical Processor 0 (Hyperthreaded)
Logical Processor to Socket Map:
Logical Processor to NUMA Node Map:
**  NUMA Node 0
Logical Processor to Cache Map:
C:\Archivos de programa\Sysinternals>
```

1.5.2.2.2 Ejemplo 2

```
C:\Archivos de programa\Sysinternals>Coreinfo
Coreinfo v1.00 - Dump information on CPU cores, NUMA nodes, sockets and caches
Copyright (C) 2008 Mark Russinovich
Sysinternals - www.sysinternals.com
Logical to Physical Processor Map:
*-  Physical Processor 0
*  Physical Processor 1
Logical Processor to Socket Map:
**  Socket 0
Logical Processor to NUMA Node Map:
**  NUMA Node 0
Logical Processor to Cache Map:
*-  Data Cache          0, Level 1,   64 KB, Assoc   2, LineSize  64
*-  Instruction Cache   0, Level 1,   64 KB, Assoc   2, LineSize  64
*-  Unified Cache       0, Level 2,  512 KB, Assoc  16, LineSize  64
-*  Data Cache          1, Level 1,   64 KB, Assoc   2, LineSize  64
-*  Instruction Cache   1, Level 1,   64 KB, Assoc   2, LineSize  64
-*  Unified Cache       1, Level 2,  512 KB, Assoc  16, LineSize  64
C:\Archivos de programa\Sysinternals>
```

1.5.2.3 Llamadas a sistema

`StraceNT` es un trazador de llamadas a sistema (*System Call Tracer*) para Windows. Provee una funcionalidad similar a la que provee `strace` para Linux. Puede trazar todas

19. http://msdn.microsoft.com/en-us/library/ms683194.aspx

las llamadas hechas por un proceso a las funciones importadas desde otras DLLs. `StraceNT` puede ser muy útil para depurar y analizar el funcionamiento interno de un programa. Muestra nombres de funciones, sus argumentos y valores devueltos. Para hacer el trazado de llamadas a funciones `StraceNT` se vale de la sección IAT (*Import Address Table*) del ejecutable PE que veremos en detalle en 3.2.2.3. Lo podemos obtener de: `http://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissasemblers/Strace2ONT.shtml`

`Texe` es una herramienta que podemos utilizar para visualizar las Tablas de Importación y Exportación de los ejecutables con formato PE. Lo podemos obtener de: `http://texe.codeplex.com`

1.5.2.4 Administrador de tareas

El Administrador de tareas o *Task Manager* de Windows nos permite ver la actividad de las aplicaciones en modo usuario y el efecto en modo núcleo. En la pestaña «Rendimiento» y luego seleccionando del menú «Ver» la opción «Mostrar cronología del núcleo». La barra de uso de la CPU mostrará el uso total de la CPU en un color (por ejemplo verde) y el tiempo en modo núcleo en otro (por ejemplo rojo) para atender las llamadas a sistema.

1.5.3 Una analogía

Como para que se imagine una llamada a sistema podría pensar en los servicios de un hotel. Normalmente cuando está en su casa puede abrir la heladera, prepararse comida, puede lavar su ropa y circular normalmente por cualquier lugar de su casa. Sin embargo, si está alojado en un hotel, por muy buen servicio que le presten, usted no puede ir a abrir la heladera, prepararse comida y lavar su ropa. Debe solicitar esos servicios al personal del hotel, quienes lo harán por usted. Sólo ellos tienen acceso a ciertas regiones del edificio a las que usted no puede acceder, tal como ocurre con un sistema operativo que tiene acceso a ciertas regiones de la memoria donde no tienen acceso los procesos de usuario (aun tratándose de usuarios con privilegios, como `root` de UNIX o `administrador` de Windows). La llamada a sistema es la «ventanilla» o «mostrador» al que usted se debe acercar para solicitar el servicio.

Procesos

2

Toda actividad en un sistema de computación está basada en la ejecución de las instrucciones en los programas de computadoras. El término «proceso» se refiere al conjunto de ejecución de instrucciones de máquina. Las instrucciones del programa en sí mismas son una entidad estática. Existen en un papel, en un archivo o, incluso, en una idea en la mente de un individuo. Un proceso es ese conjunto de instrucciones al que se le dio vida, una entidad dinámica que realiza acciones tal como está prescripto en el código; un locus de control dentro de una secuencia de intrucciones, según las palabras de Dennis y Van Horn (1966), esa entidad abstracta que se mueve a través de las instrucciones de un procedimiento a medida que el procedimiento es ejecutado por el procesador.

«Un proceso es un programa en ejecución» (Saltzer 1966).

Las diferencias entre un programa y un proceso se pueden observar en el cuadro 2.1.

Existe, a su vez, otra subclasificación de los programas que es la de aplicación (o software de aplicación) que son los programas o grupos de programas diseñados para ser utilizados directamente por los usuarios (también llamados usuarios finales) y el software de sistemas

Programa	Proceso
Es estático.	Es dinámico.
No tiene contador de programa.	Tiene un contador de programa.
Existe desde que se edita hasta que se borra.	Su ciclo de vida comprende desde que se lo «dispara» hasta que termina.

Tabla 2.1

que son los programas que interactúan de forma más directa con el sistema operativo, tales como los compiladores, depuradores y ensambladores.

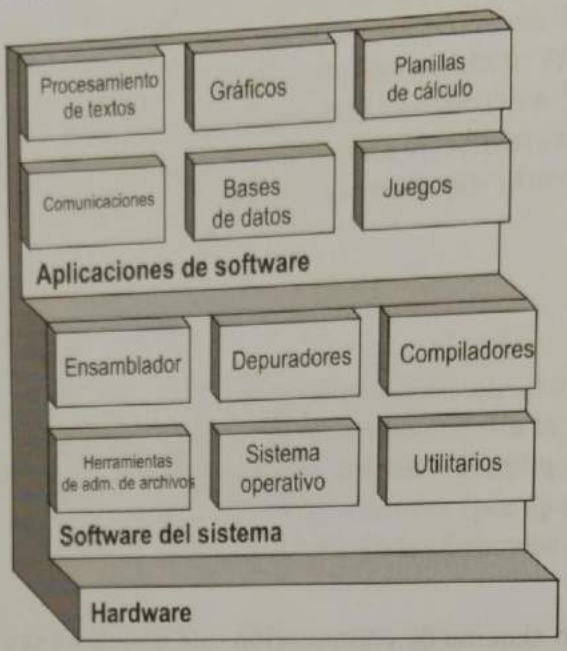


Fig. 2.1 – Software de aplicación y de sistemas.

En la figura 2.1 podemos observar cómo se apoyan las aplicaciones sobre el software del sistema porque, figurativamente hablando, no pueden ejecutar sin él. Pero para nuestro estudio a todo le llamaremos programas.

Los tipos de llamadas al sistema referidos a los procesos, que un sistema operativo normalmente ofrece, son:

- Finalizar, abortar.
- Cargar, ejecutar.
- Crear proceso, terminar proceso.
- Obtener atributos de proceso, establecer atributos de proceso.
- Esperar un lapso.
- Esperar suceso, indicar la ocurrencia del suceso.
- Asignar y liberar memoria.

Asociados a un proceso encontramos su código, sus datos, los recursos operativos asignados a él y uno o más «hilos» (*threads*) de ejecución.

> Un hilo es un flujo de ejecución[1] a lo largo del código del proceso, con su propio contador de programa, registros y pila. Pueden compartir el mismo espacio de direcciones, es decir que pueden examinar y modificar las mismas variables.

1. Es decir, con su propio lazo *fetch/decode/execute*

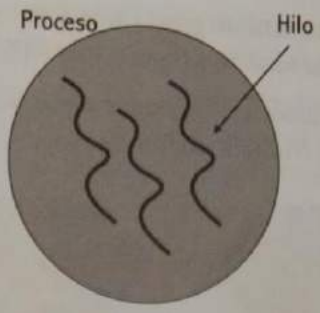


Fig. 2.2 – Hilos.

Podríamos representarlos gráficamente como se muestra en la figura2.2. Según Saltzer (1966) el término «hilo» fue acuñado por Victor Vyssotsky, y sugiere el concepto abstracto encarnado en el término «proceso», si bien no cita alguna referencia que avale su afirmación.

Si bien usualmente comparten el mismo espacio de direcciones, no necesariamente deben hacerlo, a veces se solapan, según Aas (2005). Un ejemplo de proceso con múltiples hilos podemos encontrarlo en los *browsers* para WWW. Puede existir un hilo para manejar los eventos en la interfaz de usuario, tales como detener la carga de una página; puede existir otro para manejar las transacciones con la red, y otro hilo para desplegar o «dibujar» (*render*) la página. Otro ejemplo lo podemos encontrar en un procesador de textos que va revisando la ortografía a medida que escribimos; este corrector ortográfico puede ser un hilo dentro de la aplicación.

Los sistemas operativos pueden clasificarse en tres tipos de sistemas:

- De un único proceso con un único hilo.
- De varios procesos con un único hilo.
- De múltiples procesos con varios hilos.

La disposición más simple es la de un único proceso con un único hilo, porque puede ejecutar sólo un proceso a la vez. Una vez que un proceso haya sido asignado a la CPU, ejecutará hasta finalizar antes de que otro proceso pueda ejecutar. DOS es un ejemplo de un sistema operativo de un único proceso con un único hilo.

2.1 Estados de un proceso

Decíamos con anterioridad que un proceso es dinámico; a medida que se ejecuta, cambia de estado. El estado de un proceso está definido en parte por la actividad actual de ese proceso. Cada proceso puede estar, mínimamente, en uno de los tres siguientes estados:

Ejecutándose (*running*): un proceso que está ejecutando en una CPU. Si un sistema tiene *n* CPU puede tener como máximo *n* procesos en ese estado.

Bloqueado (*blocked*): el proceso está esperando que ocurra algún evento, como por ejemplo la finalización de una operación de entrada o salida.

Listo (*ready*): un proceso que no está asignado a la CPU pero está listo para ejecutar. Un proceso listo podría ejecutar si se lo asigna a una CPU.

de acuerdo con el diagrama de transición de estados que podemos observar en la figura 2.3, correspondientes a los estados de Multics, según Vyssotsky, Corbató y Graham (1965).

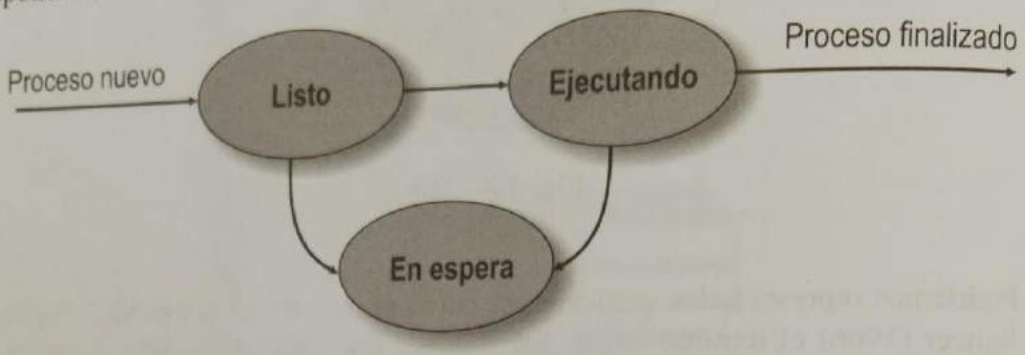


Fig. 2.3 – Diagrama de transición de estados en Multics.

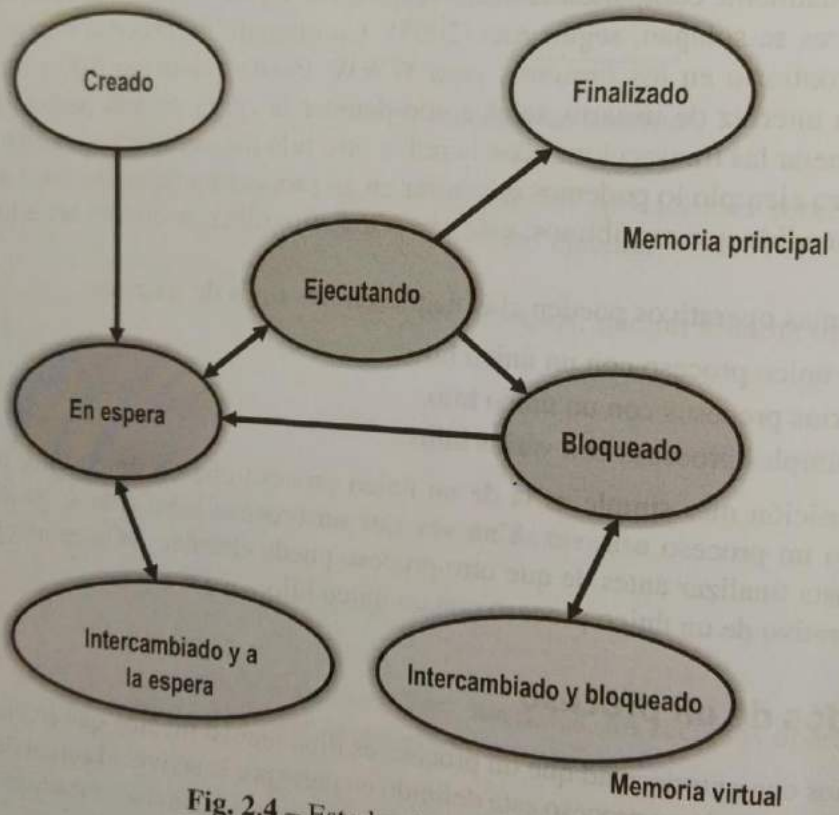


Fig. 2.4 – Estados de un proceso.

Según Stallings (2008), considerando a los sistemas operativos modernos, deberíamos definir además a los estados:

Nuevo (*new*): también llamado «creado» justamente porque el proceso se está creando, es- pera admisión al estado «listo».

Listo (*ready*): también llamado «esperando», ha sido cargado en memoria principal y espera que el despachador lo ponga a ejecutar. Puede haber varios procesos en este estado.

Bloqueado (*blocked*): un proceso que está esperando que ocurra un evento antes de que pueda continuar. Con frecuencia, este evento es la finalización de una operación de entrada o salida. Los procesos bloqueados no requieren de los servicios de la CPU, ya que su ejecución no puede continuar hasta que el evento bloqueante haya finalizado.

Terminado (*terminated*): un proceso que ha detenido su ejecución pero el sistema operativo aún mantiene un registro de él (en UNIX, a estos procesos se les suele llamar *zombie*). El sistema operativo puede desear retener información acerca de un proceso terminado por distintos motivos, ya sea porque el proceso tiene una operación de entrada o salida pendiente, porque tiene procesos hijos que aún están ejecutando y el sistema necesita retener información acerca del proceso que los creó o porque tiene un proceso relacionado que en el futuro puede requerir información acerca del proceso finalizado.

Podemos definir dos estados más en los sistemas que cuentan con memoria virtual:

Intercambiado y esperando: también se lo denomina «suspendido y esperando», fue sacado de la memoria principal y almacenado en memoria secundaria (normalmente en el disco rígido).

Intercambiado y bloqueado: o «suspendido y bloqueado»; un proceso que estaba bloqueado también puede pasar a residir en el área de intercambio (*swap*).

2.1.1 Bloque de control de procesos

Cada proceso se representa en el sistema operativo mediante una estructura de datos llamada bloque de control de proceso (PCB de *Process Control Block*), también llamado bloque de control de tarea o estructura de tarea (`task_struct`, en el código fuente de Linux). Este bloque contiene muchos elementos de información asociados a un proceso específico:

- Estado del proceso: el estado puede ser nuevo, en ejecución, bloqueado, listo, etc.
- Contador de programa: el contador indica la dirección de la siguiente instrucción que se ejecutará para este proceso.
- Registros de CPU: el número y el tipo de los registros varían dependiendo de la arquitectura del computador. Los registros incluyen acumuladores, registros índice, punteros de pila y registros de propósito general, así como cualquier información de códigos de condición que haya. Junto con el contador de programa, esta información de estado se debe guardar cuando ocurre una interrupción, para que el proceso pueda continuar correctamente después.
- Información de planificación de CPU: esta información incluye una prioridad del proceso, punteros a colas de planificación y cualquier otro parámetro de planificación que haya.
- Información de gestión de memoria: esta información puede incluir datos tales como el valor de los registros de base y de límite y las tablas de páginas o las tablas de segmentos, dependiendo del sistema de memoria empleado por el sistema operativo.

- Información contable: esta información incluye la cantidad de tiempo de CPU y de tiempo real consumido, límites de tiempo, números de cuenta, números de trabajo o proceso, entre otros.
- Información de estado de entrada o salida: la información incluye la lista de dispositivos de entrada o salida asignados a este proceso, una lista de archivos abiertos, etc.

El PCB sirve como depósito de cualquier información que pueda variar de un proceso a otro, según lo expresan Silberschatz y Galvin (1997). En los sistemas operativos más recientes, que utilizan hilos, se le suele llamar bloque de control de hilo (TCB o *Thread Control Block*). Mantienen una información similar pero suelen agregar un puntero que referencie al PCB que lo contiene.

2.1.2 Creación y terminación de procesos

Los distintos sistemas operativos han desarrollado diferentes formas de crear procesos. En los sistemas tipo UNIX, un proceso crea a otro a través de una llamada al sistema. El creador es el proceso padre y el creado es el proceso hijo. El proceso hijo podría crear otros, generándose una estructura de árbol con vértice en el padre: es una estructura jerárquica.

Un proceso usa recursos tales como tiempo de CPU, archivos, memoria y dispositivos para entrada y salida. Cuando un proceso crea un proceso hijo (o subproceso), este también necesita recursos. Puede ser que el sistema operativo le asigne esos recursos o puede que use un subconjunto de los de su padre. Este último método previene la saturación de un sistema por la cantidad de subprocesos que pueda crear un proceso. En la ejecución ante la creación de un subproceso hay dos alternativas:

- El padre continúa ejecutándose concurrentemente[2] con sus hijos.
- El padre espera que sus hijos hayan terminado para seguir la ejecución.

Un proceso termina cuando ejecuta su última instrucción y solicita al sistema operativo que lo elimine. Puede devolver información a su padre.

Con respecto al espacio de direcciones, si el proceso hijo es un duplicado del proceso padre, entonces su espacio de direcciones también lo será. Esto es lo que ocurre con la llamada al sistema `fork()` (bifurcar). Este mecanismo permite al proceso padre comunicarse fácilmente con su proceso hijo. Ambos procesos continúan su ejecución con la instrucción siguiente al `fork()`, puesto que se ha copiado el contador de programa. El código de retorno que el proceso hijo recibe luego del `fork()` es cero, pero al padre se le devuelve el identificador de proceso, o PID, del hijo.

En algunos sistemas se produce la terminación en cascada: cuando termina un padre, deben terminar todos los hijos. En otras palabras, no puede existir un proceso sin que exista su padre o un proceso no puede sobrevivir a su padre. Observemos que para las tareas de padres e hijos es preciso que los padres puedan identificarlos. Esto se hace a través del PID, que identifica unívocamente a un proceso.

Como ambos procesos continúan su ejecución de forma concurrente, el padre puede esperar a la finalización de su hijo mediante la llamada al sistema `wait()`.

2. Véase subsección 2.3.3.

Pero, normalmente, los procesos hijos son distintos a sus padres; esto lo logran a través de la llamada a sistema `execve()`, después del `fork()`, para reemplazar su espacio de memoria con un programa nuevo. Esta llamada carga un archivo binario (ejecutable) en la memoria, sobrescribiendo la imagen de memoria del programa que contenía la llamada `execve()`, e inicia su ejecución.

El sistema operativo DEC VMS creaba un proceso nuevo, cargaba un programa específico en ese proceso e iniciaba su ejecución. El sistema operativo Microsoft Windows NT maneja ambos modelos: el espacio de direcciones del padre podría duplicarse o el padre podría especificar el nombre de un programa que el sistema operativo cargará en el espacio de direcciones del proceso nuevo.

Un proceso acaba cuando termina de ejecutar su última instrucción y le pide al sistema operativo que lo elimine, utilizando la llamada al sistema `exit()`. El proceso puede devolver datos a su proceso padre, que puede estar esperándolo en un `wait()`. El sistema operativo libera todos los recursos asignados al proceso, incluyendo el espacio de memoria asignado a él, archivos abiertos y *buffers* de entrada y salida.

Un proceso también puede finalizar de forma provocada o puede causar la terminación de otro proceso con una llamada al sistema apropiada. También los usuarios pueden provocar la finalización de los procesos por distintos motivos.

2.1.2.1 Jerarquía de procesos

La secuencia de creación de procesos genera un árbol de procesos. Para referirse a las relaciones entre los procesos de la jerarquía se emplean los términos de padre, hermano o abuelo. Cuando el proceso A solicita al sistema operativo que cree el proceso B, se dice que A es padre de B y que B es hijo de A. Desde esta óptica, la jerarquía de procesos puede considerarse como un árbol genealógico. Algunos sistemas operativos, como UNIX, mantienen de forma explícita esta estructura jerárquica de procesos –un proceso sabe quién es su padre–, mientras que otros sistemas operativos, como el Windows NT, no la mantienen.

> Según M. E. Russinovich y D. A. Solomon (2004), Windows provee una extensión al modelo de procesos llamada «trabajo» (*job*). La función principal de este objeto es permitir que un grupo de procesos sean gerenciados y manipulados como si se tratara de una unidad. Un objeto «trabajo» permite el control de ciertos atributos y provee límites para los procesos asociados al mismo. También registra información contable básica para todos los procesos asociados con el «trabajo» y para todos los que lo estuvieron pero que ya han finalizado. En cierta forma, el objeto «trabajo» compensa la falta de un árbol estructurado de procesos en Windows y hasta es, en cierta manera, más poderoso que el árbol de procesos al estilo UNIX.

2.1.3 Planificador y activador

El planificador (*scheduler*) es un código que forma parte del núcleo del sistema operativo. Entra en ejecución cada vez que se activa el sistema operativo y su misión es seleccionar el proceso que se ha de ejecutar a continuación. El activador (*dispatcher*) también forma

parte del sistema operativo y su función es poner en ejecución el proceso seleccionado por el planificador.

En sistemas con multiprocesamiento, la planificación se usa para determinar a qué proceso se le da el control de la CPU. La planificación puede ser dividida en tres fases: la de largo, la de medio y la de corto plazo. La planificación a largo plazo (también llamada planificación de trabajos) determina qué trabajos o procesos pueden competir por los recursos del sistema. Típicamente, una vez que el planificador de trabajos activa un trabajo, se mantiene activo hasta que termina. El objetivo principal de este planificador es proveer al planificador de mediano plazo de un apropiado número de trabajos. Con pocos trabajos, la CPU estará muy ociosa porque todos los trabajos estarán bloqueados. Con muchos trabajos, el sistema de administración de la memoria se sobrecargará y degradará la eficiencia del sistema.

El planificador de mediano plazo o intercambiador (*swapper*) intercambia procesos entre la memoria RAM y un espacio en dispositivos de almacenamiento secundario, generalmente un disco magnético. Cualquier sistema de administración de la memoria que soporte multiprocesamiento puede usar intercambio para permitir que más procesos de los que físicamente caben en la memoria compartan el sistema. Los sistemas de administración de memoria, tales como el paginado, pueden intercambiar todos los procesos de la memoria, sacándolos de la misma, para limitar el número de procesos que compiten por ella.

El planificador de corto plazo selecciona al proceso que será asignado a la CPU y el despachador o activador (*dispatcher*) es el que efectivamente carga el proceso en la CPU y es también el que se encarga de pasar a modo usuario el proceso que esta activando y «saltar» a la dirección de la instrucción que comienza la ejecución del programa.

Si bien acá los distinguimos, en la práctica están tan íntimamente ligados que suelen formar un mismo trozo de código. Generalmente, el planificador de corto plazo asigna la CPU por una cantidad fija de tiempo máximo. Un proceso que debe liberar la CPU luego de agotar su tiempo vuelve al conjunto de procesos, desde los cuales el planificador selecciona el proceso por ejecutar. Ampliaremos este tema en el Capítulo 4.

En los sistemas multihilados existe una variedad de opciones para la planificación a corto plazo. En lugar de planificar procesos, el planificador de corto plazo puede planificar hilos. No obstante, este planificador puede ser también de dos niveles, es decir que puede planificar hilos dentro de la planificación de procesos. Según Harris (2002), puesto que los hilos dentro de un proceso comparten memoria y otros recursos, la sobrecarga requerida para intercambiar entre hilos en un proceso es menor que la sobrecarga que hay al intercambiar procesos (pesados) o hilos de diferentes procesos.

2.1.4 Cambio de contexto

La activación del sistema operativo se realiza mediante el mecanismo de las interrupciones. Cuando se produce una interrupción se realizan las dos operaciones siguientes:

1. Se salva el estado del procesador en el correspondiente PCB.
2. Se pasa a ejecutar la rutina de tratamiento de interrupción del sistema operativo.

Llamaremos **cambio de contexto** (*context switch*) al conjunto de estas operaciones, como vemos en la figura 2.5. El tiempo de conmutación de contexto es exclusivamente gasto extra (*overhead*), porque el sistema no realiza trabajo útil durante la conmutación. Por ser un

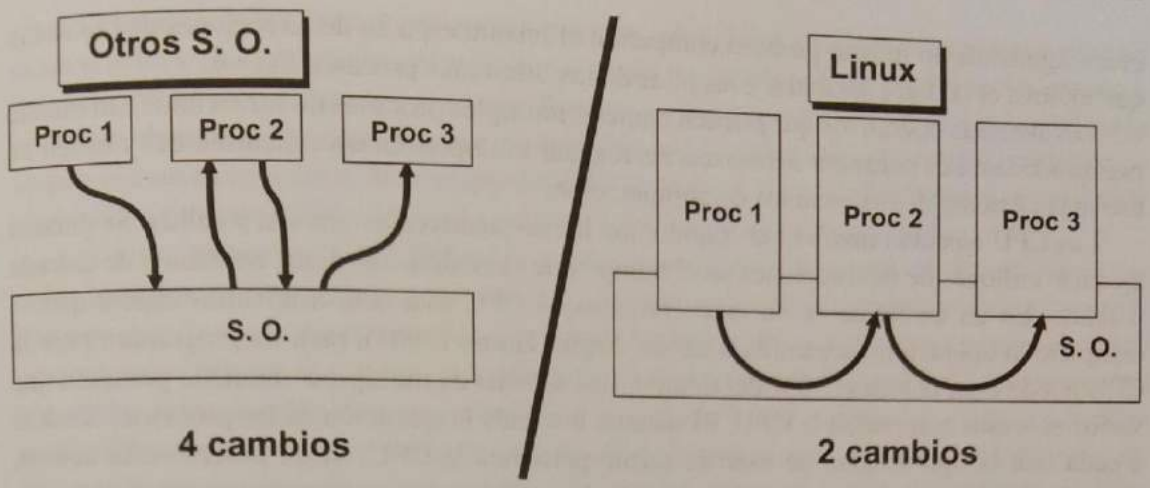


Fig. 2.5 – Cambio de contexto.

cuello de botella tan importante para el desempeño del sistema operativo, se están empleando estructuras nuevas (hilos) para evitarlo hasta donde sea posible. Al tiempo entre detener un proceso y comenzar a correr otro se le llama «latencia [3] de activación» o *dispatch latency*.

El cambio de contexto es el mecanismo que permite que un proceso ceda el control del procesador a otro proceso, retomando más tarde su ejecución por donde la dejó. Esto permite disponer de más procesos que procesadores.

Los microprocesadores de la arquitectura i386 disponen de soporte para multitarea: gestión de estado del procesador por proceso, gestión de mapas de memoria por proceso y mecanismo de cambio de contexto.

En la arquitectura i386 se denomina *task* a cada uno de los hilos de ejecución. Cada una de estas *tasks* será un proceso de Linux.

Desde el punto de vista del procesador, el estado de una tarea es toda la información que el procesador necesita para poder ejecutarla. Cuando la tarea está en ejecución, el estado de la tarea se compone de todos los registros del procesador.

El i386 dispone de instrucciones para guardar y reponer el estado del procesador (estado de la tarea en ejecución) desde memoria principal. Este cambio de estado es lo que se denomina cambio de contexto. Relacione con y diferencie de «Llamadas a sistema», Subsección 1.3.1.7.

2.2 Procesos ligeros, hilos o threads

Un proceso ligero es un programa en ejecución (flujo de ejecución) que comparte la imagen de memoria, y otras informaciones, con otros procesos ligeros. Un proceso puede contener un solo flujo de ejecución, como ocurre en los procesos clásicos, o más de un flujo de ejecución (procesos ligeros). Desde el punto de vista de la programación, un proceso ligero se define como una función cuya ejecución se puede lanzar en paralelo con otras. El hilo de ejecución primario, o proceso ligero primario, corresponde a la función `main`. Todos los pro-

3. Es la medida de una demora temporal experimentada en un sistema.

cesos ligeros de un mismo proceso comparten el mismo espacio de direcciones de memoria, que incluye el código, los datos y las pilas de los diferentes procesos ligeros.

Los sistemas operativos que pueden manejar múltiples procesos de varios hilos son mucho más prácticos. Les permiten a los usuarios realizar múltiples tareas concurrentes[4] y hacen un uso más eficiente de los recursos de computación.

Las CPU ejecutan mucho más rápido que los dispositivos de entrada y salida. Se pueden ejecutar millones de instrucciones en el tiempo que se realiza una única operación de entrada o salida. En un ambiente de un único proceso, la CPU está ociosa mientras espera que se complete la operación de entrada o salida. Según Harris (2002), para un programa típico la CPU puede estar ociosa el 50% del tiempo. Los sistema de multiprocesamiento permiten que varios procesos compartan la CPU. El sistema intercala la ejecución de los procesos, dándole a cada uno la oportunidad de usar de forma periódica la CPU. Si un proceso está ocioso, esperando que finalice una operación de entrada o salida, otro proceso que no esté esperando puede ser asignado a la CPU. Con suficiente memoria y una sintonización cuidadosa, puede lograrse una utilización de la CPU cercana al 100%.

El proceso ligero constituye la unidad ejecutable en Windows NT.

Los hilos son una forma popular de mejorar el desempeño de las aplicaciones a través del (pseudo) paralelismo. En los sistemas operativos tradicionales la unidad básica de utilización de CPU es un proceso. Cada proceso tiene su propio contador de programa, sus propios registros de estado, su propia pila y su propio espacio de direcciones. Por otro lado, en los sistemas operativos con hilos la unidad básica de utilización de CPU es el hilo; en ellos, un proceso está compuesto de un espacio de direcciones y uno o más hilos de control (ver figura 2.6).

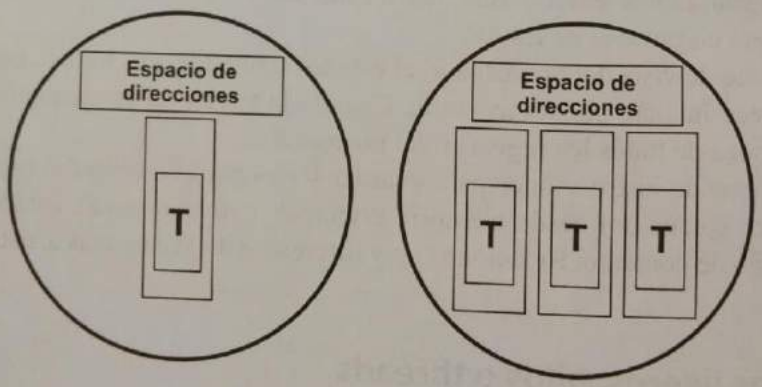


Fig. 2.6 – Procesos mono-hilo y multi-hilo.

Como decíamos, cada hilo de un proceso tiene su propio contador de programa, sus propios estados de registro y su propia pila. Pero todos los hilos de un proceso comparten el mismo espacio de direcciones y, también, las mismas variables globales. Además, comparten el mismo conjunto de recursos del sistema operativo, tales como archivos abiertos, procesos hijos, semáforos, señales, información contable, entre otros. Debido a la compartición del

4. Véase subsección 2.3.3

espacio de direcciones, no hay protección entre los hilos de un proceso. Sin embargo, esto no es un problema. La protección entre los procesos es necesaria porque los diferentes procesos podrían pertenecer a diferentes usuarios. Pero un proceso (y por lo tanto todos sus hilos) le pertenece siempre a un único usuario. Es por ello que la protección entre hilos múltiples de un proceso no es necesaria. Si se requiere protección entre hilos de un proceso, es preferible ponerlos en procesos diferentes, en lugar de colocarlos en un proceso único.

Los hilos comparten una CPU de la misma manera que los procesos. En un uniprocesador los hilos corren en forma casi-paralela (tiempo compartido), mientras que en un multiprocesador de memoria compartida pueden correr tantos hilos en forma simultánea como procesadores haya en el sistema. Además, como los procesos tradicionales, los hilos pueden crear hilos-hijos, pueden bloquearse esperando que complete una llamada a sistema y pueden cambiar de estado durante el curso de su ejecución. En una instancia particular de tiempo, un hilo puede estar en uno de varios estados: ejecutando, bloqueado, listo o terminado. Debido a estas similitudes, los hilos a menudo son vistos como miniprocesos. De hecho, en sistemas operativos con facilidades para hilos, un proceso que tiene un único hilo corresponde a un proceso de un sistema operativo tradicional (ver figura 2.6).

En los procesadores recientes, a partir del Pentium V de Intel, se ha incrementado notablemente el soporte para hilos; además, se ha logrado incorporar más de un núcleo al procesador, lo que los sistemas operativos modernos detectan como si existiera más de un procesador en el ordenador.

A los hilos a veces se los llama **procesos livianos** y a los procesos tradicionales **procesos pesados**.

2.2.1 Motivaciones para el uso de hilos

Las motivaciones principales para el uso de procesos con múltiples hilos de ejecución en lugar de procesos con un único hilo para realizar algunas actividades de computación son:

1. Las sobrecargas involucradas en la creación de un proceso nuevo son, en general, considerablemente mayores que las involucradas en la creación de un hilo nuevo dentro de un proceso.
2. La conmutación entre hilos que comparten el mismo espacio de direcciones es considerablemente más barata que la conmutación entre procesos que tienen sus propios espacios de direcciones.
3. Los hilos permiten que el paralelismo sea combinado con la ejecución secuencial y llamadas a sistema bloqueantes, según Tanenbaum y Woodhull (1998). El paralelismo mejora el desempeño y las llamadas a sistema bloqueantes hacen más fácil la programación.
4. El compartir recursos puede lograrse de manera más eficiente y naturalmente entre hilos de un proceso que entre procesos, porque todos los hilos de un proceso comparten el mismo espacio de direcciones.

Dijimos que las sobrecargas involucradas en la creación de un proceso nuevo y en la construcción de su ambiente de ejecución tienden a ser mucho más grandes que crear un hilo nuevo dentro de un proceso existente y esto es así principalmente porque, cuando se crea un proceso nuevo, su espacio de direcciones debe crearse desde cero, a pesar de que una parte

de él puede heredarse del proceso padre. Sin embargo, cuando se crea un hilo nuevo, usa el espacio de direcciones de su proceso, que no necesita crearse desde cero. Por ejemplo, en el caso de un sistema de memoria virtual con soporte en el núcleo, un proceso nuevo incurrirá en fallos de páginas a medida que se vaya haciendo referencia (solicitando) a datos e instrucciones. Además, la caché hardware inicialmente no contendrá valores para el proceso nuevo y las entradas de la caché para los datos del proceso se crearán a medida que el proceso ejecute. Estas sobrecargas pueden ocurrir también en la creación de hilos, pero tienden a ser menores. Esto es porque cuando el hilo nuevo accede a código y datos que han sido accedidos recientemente por otros hilos dentro del proceso, automáticamente toma ventaja de cualquier *caching* en hardware o memoria principal que haya tenido lugar.

Los hilos también minimizan el tiempo de cambio de contexto, permitiendo a la CPU conmutar desde un hilo de ejecución a otro con mínima sobrecarga. Debido a la compartición del espacio de direcciones y otros recursos del sistema operativo entre los hilos de un proceso, la sobrecarga involucrada en la conmutación de CPU entre hilos pares es muy pequeña comparada con la conmutación de CPU entre procesos que tienen sus propios espacios de direcciones. Esta es la razón por la que los hilos se llaman procesos livianos.

Para clarificar cómo los hilos permiten que el paralelismo sea combinado con la ejecución secuencial y llamadas al sistema bloqueantes, consideremos las diferentes maneras en las cuales un proceso servidor puede construirse. Uno de los siguientes tres modelos puede usarse para construir un proceso servidor (consideremos el caso de un servidor de archivos), según Tanenbaum y Woodhull (1998):

1. **Como un proceso de un solo hilo**: este modelo usa llamadas al sistema bloqueantes pero sin ningún paralelismo. En este método, el servidor de archivos obtiene un pedido de acceso al archivo del cliente desde la fila o cola de pedidos, revisa los permisos de acceso del pedido y, si se le permite el acceso, revisa si es necesario un acceso al disco para servir el pedido. Si no es necesario el acceso al disco, el pedido es atendido inmediatamente y se envía una respuesta al proceso cliente. Si no es así, el servidor de archivos envía un pedido de acceso al disco al servidor de disco y espera una respuesta. Después de recibir la respuesta del servidor de disco, responde el pedido del cliente, envía una respuesta al proceso cliente y regresa a obtener el próximo pedido desde la fila o cola de pedidos.
   En este método, la programación del proceso servidor es simple a causa del uso de la llamada a sistema bloqueante; después de enviar su pedido, el servidor de archivos se bloquea hasta que reciba una respuesta desde el servidor de disco. Sin embargo, si se usa una máquina dedicada para el servidor de archivo, la CPU permanece ociosa mientras que el servidor de archivos está esperando una respuesta desde el servidor de disco. Por lo tanto, en este método no se logra paralelismo y se procesan menos solicitudes de clientes por unidad de tiempo.
   El desempeño de un servidor implementado como un proceso de un solo hilo a menudo es inaceptable. Por lo tanto, es necesario solapar la ejecución de múltiples solicitudes de clientes, permitiendo que el servidor de archivos trabaje sobre varios pedidos simultáneamente.
2. **Como una máquina de estados finitos**: este modelo soporta paralelismo pero con llamadas a sistema no bloqueantes. En este método, el servidor se implementa como

un proceso de un hilo único y es operado como una máquina de estados finitos. Se mantiene una cola de eventos, en la cual están en cola los mensajes de pedidos de los clientes y los mensajes de pedidos desde el servidor de disco. Cada vez que el hilo está ocioso, toma el próximo mensaje desde la cola de eventos. Si es un mensaje de pedido de un cliente, se controla el permiso de acceso y la necesidad de acceso al disco. Si es necesario el acceso al disco para atender el requerimiento, el servidor de archivo envía un mensaje de pedido de acceso al disco al servidor de disco. Sin embargo, esta vez, en lugar de bloquearse, registra el estado actual del pedido del cliente en una tabla y luego va en busca del próximo mensaje desde la cola de eventos. Este mensaje puede ser un pedido desde un cliente nuevo o una respuesta desde el servidor de disco de un pedido de acceso de disco previo. Si es un pedido de cliente nuevo, se procesa como fue descripto. Por otra parte, si es una respuesta desde el servidor de disco, el estado del pedido del cliente que corresponda a la respuesta se trae desde la tabla y el pedido del cliente se sigue procesando.

A pesar de que este método logra paralelismo, es difícil de programar el proceso servidor en este método, debido al uso de llamadas a sistema no bloqueantes. El proceso servidor debe mantener las entradas para cada solicitud de cliente pendiente y, cada vez que se completa una operación de disco, debe recuperarse el trozo apropiado de estado del cliente para saber cómo continuar llevando a cabo el pedido.

3. **Como un grupo de hilos**: este modelo soporta paralelismo con llamadas a sistema bloqueantes. En este método, el proceso servidor se compone de un hilo **despachador** único y múltiples hilos **trabajadores**. Puede que los hilos trabajadores puedan crearse en forma dinámica cada vez que ingresa un pedido, o puede crearse un reservorio (*pool*) de hilos al momento de comenzar (el proceso) para tratar con tantas solicitudes simultáneas como hilos haya en el sistema. El hilo despachador se mantiene esperando pedidos desde los clientes en un lazo. Cuando llega una solicitud de un cliente, revisa el permiso de acceso. Si tiene permiso, puede crear un hilo trabajador nuevo, o bien, elegir un hilo trabajador desocupado a partir del reservorio (dependiendo de si son creados en forma dinámica o estática) y entregar el pedido al hilo trabajador. Luego se pasa el control al hilo trabajador y el estado del hilo despachador cambia de «ejecutando» a «listo». Ahora el hilo trabajador revisa si es necesario un acceso a disco para este pedido o si puede ser satisfecho desde la caché de bloques, que es compartida por todos los hilos. Si es necesario el acceso al disco, envía una solicitud de acceso a disco al servidor de disco y se bloquea mientras espera una respuesta desde el servidor de disco. En este punto, el planificador será invocado y será seleccionado un hilo para ser ejecutado a partir del grupo de hilos que están en estado listo para ejecutar.

   Este método logra paralelismo a la vez que retiene la idea de procesos secuenciales que hacen llamadas a sistema bloqueantes. Por lo tanto, un proceso servidor diseñado de esta manera tiene buen desempeño y también es fácil de programar.

Hemos visto la motivación para usar hilos en el diseño de procesos servidores. A menudo, hay algunas situaciones en las que los procesos clientes se pueden beneficiar también de la concurrencia hecha posible gracias a los hilos. Por ejemplo, un proceso cliente puede usar un algoritmo «divide y vencerás» para dividir datos en bloques que puedan ser procesados en forma separada. Puede enviar cada bloque a un servidor para ser procesado y, finalmente,

recolectar y combinar estos resultados. En este caso, se puede usar un hilo cliente separado para manejar cada bloque de datos, para interactuar con los diferentes procesos servidores. En forma similar, cuando un archivo sea replicado en múltiples servidores, un hilo cliente separado se puede usar para interactuar con cada servidor. Algunas aplicaciones cliente de interfaces de usuario también se pueden beneficiar al usar hilos, devolviendo la interfaz al usuario mientras transcurre una operación larga. Los procesos cliente que realizan muchas operaciones distribuidas se pueden también beneficiar de los hilos al usar un hilo separado para controlar cada operación.

Finalmente, el uso de hilos está también motivado por el hecho de que un conjunto de hilos que usan un espacio de direcciones compartido es la forma más natural de programar muchas aplicaciones. Por ejemplo, en una aplicación que usa el modelo productor-consumidor, los procesos productor y consumidor deben compartir una memoria *buffer* común. Por lo tanto, programar la aplicación de manera que el productor y el consumidor sean dos hilos del mismo proceso hace más simple el diseño del software.

2.2.2 Modelos de organización

Dependiendo de las necesidades de la aplicación, los hilos de un proceso de la aplicación pueden organizarse de diferentes maneras. Tres de ellas se describen a continuación, según Tanenbaum y Woodhull (1998):

1. **Modelo despachador-trabajadores**: en este modelo, el proceso está compuesto de un hilo despachador único y múltiples hilos trabajadores. El hilo despachador acepta pedidos desde clientes y, luego de examinar el pedido, despacha el mismo a uno de los hilos trabajadores libres para su procesamiento. Cada hilo trabajador trabaja sobre un pedido de cliente diferente. Por lo tanto, pueden procesarse múltiples pedidos de clientes en paralelo. Un ejemplo de este modelo se ve en la figura 2.7(a).
2. **Modelo *pipeline***: este modelo es útil para aplicaciones basadas en el modelo productor-consumidor, en las cuales los datos de salida generados por una parte de la aplicación se usan como entrada de otra parte de la aplicación. En este modelo, los hilos de un proceso se organizan como una tubería o línea encauzada (*pipeline*) de manera tal que los datos de salida generados por el primer hilo se usan para el procesamiento por parte del segundo hilo, la salida del segundo hilo se usa como entrada del tercer hilo, y así sucesivamente. La salida del último hilo de la tubería es la salida final del proceso al cual pertenece el hilo. Un ejemplo de este modelo se muestra en la figura 2.7 (b).
3. **Modelo equipo**: en este modelo todos los hilos se comportan como iguales, en el sentido de que no hay una relación despachador-trabajador para el procesamiento de pedidos de clientes. Cada hilo obtiene y procesa solicitudes de clientes por sí mismo. Este modelo a menudo se usa para la implementación de hilos especializados dentro de un proceso. Es decir, cada hilo del proceso está especializado en atender un tipo específico de pedido. Por lo tanto, pueden manejarse varios tipos de pedidos simultáneamente en el proceso. Un ejemplo de este modelo se muestra en la figura 2.7 (c).

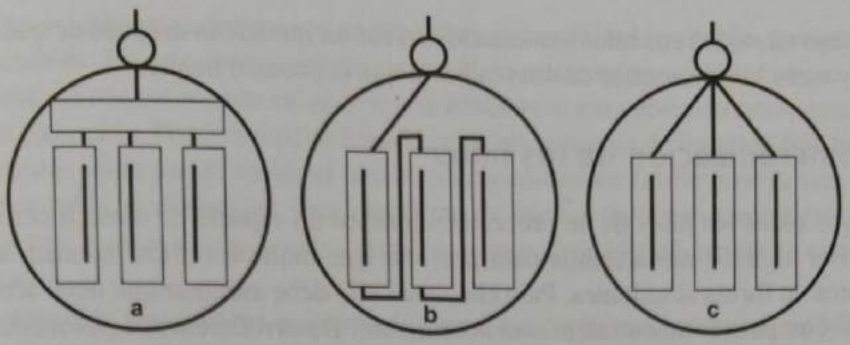


Fig. 2.7 – Modelos de organización de hilos.

2.2.3 Aspectos del diseño de hilos

Un sistema que soporta facilidades para hilos debe proveer a los usuarios un conjunto de primitivas a los usuarios para operaciones relacionadas con ellos. Estas primitivas del sistema forman parte del **paquete de hilos** y algunas se describen a continuación:

2.2.3.1 Creación de los hilos

Los hilos se pueden crear ya sea en forma estática o dinámica. En el enfoque estático, el número de hilos de un proceso permanece fijo durante todo su tiempo de vida. En el enfoque dinámico, el número de hilos de un proceso cambia dinámicamente.

En el enfoque dinámico, el proceso se inicia con un hilo; se crean hilos nuevos cuando se necesitan durante la ejecución del proceso y un hilo puede destruirse a sí mismo cuando finaliza su tarea al hacer una llamada (a sistema) de salida. Por otra parte, en el enfoque estático, el número de hilos de un proceso se decide ya sea al momento de escribir el programa o cuando el programa es compilado. En el enfoque estático, se asigna una pila fija a cada hilo, pero en el enfoque dinámico, el tamaño de la pila de un hilo se especifica como un parámetro pasado en la llamada a sistema para la creación del hilo. Otros parámetros que se requieren usualmente por esta llamada al sistema incluyen la prioridad de planificación y el procedimiento que se ha de ejecutar en este hilo. La llamada a sistema devuelve un identificador de hilo para el hilo recientemente creado. Este identificador se usa en llamadas subsiguientes que lo involucren.

2.2.3.2 Terminación de los hilos

La terminación de los hilos se realiza de manera similar a la terminación de procesos convencionales. Es decir, un hilo puede destruirse a sí mismo cuando finaliza su trabajo al hacer un llamado de salida (`exit()` *call*) o ser finalizado desde el exterior al usar el comando `kill` (en la norma POSIX) y especificando el identificador de hilo como parámetro. En muchos casos, nunca se finalizan los hilos. Vimos que en un proceso que usa hilos creados en forma estática, el número de hilos permanece constante durante todo la vida de un proceso.

En un proceso tal, todos sus hilos son creados en forma inmediata después de que el proceso comienza y estos hilos nunca se destruyen hasta que el proceso finalice.

2.2.3.3 Sincronización de los hilos

Dado que todos los hilos de un proceso comparten un espacio de direcciones en común, se deben usar algunos mecanismos para prevenir que múltiples hilos intenten acceder los mismos datos en forma simultánea. Para ello, cada hilo debe asegurar que tiene acceso exclusivo al dato compartido mientras lo está accediendo. Desarrollaremos estos mecanismos en el capítulo 5.

2.2.3.4 Planificación de los hilos

Otro aspecto importante en el diseño de un paquete de hilos es cómo planificarlos. Los paquetes de hilos a menudo proveen llamadas para dar a los usuarios la flexibilidad de especificar la política de planificación a usarse en sus aplicaciones. Con esta facilidad, un programador de aplicaciones puede usar las heurísticas del problema para decidir la manera más efectiva de planificación. Estas son algunas de las características especiales de la planificación de hilos que pueden estar soportadas en un paquete:

1. **Facilidad de asignación de prioridades**: en un algoritmo de planificación simple, los hilos se planifican sobre una base de «primero en ingresar, primero en salir» (FIFO, de *First In First Out*), o se usa la política de turno circular (*round-robin*) para compartir el tiempo de los ciclos de la CPU entre los hilos sobre una base del «*quantum* por *quantum*»[5], con todos los hilos tratados como iguales por el algoritmo de planificación. Sin embargo, un esquema de planificación de hilos puede proveer a los programadores de aplicaciones la flexibilidad de asignar prioridades a los distintos hilos de una aplicación, con el objeto de asegurar que los importantes puedan ser ejecutados sobre una base de mayor prioridad.
   Un esquema de planificación de hilos basado en prioridades puede ser expulsivo (*preemptive*[6]) o no. En el caso de que no haya expulsión, una vez que se le asigna la CPU a un hilo, este puede usarla hasta que se bloquee, finalice o use todo su *quantum*. Es decir, la CPU no le es arrebatada al hilo al cual ya ha sido asignado, aun si un hilo de mayor prioridad queda listo para correr. El hilo de prioridad más alta se selecciona para correr sólo después de que el hilo que está actualmente usando la CPU lo libere. Por otra parte, en el esquema expulsivo, un hilo de más alta prioridad siempre desaloja a uno de menor prioridad. Esto es, cada vez que un hilo de mayor prioridad queda listo para ejecutar, el hilo de prioridad menor que está corriendo actualmente se suspende, y se le asigna la CPU al hilo de mayor prioridad. En este esquema, un hilo puede correr sólo cuando ningún otro hilo de mayor prioridad esté listo para correr.
2. **Flexibilidad para variar el *quantum* dinámicamente**: un esquema simple de planificación por turno circular asigna un *quantum* de longitud fija para compartir los ciclos de CPU entre los hilos. Sin embargo, un *quantum* de longitud fija no es apropiado en un

5. Es la cantidad de tiempo de procesamiento que se le asigna a un hilo o proceso.
6. En inglés significa desalojar, expulsar o echar a alguien de su vivienda.

sistema multiprocesador porque puede haber menos hilos ejecutables que procesadores disponibles. En este caso, sería un desperdicio interrumpir un hilo con un cambio de contexto al núcleo cuando su *quantum* se acabe sólo para ubicarlo nuevamente en estado de ejecución. Por lo tanto, en lugar de usar un *quantum* de longitud fija, un esquema de planificación puede variar el tamaño del *quantum* en forma inversa con el número total de hilos en el sistema. Este algoritmo da buenos tiempos de respuesta a solicitudes cortas, aun en sistemas muy cargados, pero provee alta eficiencia en sistemas poco cargados.

3. **Planificación forzada**: un esquema de planificación forzada permite a un hilo nombrar a sus sucesores si quiere. Por ejemplo, luego de enviar un mensaje a otro hilo, el hilo transmisor puede liberar la CPU y solicitar que al hilo receptor se le permita ejecutar a continuación. Por lo tanto, este esquema provee la flexibilidad de saltar la cola de hilos ejecutables y directamente conmutar la CPU al hilo especificado por el hilo que ejecuta actualmente. La planificación forzada puede mejorar el desempeño si es usada sabiamente.
4. **Planificación por afinidad**: otra política de planificación que se puede usar para un mejor desempeño en un sistema multiprocesador es la planificación por afinidad. En este esquema, se planifica un hilo a una CPU en la que ejecutó antes con la esperanza de que parte de su espacio de direcciones aún esté en la caché de esa CPU.

2.2.3.5 Manejo de señales

Las señales proveen interrupciones generadas por software y excepciones. Las interrupciones son disrupciones generadas externamente a un hilo o proceso, mientras que las excepciones son causadas por la ocurrencia de condiciones inusuales durante la ejecución de un hilo. Los dos aspectos principales asociados con el manejo de señales en un ambiente con varios hilos son:

1. Una señal debe ser manejada apropiadamente, no importa qué hilo del proceso la haya recibido. Recuerde que en UNIX un manejador de señales debe ser una rutina en el proceso que recibe la señal.
2. Se debe prevenir que se pierdan las señales. Una señal se pierde cuando otra señal del mismo tipo ocurre en algún otro hilo antes que la primera sea manejada por el hilo en el que ocurrió. Esto sucede porque una condición de excepción que está provocando la señal es almacenada en una variable global a nivel de todo el proceso, que es sobrescrita por otra condición de excepción que provoca una señal del mismo tipo.

Un enfoque para el manejo del primer caso es crear un hilo separado manejador de excepciones en cada proceso. En este enfoque, el hilo manejador de excepciones de un proceso dado es responsable de manejar todas las condiciones de excepción que ocurren en cualquier hilo restante de ese proceso. Cuando un hilo recibe una señal de una condición de excepción, envía un mensaje de ocurrencia de excepción al hilo manejador de excepciones y espera hasta que la excepción haya sido manejada. El mensaje de excepción usualmente incluye información acerca de la condición de excepción, el hilo y el proceso que provocaron la excepción. Esto puede involucrar acciones tales como limpiar la excepción, provocando que retome el hilo víctima o que termine el hilo víctima.

Por otra parte, un enfoque para el manejo del segundo caso es asignar a cada hilo sus propias variables globales privadas para el señalamiento de condiciones de excepción, de manera tal que nunca ocurran conflictos entre hilos sobre el uso de tales variables globales. Se dice que tales variables son globales a nivel de hilo (*threadwide global*) porque el código de un hilo normalmente está compuesto de múltiples procedimientos. En este enfoque, se necesitan nuevos procedimientos de biblioteca para crear, establecer y leer estas variables globales a nivel de hilo.

2.2.4 Implementación

Un paquete de hilos puede ser implementado ya sea en espacio de usuario o en el núcleo. En el enfoque de nivel de usuario, el espacio de usuario está compuesto de un sistema en tiempo de ejecución (*runtime*), que es una colección de rutinas de manejo de hilos. Los hilos corren en el espacio de usuario en el tope del sistema de tiempo de ejecución y son manejados por él. El sistema *runtime* también mantiene una tabla de información de estado para seguir la pista del estado actual de cada hilo. Esta tabla tiene una entrada por hilo. Una entrada de esta tabla tiene campos para valores de registro, estado, prioridad y otra información acerca del hilo. Todas las llamadas del paquete de hilos se implementan como llamada a los procedimientos del sistema *runtime* que realiza las funciones correspondientes a las llamadas. Estos procedimientos también realizan conmutación de hilo si el hilo que hizo la llamada tiene que ser suspendido durante la misma. Es decir, se está realizando una planificación de dos niveles en este enfoque. El planificador en el núcleo asigna *quantums* a procesos pesados, y el planificador del sistema *runtime* divide un *quantum* asignado a un proceso entre los hilos del mismo. De esta manera, la existencia de hilos se hace totalmente invisible para el núcleo. El núcleo funciona de una manera similar a un núcleo común que maneja sólo procesos pesados de un único hilo. Este enfoque es el usado por el paquete Lightweight Processes de SunOS 4.1. (ver figura 2.8)

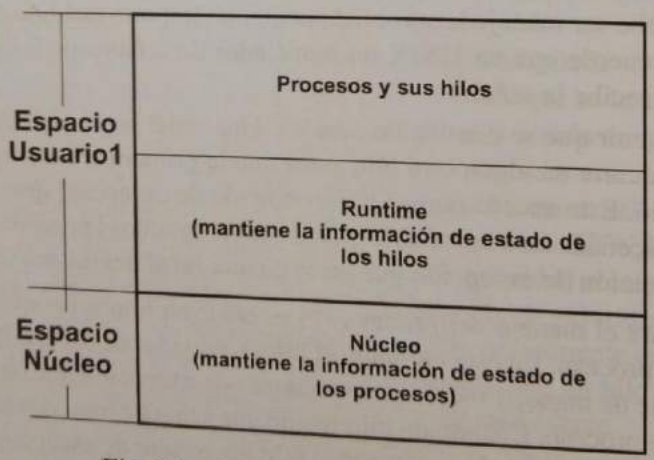


Fig. 2.8 – Hilos en espacio de usuario.

Por otra parte, en el enfoque de nivel de núcleo no se usa un sistema *runtime* y los hilos son manejados por el núcleo. Por lo tanto, la tabla de información de estado de los hilos es

mantenida dentro del núcleo. Todas las llamadas que deberían bloquear a un hilo se implementan como llamadas a sistema que pisan la trampa al núcleo. Cuando un hilo se bloquea, el núcleo selecciona otro hilo para ser ejecutado. El hilo seleccionado puede pertenecer al mismo proceso que el del hilo que estaba ejecutando previamente o a un proceso diferente. De aquí que, la existencia de hilos es conocida por el núcleo y en este enfoque se usa una planificación de un único nivel.

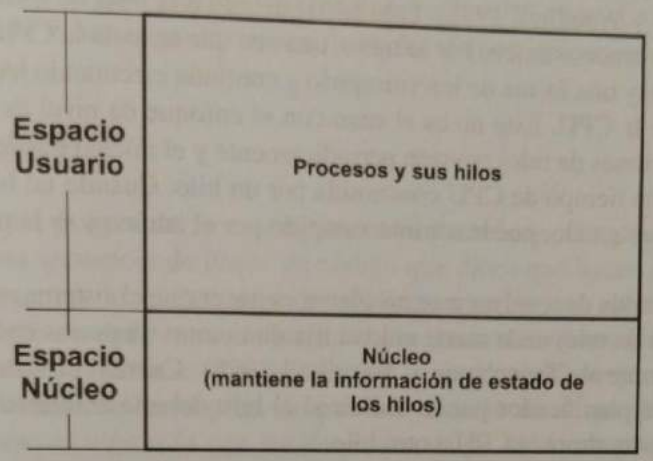


Fig. 2.9 – Hilos en espacio de núcleo.

Las figuras 2.8 y 2.9 ilustran los dos enfoques para la implementación de un paquete de hilos. Las ventajas y desventajas relativas de los dos enfoques se exponen a continuación.

2.2.4.1 Ventajas y desventajas de los dos enfoques

1. La ventaja más importante del enfoque de nivel de usuario es que un paquete de hilos puede implementarse en el tope de un sistema operativo existente que no soporta hilos. Esto no es posible en el enfoque del nivel de núcleo porque en este el concepto de hilos debe estar incorporado en el diseño del núcleo de un sistema operativo.
2. En el enfoque de nivel de usuario, debido al uso de la planificación de dos niveles, los usuarios tienen la flexibilidad de usar sus propios algoritmos personalizados para planificar los hilos de un proceso. Por lo tanto, dependiendo de las necesidades de la aplicación, un usuario puede diseñar y usar el algoritmo de planificación más apropiado para la aplicación. Esto no es posible en el enfoque de nivel de núcleo porque se usa un planificador de un único nivel que está construido dentro del núcleo. Por lo tanto, los usuarios sólo tienen la flexibilidad de especificar a través de los parámetros de las llamadas a sistema las prioridades para asignarse a los distintos hilos de un proceso y seleccionar un algoritmo existente a partir de un conjunto de algoritmos de planificación ya implementados.
3. La permutación de contexto de un hilo a otro es más rápido en el enfoque de nivel de usuario que en el enfoque de nivel de núcleo. Esto es así porque en el primer enfoque el cambio de contexto lo lleva a cabo por el sistema *runtime*, mientras que en el segundo enfoque es necesaria una trampa al núcleo.

4. En el enfoque de nivel de núcleo, la tabla de información de estado para hilos es mantenida dentro del núcleo. Debido a esto, la escalabilidad del enfoque de nivel de núcleo es pobre comparada con la del enfoque de nivel de usuario.
5. Un inconveniente serio asociado con el enfoque de nivel de usuario es que con este enfoque no es posible el uso de una política de planificación por turno circular para compartir los ciclos de CPU entre los hilos sobre una base de *quantum-por-quantum* (Tanenbaum y Woodhull 1998). Esto es así debido a la falta de interrupciones de reloj dentro de un proceso único. Por lo tanto, una vez que se le da la CPU a un hilo para que ejecute, no hay una forma de interrumpirlo y continúa ejecutando hasta que voluntariamente libere la CPU. Este no es el caso con el enfoque de nivel de núcleo, en el cual las interrupciones de reloj ocurren periódicamente y el núcleo puede seguirle la pista a la cantidad de tiempo de CPU consumida por un hilo. Cuando un hilo finaliza de usar su *quantum* asignado, puede ser interrumpido por el núcleo y se le puede dar la CPU a otro hilo.
   Una forma cruda de resolver este problema es hacer que el sistema *runtime* solicite una interrupción de reloj cada cierta unidad fija de tiempo (digamos cada medio segundo) para darle control (Tanenbaum y Woodhull 1998). Cuando el sistema *runtime* toma el control, el planificador puede decidir si el hilo debería continuar corriendo o se le debería asignar ahora la CPU a otro hilo.
6. Otro inconveniente del enfoque del nivel de usuario está asociado con la implementación de llamadas a sistema bloqueantes. En el enfoque de nivel de núcleo la implementación de llamadas a sistema bloqueantes es directa, porque cuando un hilo hace tal llamada, hace una trampa hacia el núcleo, donde es suspendido, y el núcleo inicia un hilo nuevo. Sin embargo, en el enfoque de nivel de usuario, a un hilo no se le debería permitir hacer llamadas a sistema bloqueantes en forma directa. Esto se debe a que si un hilo directamente hace una llamada a sistema bloqueante, todos los hilos de su proceso se detendrán y el núcleo planificará otro proceso para ejecutar. Por lo tanto, el propósito básico para usar hilos se pierde.
   Un enfoque utilizado comúnmente para solucionar este problema es usar rutinas de envoltura (*jacket*). Una rutina de envoltura contiene código extra antes de cada llamada bloqueante al sistema para primero hacer una revisión que asegura si la llamada provocará una trampa al núcleo. La llamada se hace sólo si es segura (no producirá una trampa al núcleo); si no es así, el hilo es suspendido y otro hilo es planificado para correr. El control de la condición de seguridad y el hacer la llamada debe hacerse en forma atómica.

Con cierto éxito, se han hecho pocos intentos de combinar las ventajas de los dos enfoques en el desarrollo de los paquetes de hilos.

2.2.5 Fibras

En Windows, una fibra es una unidad de ejecución que debe ser manualmente planificada por la aplicación. Según M. E. Russinovich y D. A. Solomon (2004), las fibras le permiten a una aplicación planificar sus propios hilos de ejecución, en lugar de atenerse al mecanismo de planificación basado en prioridades proporcionado por Windows. A las fibras también se

las llama «hilos livianos» y, en términos de planificación, son invisibles al núcleo porque están implementadas en modo usuario dentro de `kernel32.dll`. Para usar fibras primero se hace una llamada a la función `ConvertThreadToFiber`. Esta función convierte al hilo en una fibra en ejecución. Luego de eso, la fibra recientemente convertida puede crear fibras adicionales con la función `CreateFiber`. Cada fibra puede, a su vez, tener su propio conjunto de fibras. A diferencia de un hilo una fibra no comienza a ejecutar hasta que sea seleccionada manualmente a través de una llamada a la función `SwitchToFiber`. La fibra nueva ejecuta hasta finalizar y salir o hasta que invoque a `SwitchToFiber`, seleccionando así a otra fibra para ejecutar.

2.3 Comunicación

Decíamos que un **programa** es un conjunto de instrucciones. Es, simplemente, un texto que consiste en una secuencia de líneas de código que dicen qué hacer con un conjunto de datos de entrada para producir algún tipo de salida. Se trata de algo **estático**. Puede compararse con el concepto de **clase** en el ámbito de la Programación Orientada a Objetos (POO). Para que el programa pueda hacer algo de verdad hay que ponerlo en ejecución.

Una primera definición incompleta de **proceso** sería la de un programa en ejecución. Es decir, un proceso es algo más que las líneas de código de un programa. Un proceso es algo **dinámico**. Está representado por el valor del contador de programa, el contenido de los registros del procesador, una pila y una sección de datos que contiene variables globales. Puede compararse con el concepto de **objeto** en el ámbito de la POO.

De igual manera que en POO pueden existir múltiples objetos de una clase determinada, aquí pueden existir múltiples procesos que corresponden al mismo programa. Como ejemplo consideremos un servidor de aplicaciones donde reside una aplicación de navegador de Internet y existen varios usuarios ejecutando ese navegador, cada uno de ellos navegando por un sitio diferente. Cada instancia del programa es un proceso. Cada proceso tendrá su propio contador de programa, así como sus propios registros, pila y variables. Como estos procesos se juntan o **concurren** en el mismo espacio y tiempo y acceden a los mismos recursos, deberemos establecer alguna forma de **sincronización** para que no se estorben, que desarrollaremos en el Capítulo 5.

Es posible clasificar los modos en que interactúan los procesos en función del nivel de conocimiento que cada proceso tiene de la existencia de los demás.

2.3.1 Procesos independientes

Un proceso es **independiente** si no afecta o es afectado por la ejecución de otros procesos en el sistema. Aunque los procesos no trabajen juntos, el sistema operativo tiene que encargarse de la competencia por los recursos.

2.3.2 Procesos cooperantes

Por otra parte, los procesos pueden mostrar cooperación para compartir un objeto común. Cuando dos o más procesos tienen que cooperar uno con el otro, entonces deben estar

conectados, debe haber alguna forma de comunicación entre ellos para poder intercambiar información.

> Un proceso cooperativo es uno que puede afectar o ser afectado por los demás procesos que se ejecutan en el sistema.

Los procesos cooperativos podrían compartir directamente un espacio de direcciones lógico (es decir, tanto código como datos), o bien compartir datos únicamente a través de archivos. El primer caso se lleva a la práctica mediante **procesos ligeros, *threads* o hilos**, que vimos en la sección 2.2. El acceso a datos compartidos puede dar pie a inconsistencia de los datos.

Supongamos que dos procesos están compartiendo datos a través de un archivo. Mientras un proceso está escribiendo en el archivo el otro lo está leyendo. Entonces, el proceso que está leyendo está siendo afectado por el que está escribiendo.

Hay varias razones por las cuales podemos querer que dos o más procesos cooperen:

- Para que compartan información: Varios usuarios pueden estar interesados en la misma información, por ejemplo, un archivo compartido. Debemos proveer un ambiente que permita el acceso a este tipo de recursos.
- Para mejorar la velocidad: Si una tarea en particular necesita ejecutar más rápido puede ser dividida en sub tareas, cada una de las cuales puede ejecutar en paralelo con las otras, si contamos con un sistema con múltiples procesadores.
- Modularidad: Puede ser deseable construir un sistema de forma modular, dividiendo las funciones del sistema en procesos separados.
- Conveniencia: Aún un usuario individual puede tener muchas tareas sobre las que trabajar al mismo tiempo. Por ejemplo, si está editando un programa a la vez que compilándolo, ejecutándolo e imprimiéndolo.

2.3.3 Procesos concurrentes

Según el diccionario de la Real Academia Española, una de las acepciones de la palabra **concurrencia** es:

> «Acaecimiento o concurso de varios sucesos en un mismo tiempo».

Si en esta definición sustituimos la palabra suceso por proceso, ya tenemos una primera aproximación a lo que va a ser la concurrencia en computación.

Dos **procesos** son **concurrentes** cuando la primera instrucción de uno de ellos se ejecuta después de la primera instrucción del otro y antes de la última. Es decir, existe un solapamiento en la ejecución de sus instrucciones. No tienen por qué ejecutarse exactamente al mismo tiempo, simplemente es suficiente con el hecho de que exista un intercalado entre la ejecución de sus instrucciones. Si se ejecutan al mismo tiempo los dos procesos, entonces tenemos una situación de **programación paralela**. La programación concurrente es un paralelismo potencial.

No necesariamente un proceso tiene por qué ser todo el programa en ejecución, sino que puede ser parte de él. Dicho de otra forma, un programa, al ponerse en ejecución, puede dar

lugar a más de un proceso, cada uno de ellos ejecutando una parte del programa. Continuando con el mismo ejemplo, el programa del navegador de Internet puede dar lugar a más de un proceso: uno que controla las acciones del usuario con la interfaz, otro que hace las peticiones al servidor, etcétera.

> Así, una definición más acertada de proceso es la de una **actividad asíncrona**, susceptible de ser asignada a un procesador (Palma Méndez y col. 2003).

Cuando varios procesos se ejecutan concurrentemente pueden existir procesos que colaboren para un determinado fin, mientras que pueden existir otros que compitan por los recursos del sistema. Incluso aquellos procesos que colaboran deberán competir a la hora de obtener tiempo de procesador.

Para llevar a cabo estas tareas de colaboración y competencia por los recursos, se hace necesaria la introducción de mecanismos de **comunicación** y **sincronización** entre procesos.

Obviamente, no todas las partes de un programa pueden ejecutarse de forma concurrente. Consideremos el siguiente fragmento de programa:

```
x=x+1;
y=x+2;
```

Está claro que la primera sentencia se tiene que ejecutar antes que la segunda. Sin embargo, consideremos ahora lo siguiente:

```
x=1;
y=2;
z=3;
```

Queda claro que el orden en que se ejecuten no interviene en el resultado final. Si dispusiésemos de tres procesadores, podríamos ejecutar cada una de las líneas en uno de ellos, incrementando la velocidad del sistema. Por lo tanto, diremos que:

> Dos eventos son **concurrentes** si mediante la observación del programa no podemos establecer cuál ocurrirá primero.

La ejecución de sistemas concurrentes tiene algunas características que los diferencian claramente de los sistemas secuenciales. Destacamos dos: el orden de ejecución de las instrucciones y el indeterminismo. En los programas secuenciales hay un **orden total** en la ejecución de las líneas de código. Ante un conjunto de datos de entrada se sabe siempre por dónde va a ir el programa (su flujo de ejecución). En los programas concurrentes hay un **orden parcial**, ante el mismo conjunto de datos de entrada no se puede saber cuál va a ser el flujo de ejecución. En cada ejecución del programa el flujo puede ir por distintos sitios. Este orden parcial lleva a que los programas concurrentes puedan tener un comportamiento indeterminista, es decir, puedan arrojar diferentes resultados cuando se ejecutan repetidamente sobre el mismo conjunto de datos de entrada.

Implementar adecuadamente concurrencia requiere que se utilicen mecanismos de sincronización (que veremos en el Capítulo 5) y comunicación entre procesos.

2.3.4 Paradigmas de comunicación entre procesos

La comunicación entre procesos (comúnmente IPC, del inglés *Inter-Process Communication*) es una función básica de los sistemas operativos. Las dos maneras básicas de abordar este tema son:

2.3.4.1 Memoria compartida



Fig. 2.10 – El enfoque de datos compartidos.

También llamado original compartido o enfoque de datos compartidos. En este enfoque, la información a ser compartida es ubicada en un área de memoria común –variables compartidas o *buffers*– que es accesible a todos los procesos involucrados en la IPC. El paradigma de datos compartidos ofrece el patrón de comunicación conceptual ilustrado en la figura 2.10. Por ejemplo, los semáforos –que veremos en el apartado 5.4– utilizan este método.

2.3.4.2 Pase de mensajes

También llamado copias compartidas o enfoque de pase de mensajes. En el enfoque de pase de mensajes (o *message passing*, en inglés) la información a ser compartida se copia físicamente desde el espacio de direcciones del proceso remitente a los espacios de direcciones de todos los procesos receptores (destinatarios), y esto se hace mediante la transmisión de los datos a ser copiados en la forma de mensajes.

> Un **mensaje** es un bloque de información, armado por un proceso transmisor de manera tal de tener significado para el proceso receptor.

Los procesos e hilos pueden estar ejecutando en la misma computadora o en dos o más, conectadas mediante una red. El paradigma de pase de mensajes ofrece el patrón de comunicación conceptual ilustrado en la figura 2.11.

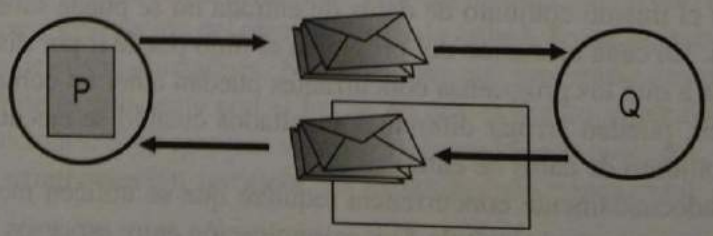


Fig. 2.11 – El enfoque de pase de mensajes.

El pase de mensajes es utilizado en computaciones paralelas, concurrentes y orientadas a objetos. Como los procesos receptores deben esperar la llegada de los mensajes, también es una forma de sincronizarlos.

Hay varias formas de sistemas de pase de mensajes. El sistema operativo provee mínimamente dos primitivas: `send()` y `receive()`, es decir, enviar y recibir.

```
send(destino, mensaje)
receive(origen, mensaje)
```

Un proceso envía información en forma de un `mensaje` a otro proceso designado como `destino`. Mientras que el otro proceso recibe el `mensaje` desde el proceso `origen` invocando la primitiva `receive`.

Tamaño: los mensajes que envía un proceso a otro pueden ser de tamaño **fijo** o **variable**. Es más sencilla la implementación de pase de mensajes de tamaño fijo, pero dificulta la tarea del programador si los mensajes son grandes ya que lo obliga a descomponerlos en varios mensajes de longitud fija más pequeños.

Flujo de datos: de acuerdo al flujo de datos la comunicación puede ser **unidireccional** o **bidireccional**. Un enlace es unidireccional cuando cada proceso conectado a él únicamente puede enviar o recibir mensajes, pero no ambas cosas. Si cada proceso puede enviar o recibir mensajes, entonces el pase de mensajes es bidireccional.

Nombrado: los procesos que utilizan mensajes para comunicarse o sincronizarse deben tener alguna forma de referirse unos a otros. En este sentido, la comunicación puede ser **directa** o **indirecta**. Con la comunicación directa, la primitiva `send()` incluye una identificación específica del proceso de destino. En cambio, si nos dirigimos de forma indirecta, los mensajes no se envían directamente del emisor al receptor, sino a una estructura de datos compartida formada por colas que pueden guardar los mensajes temporalmente. Estas colas se denominan generalmente **buzones** (*mailboxes*, en inglés). De este modo, para que dos procesos se comuniquen, uno envía mensajes al buzón y el otro los lee a partir de él, como se ilustra en la figura 2.12.

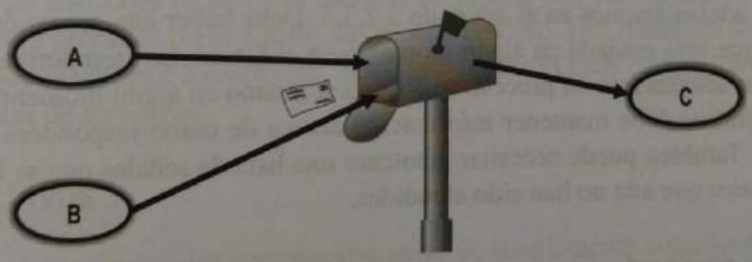


Fig. 2.12 – Buzón.

La relación entre emisores y receptores puede ser de uno a uno, de muchos a uno, de uno a muchos o de muchos a muchos. Una relación uno a uno permite que se establezca un enlace privado de comunicaciones entre dos procesos, lo que aísla su interacción de injerencias erróneas de otros procesos. Una relación de muchos a uno resulta útil para interacciones cliente/servidor, donde un proceso ofrece un servicio a un conjunto de procesos. Además, la asociación de procesos a buzones puede ser estática o dinámica.

2.3.4.3 Puerto

Es un buzón con una relación de muchos a uno, utilizada típicamente en comunicaciones cliente/servidor, como se observa en la figura 2.13. Los puertos suelen estar asociados estáticamente con algún proceso en particular, es decir, el puerto se crea y se asigna al proceso permanentemente, cuando se destruye el proceso, también se destruye el puerto.

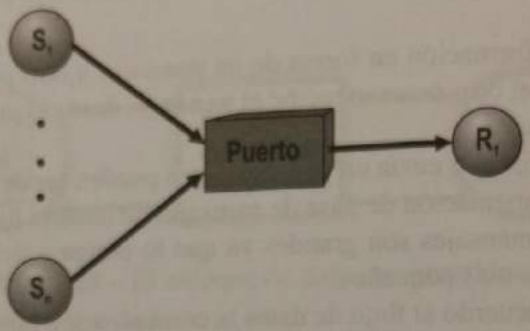


Fig. 2.13 – Relación de muchos a uno con un puerto.

2.3.4.4 Señales

La **señal** es una forma limitada de IPC utilizada en UNIX. Es una notificación asíncronica enviada a un proceso o a un hilo específico dentro del mismo proceso con el objeto de notificarlo de un evento ocurrido. Esta interrupción de software –como la llama R. W. Stevens (2005)– creada por el sistema operativo, que se puede originar con un proceso o con el sistema operativo, y que puede ser enviada a uno o más procesos, interrumpe el flujo de ejecución normal del proceso para entregar la señal. El proceso que recibe la señal puede responder ignorándola, finalizando o ejecutando una función especificada, es decir, un manejador de señales, como adelantábamos en el apartado 2.2.3.5. Debe haber una provisión para planificar una señal que será enviada en algún momento en el futuro. Los temporizadores (*timers*) en software son señales que un proceso se envía a sí mismo en algún momento en el futuro. El sistema operativo debe mantener información acerca de cómo responderá un proceso si recibe señales. También puede necesitar mantener una lista de señales que se le han enviado a un proceso, pero que aún no han sido atendidas.

2.3.4.5 Temporizadores

Un temporizador (*timer interrupt*, en inglés) es un tipo de interrupción de reloj que se puede utilizar para controlar la secuencia de un evento y que interrumpe al procesador al llegar a cero. El sistema operativo mantiene en cada PCB un temporizador que suele estar expresado en segundos. Cada vez que la rutina del sistema operativo que trata las interrupciones de reloj comprueba que ha transcurrido un segundo, decrementa todos los temporizadores que no estén a cero y comprueba si han llegado a cero. Para aquellos procesos cuyo temporizador acaba de llegar a cero, el sistema operativo notifica al proceso que el temporizador

ha vencido. En Linux se lo llama *jiffy*, que es un término informal para indicar un lapso de tiempo súmamente corto, como diríamos «periquete» en castellano.

Como el número de temporizadores en hardware en una computadora o un procesador es finito y limitado, los sistemas operativos y los sistemas embebidos a menudo usan a un único temporizador hardware para implementar a un conjunto extensible de temporizadores en software. En este escenario la rutina de servicio de interrupciones del temporizador de hardware manejaría el *house-keeping* y la administración de tantos temporizadores en software como se necesitasen y el temporizador del hardware expiraría cuando el próximo temporizador de software deba expirar.

2.3.4.6 Socket

El *socket* (zócalo o enchufe) también es una forma de comunicación, generalmente entre procesos remotos ejecutando en computadoras conectadas mediante una red (*network socket*), pero no necesariamente, los procesos pueden residir en la misma computadora. Ampliaremos este tema en el apartado 7.6.

2.3.4.7 Tubería

Una tubería (en inglés, *pipe*)[7] es un mecanismo de comunicación y sincronización. Desde el punto de vista de su utilización, es como un pseudoarchivo mantenido por el sistema operativo.

En POSIX existen tuberías sin nombre, o simplemente *pipes*, y tuberías con nombre, o FIFOS. Un *pipe* no tiene nombre y, por tanto, sólo puede ser utilizado entre los procesos que lo hereden a través de la llamada `fork()`. Retomaremos este tema en el apartado 5.6.3.

2.3.4.8 Cola de mensajes

Las colas de mensajes (o *message queues*, en inglés) son otra forma de comunicación asincrónica entre procesos o entre hilos del mismo proceso. Los mensajes se almacenan en la cola hasta que el receptor los extraiga. Tanto el estándar POSIX como el Win32 ofrecen el servicio de colas de mensajes.

2.3.4.9 Archivos

Dos o más procesos pueden comunicarse entre sí, simplemente accediendo a un archivo por su nombre.

2.3.4.10 Archivos proyectados en memoria

Un archivo proyectado en memoria (*memory-mapped file*, en inglés) es una porción de memoria virtual en la que se establece una correlación directa byte a byte con una parte de un archivo. Las lecturas y escrituras a esa región de la memoria se tratan como lecturas y

7. Se le atribuye el concepto a Doug McIlroy (1964). Ver `http://doc.cat-v.org/unix/pipes`

escrituras al archivo, con lo que se simplifica mucho el uso de este último. Cuando se cierra el archivo, todos los datos que tenían correspondencia en la memoria se escriben en el disco y se borran de la memoria virtual de los procesos.

2.3.5 Excepciones

Cuando un proceso o hilo provoca una división por cero, acceder a una dirección de memoria protegida o inexistente, o intenta ejecutar una instrucción privilegiada desde el modo usuario, se provoca una **excepción**. Así como en el caso de las interrupciones, existe un **manejador de excepciones** que se hará cargo de la situación y también implicará un cambio en el flujo normal de ejecución. Pero a diferencia de las interrupciones, que son externas al proceso, las excepciones son internas a él. No obstante esto, una excepción puede generar una interrupción. Por ejemplo, un fallo de página es una excepción que genera una interrupción. La interrupción suspende el proceso actual y salta al manejador de fallos de página en el núcleo.

2.4 Trabajos prácticos

2.4.1 Práctica con Linux

2.4.1.1 Estados de los procesos

Volvamos a abrir una ventana de Terminal, tal como vimos en la subsección 1.5.1.

UNIX provee el comando «ps» (*process status*) para ver el estado de los procesos. Si coloca este comando en Linux verá básicamente dos procesos: el del intérprete de comandos `bash` y el del propio proceso `ps`. Fíjese en las columnas: la primera indica el número identificador del proceso (PID, de *process identification*). Con la flecha de cursor puede recuperar este comando para colocarlo de nuevo; repita esta operación varias veces, notará que el PID del «`bash`» no cambia, pero sí lo hace el de «`ps`». Esto es así porque «`bash`» es un proceso que sigue ejecutando, en cambio «`ps`» cumple todo su ciclo de vida y cada invocación que hace es, en realidad, un nuevo proceso y, por lo tanto, el sistema operativo le asigna un número nuevo. Relacione esto con lo visto en la subsección_2.1.2.

Cada vez que ejecuta «`ps`» el intérprete de comandos genera un hijo con las llamadas al sistema `fork()` y luego `exec()`, las cuales utilizaremos en un programa en lenguaje C. El comando «`ps`» sin opciones nos muestra lo que ya hemos visto. Si queremos ver todos los procesos que se están ejecutando en el sistema coloquemos el comando

```
ps aux
```

La opción «`a`» significa *all*, es decir, todos los procesos; la opción «`u`» indica «formato de usuario» y la opción «`x`» es para que muestre también a los procesos no asociados a terminal.

Si quiere ver parcialmente el árbol de procesos debería utilizar el comando

```
ps axf
```

También existe el comando `pstree`, que nos muestra un árbol de procesos más completo y más claro a partir del proceso `init`.

Si quiere aprender más acerca de las opciones del comando «ps», consulte las páginas del manual con el comando

```
man ps
```

Podemos utilizar además el comando «top», que nos muestra en forma decreciente los procesos que más recursos consumen, y va actualizando la lista periódicamente. Para salir de este programa presione la tecla «q».

Observe que el orden se va alterando. Es decir, que de la cola de procesos listos para ejecutar, el planificador los va eligiendo por su prioridad, pero esta no es constante en Linux, sino que se ve alterada de acuerdo con un sistema de «créditos», que veremos más adelante.

2.4.1.2 Obtener el PID

Ahora veremos a los principales servicios que ofrece POSIX y Win32 para la gestión de procesos, procesos ligeros y planificación.

Para ello, obtendremos el identificador del proceso mediante la llamada (*system call*) al sistema, cuyo prototipo en lenguaje C es:

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

Desde la misma CLI vamos a ejecutar un comando que nos abrirá una ventana de edición de textos muy intuitiva. Supongamos que vamos a crear el archivo nuevo `obtenerpid.c`:

```
gedit obtenerpid.c
```

en el que cargaremos el siguiente programa:

```
#include <sys/types.h>
#include <stdio.h>
main()
{
pid_t id_proceso;
pid_t id_padre;
id_proceso=getpid();
id_padre=getppid();
printf("Identificador de proceso:%d\n", id_proceso);
printf("Identificador del proceso padre:%d\n", id_padre);
}
```

Una vez hecho esto, guardamos y cerramos la ventana de edición y volvemos al *prompt*. Este programa se compila simplemente así:

```
gcc -o obtenerpid obtenerpid.c
```

suponiendo que así ha nombrado al programa. Y se ejecuta así:

```
./obtenerpid
```

es decir, «punto» «barra» «obtenerproceso» sin espacios.

Si usted alguna vez ejecutó programas en el intérprete de DOS, recordará que simplemente había que colocar el nombre del programa. Los intérpretes de comandos tienen una ruta (*path*) de búsqueda para encontrar los ejecutables. En el caso del DOS, esta ruta incluye el directorio actual (donde está usted parado ahora), pero en el caso de UNIX (Linux en este caso) no se incluye. Por este motivo, si usted crea ejecutables fuera de los directorios asignados para ejecutables (`/bin`, `/sbin`, `/usr/bin`, `/usr/sbin`, `/usr/local/bin`, `/usr/local/sbin`), el intérprete no podrá encontrarlos y le devolverá el error de «comando desconocido». Como ahora usted ha creado un ejecutable en un directorio fuera del *path* de búsqueda de ejecutables (concretamente en `/home/ubuntu`), debe indicarle al intérprete dónde se encuentra el ejecutable, lo que puede hacer de dos maneras:

- **Absoluta**: para indicar la ruta a un archivo (ejecutable en este caso), se especifica su posición desde la «raíz» del sistema de archivos, pasando por los directorios hasta llegar al archivo. Por ejemplo «`/home/ubuntu/obtenerpid`».
- **Relativa**: la posición del archivo a partir del directorio en el cual estamos posicionados actualmente. El punto «.» significa «el directorio actual, donde estoy posicionado actualmente», es decir «`/home/ubuntu`» y, a partir de ahí, continúa con «`/ejemplo`».

2.4.1.3 Crear procesos con fork()

Este es un ejemplo en POSIX de utilización de servicio para la creación de procesos.

Crearemos con `gedit` y luego ejecutaremos el siguiente programa, que podríamos llamar `creaproceso.c`

```
#include <sys/types.h>
#include <stdio.h>
main()
{
pid_t pid;
pid=fork();
switch(pid){
 case -1: /* error en el fork() */
   perror("fork");
   break;
 case 0: /* proceso hijo */
   printf("Proceso%d; padre =%d \n", getpid(), getppid() );
   break;
 default: /* proceso padre */
   printf("Proceso%d; padre =%d \n", getpid(), getppid() );
          }
}
```

De la misma manera, guardamos y cerramos la ventana de edición y volvemos al *prompt*. Este programa se compila con:

```
gcc  -o  creaproceso  creaproceso.c
```

Y se ejecuta así:

```
./creaproceso
```

Observe que el proceso hijo es una copia del proceso padre en el instante en que este solicita el servicio `fork()`. Esto significa que los datos y la pila del proceso hijo son los que tiene el padre en ese instante de ejecución.

Piense y responda: ¿El proceso hijo empieza la ejecución del código en su punto de inicio, o en la sentencia que está después del `fork()`?

Observe que el hijo no es totalmente idéntico al padre, algunos de los valores del BCP han de ser distintos.

Piense y responda: ¿Cuáles deberían ser las diferencias más importantes?

	V	F
El proceso hijo tiene su propio identificador de proceso, distinto al del padre		
El proceso hijo tiene una nueva descripción de la memoria.		
El tiempo de ejecución del proceso hijo se iguala a cero		
Todas las alarmas pendientes se desactivan en el proceso hijo		
El conjunto de señales pendientes se pone a vacío		
El valor que retorna el sistema operativo como resultado del `fork()` es distinto en el hijo y en el padre		

Tabla 2.2

Observe que las modificaciones que realice el proceso padre sobre sus registros e imagen de memoria después del `fork()` no afectan al hijo, y viceversa, las del hijo no afectan al padre. **Sin embargo**, el proceso hijo tiene su propia copia de los descriptores del proceso padre. Este hace que el hijo tenga acceso a los archivos abiertos por el proceso padre. El padre y el hijo comparten el puntero de posición de los archivos abiertos en el padre.

Piense y responda si esto podría afectar y de qué manera a lo que acaba de observar.

2.4.2 Práctica con Windows

2.4.2.1 tlist

En M. E. Russinovich y D. A. Solomon (2004) se nos propone un interesante ejercicio para visualizar el árbol de procesos en Windows. El identificador del proceso y de sus padres o creadores es un atributo particular de los procesos y que, no obstante, la mayoría de las herramientas no lo muestran. La herramienta `tlist.exe`, incluida en las Herramientas de Depuración de Windows (*Windows Debugging Tools*), nos muestra el árbol de procesos si utilizamos el modificador `/t`.

Recordemos que quedan instaladas en
`\Archivos de programa\Debugging Tools for Windows (x86)\`.

2.4.2.1.1 Ejemplo 1

```
c:\> cd "\Archivos de programa\Debugging Tools for Windows (x86)"
c:\Archivos de programa\Debugging Tools for Windows (x86)\> tlist /t
System Process (0)
System (4)
  smss.exe (524)
    csrss.exe (572)
    winlogon.exe (596)
      services.exe (640)
        svchost.exe (808)
        svchost.exe (856)
        svchost.exe (932)
        svchost.exe (1016)
        svchost.exe (1056)
        spoolsv.exe (1248)
        nod32krn.exe (1768) NOD32KrnSvcWindow
        nvsvc32.exe (1784) NVSVCPMMWindowClass
        SMAgent.exe (1844)
        alg.exe (380)
      lsass.exe (652)
explorer.exe (1176) Program Manager
  nod32kui.exe (1444) IHW2
  SMax4PNP.exe (1452) SMax4PNP
  SMax4.exe (1460) SoundMax4
  rundll32.exe (1484) MediaCenter
  TaskSwitchXP.exe (1496) TaskSwitchXP Pro 2.0
  ctfmon.exe (1504)
  cmd.exe (1620) Command Prompt
  cmd.exe (2860) Command Shell
  cmd.exe (2904) Command Shell - tlist /t
    tlist.exe (3800)
C:\Archivos de programa\Debugging Tools for Windows (x86)
```

Se dejan sangrías en el listado para mostrar la relación padre/hijo. Los procesos cuyos padres ya no existen están sobre la margen izquierda, como lo está `explorer.exe` en este ejemplo. Aunque exista su abuelo no hay forma de mostrar esa relación, porque Windows sólo mantiene la relación padre/hijo y no un enlace previo al creador del padre. En definitiva, vemos varias ramas pero no un árbol.

2.4.2.1.2 Ejemplo 2

```
C:\Archivos de programa\Debugging Tools for Windows (x86)>tlist /t
System Process (0)
System (4)
smss.exe (488)
csrss.exe (592)
wininit.exe (644)
  services.exe (688)
    svchost.exe (916)
    svchost.exe (976)
    svchost.exe (1012)
    svchost.exe (1104)
      audiodg.exe (1272)
```

```
    svchost.exe (1128)
      WUDFHost.exe (892)
      dwm.exe (2500) DWM Notification Window
    svchost.exe (1188)
      taskeng.exe (2476) MCI command handling window
      taskeng.exe (2392) TaskEng - Proceso del Motor de Programador de tareas
    SLsvc.exe (1300)
    svchost.exe (1328)
    svchost.exe (1472)
    spoolsv.exe (1632)
    svchost.exe (1660)
    ekrn.exe (1860)
    InCDsrv.exe (1968)
    mdm.exe (1988)
    PnkBstrA.exe (2032)
    svchost.exe (300)
    RichVideo.exe (364)
    svchost.exe (504)
    svchost.exe (540)
    SearchIndexer.exe (636)
    wmpnetwk.exe (3280)
  lsass.exe (704)
  lsm.exe (712)
csrss.exe (656)
winlogon.exe (788)
explorer.exe (2580) Program Manager
  RtHDVCpl.exe (2752) MMDEVAPI Device Window
  rundll32.exe (2860) MediaCenter
  GrooveMonitor.exe (2896)
  UnlockerAssistant.exe (2932) UnlockerAssistant
  PDVDServ.exe (2952) CL RC Engine3 Dummy Winidow
  InCD.exe (2988) InCD Log
  egui.exe (3012) ESET Smart Security
  wmpnscfg.exe (3104) Servicio de uso compartido de red del
      Reproductor de Windows Media
  Rainlendar.exe (3120) Rainlendar control window
  CursorXP.exe (4012)
  cmd.exe (3132) Símbolo del sistema - tlist  /t
    conime.exe (3424)
    tlist.exe (348)
rundll32.exe (3028)
C:\Archivos de programa\Debugging Tools for Windows (x86)
```

2.4.2.2 Información de los procesos con el Administrador de Tareas

El Administrador de Tareas de Windows nos provee una lista rápida de los procesos que están ejecutando en el sistema. Se puede arrancar el Administrador de Tareas de tres maneras:

1. Presionando simultáneamente Ctrl+Mayúsculas+Esc
2. Haciendo clic con el botón derecho del *mouse* sobre la barra de tareas y seleccionando Administrador de Tareas.
3. Presionando simultánemente Ctrl+Alt+Supr y haciendo clic en el botón del Administrador de Tareas.

Una vez que ha comenzado el Administrador de Tareas, haga clic en la pestaña `Procesos` para ver la lista de procesos que se están ejecutando. Note que los procesos están identificados por el nombre de la imagen de la cual son una instancia. A diferencia de algunos objetos en Windows, a los procesos no se les puede dar nombres globales. Para ver detalles adicionales, elija `Seleccionar Columnas` desde el menú `Ver` y seleccione otras columnas para agregar.

Sistemas operativos

MARTÍN SILVA



Alfaomega

Memoria

3

La finalidad de una computadora es la de procesar información mediante la ejecución de los programas apropiados. Estos programas, junto con los datos a los que acceden, deben co-residir durante su ejecución en la memoria principal (RAM) con el sistema operativo. La administración de memoria es responsable de la asignación de memoria a los procesos y de proteger esa memoria asignada a cada proceso de accesos no deseados de otros procesos, también es responsable de proteger la memoria asignada al sistema operativo de accesos no autorizados.

El precio de la memoria ha descendido drásticamente con el pasar de los años; la cantidad de memoria de las primeras computadoras era de unos pocos *kilobytes* y la de las actuales de varios *gigabytes*. No obstante, el software también ha crecido enormemente, dándonos siempre la impresión de que nuestra memoria es escasa. Y, por otra parte, ciertas técnicas que veremos le permiten al sistema operativo hacerle creer a las aplicaciones que tienen más memoria principal que la que realmente poseen.

La administración de memoria no es sólo tarea del software. El sistema operativo requiere soporte de hardware para implementar cualquiera de los esquemas de administración de memoria no triviales. Por esto, algunos aspectos del diseño del sistema operativo también son aspectos de diseño del hardware. El hardware de administración de memoria típicamente está protegido del acceso por parte de los usuarios; el sistema operativo solamente es responsable de su control.

Esta fuera del alcance de este capítulo extenderse sobre el hardware, pues el objetivo es manejarnos en un nivel de abstracción con respecto a la capa física. Se detallará sólo cuando sea necesario.

3.1 Funciones y operaciones del administrador de memoria

Como administrador de la memoria, un sistema operativo multitarea debería cumplir con las siguientes funciones, según Lister y R. D. Eager (1993):

Reubicación. En sistemas con memoria virtual, los programas cargados en memoria deben ser capaces de residir en diferentes partes de la misma en distintos momentos de su ejecución. Una vez que se ha descargado (*swap*) un programa al disco, no es deseable que deba volver a cargarse en la misma región de la memoria principal, sino que debería poder reubicarse en otra, y el administrador de memoria del sistema operativo debería ser capaz de manejar todas las referencias a memoria que hace el programa, de manera que apunten a la dirección correcta.

Protección. Los procesos no deben ser capaces de acceder a direcciones de memoria de otros procesos, a menos que esté explícitamente permitido, y con ello nos referimos tanto a interferencias accidentales como intencionales. En última instancia, es el procesador (hardware), y no el sistema operativo (software), el que debe satisfacer las exigencias de protección de memoria. El sistema operativo no puede anticiparse a todas las referencias a la memoria que hará un programa.

Compartimiento. No obstante lo anterior, muchas veces es deseable que los procesos puedan compartir información y, por lo tanto, acceder a un área de memoria ajena. La memoria compartida es una técnica muy utilizada en la comunicación entre procesos.

Organización lógica La memoria de un equipo informático está organizada de manera lineal o unidimensional, como una secuencia de *bytes*; la de un disco, de manera similar. Esta organización del hardware no es apta para los programas que la utilizan. A menudo, los programas están organizados en módulos o áreas, que pueden ser modificables o no, ya sea porque contienen datos o áreas de sólo lectura o solamente ejecutables. Una forma de organización lógica es la **segmentación** (ver 3.3.4).

Organización física. De forma básica podemos decir que la memoria de un computador se divide en dos niveles. Por una parte, la memoria principal, cuyo acceso es rápido, en el orden de nanosegundos (10^{-9}) y accesible en forma directa por la CPU y, por otra, la memoria secundaria, de acceso lento (en el orden de milisegundos (10^{-3}). Es requisito del código gestor de memoria del sistema operativo que se encargue de la transferencia de datos entre una y otra.

Podríamos agregar otro requisito que se ha de satisfacer, que es el de la **compactación** de la memoria, para lograr tener pocos grandes espacios libres de memoria contigua y no muchos pequeños espacios libres y dispersos.

Según sugiere Stallings (2001) en su libro, para lograr un buen resultado no deberíamos depender de un tipo de componente o tecnología. Existe una jerarquía de memoria que nos permite hacer un categoría de velocidad de acceso, capacidad y costo.

A medida que bajamos de nivel de jerarquía, vamos obteniendo menos costo por bit, más capacidad y más tiempo de acceso.

Un punto sumamente importante es la frecuencia de acceso a la memoria por parte del procesador. Para entenderlo, consideremos que a la información mantenida en el primer nivel (registros) el acceso es inmediato. Si está en el segundo nivel (caché), primero debe pasar al primer nivel y luego acceder al procesador.

Jerarquía tradicional	Jerarquía moderna
Registros	Registros
Caché	Caché
Memoria principal	Memoria principal
Disco magnético	Caché de disco
Cinta magnética	Disco magnético
	Cinta magnética/disco óptico

Tabla 3.1 – Jerarquía de memorias.

Hay un principio, observado por Peter J. Denning en 1967 (ver Denning 2005), llamado «cercanía de referencias», que observa que cuando se ejecuta un programa, las referencias a memoria que hace el procesador tienden a estar agrupadas, por ejemplo cuando se accede sucesivamente a todos los elementos de un arreglo. Lo mismo ocurre con las referencias a instrucciones cercanas y al llamar a subrutinas. Es el concepto de localidad que veremos más profundamente en la subsección 3.4.2 y en hiperpaginación o *thrashing* (3.4.7.1).

Lo ideal seria tener acceso rápido al conjunto de direcciones agrupadas (o localidad) pues son las de uso inminente, y que cada nueva localidad desplace a la anterior y ocupe este lugar de preferencia.

> Una **caché** (del inglés *cache*, antememoria), también llamada *look-aside* por Bloom, Cohen y Porter (1962) o «memoria esclava» por Wilkes (1965), puede acelerar los accesos a todas las formas de dispositivos de almacenamiento, incluyendo a las unidades de cinta, a las de disco, memoria principal, servidores en una red (por ejemplo, los servidores web son un tipo de dispositivo de almacenamiento) o incluso otras cachés. Funcionan sobre el principio de localidad de referencia, la tendencia de las aplicaciones a referenciar una pequeña cantidad predecible de datos dentro de un ventana de tiempo dada (Belady 1966; Denning 1970).

Es de fundamental importancia que quede muy bien entendido los «cuellos de botella» con los que se encuentra un proceso. Vea en la tabla 3.2 los tiempos típicos de los distintos dispositivos, pero compárelos con respecto al primero:

Analizando la evolución de los diferentes esquemas de administración de memoria, veremos que es semejante a la evolución de la administración del espacio en disco para la asignación de bloques a los archivos.

3.2 Modelo de memoria de un proceso

El sistema operativo gestiona el mapa de memoria de un proceso durante la vida del mismo.

Tipo de dispositivo	Tiempo típico de servicio	Relativo al segundo
Buffer del procesador	10 ns	1 s
Memoria de acceso aleatorio	60 ns	6 s
Llamada a procedimiento	1 μs	2 m
Memoria expandida	25 μs	1 h
RPC local	100 μs	4 h
Disco estado sólido	1 ms	1 d
Disco «cacheado»	10 ms	12 d
Disco magnético	25 ms	4 sem
Disco via LAN alta velocidad	27 ms	1 mes
Disco via LAN serv.	35-50 ms	6-8 semanas
Disco via WAN serv.	1-2 s	3-6 años
Disco/cinta montable	3-15s	10-15 años

Tabla 3.2 – Tiempos de acceso a distintas memorias.

La correspondencia, proyección o trazado de un mapa en la memoria (*memory mapping*) es una técnica que consiste en hacer que una parte del espacio de direcciones parezca contener un objeto, tal como un archivo o dispositivo, de manera que los accesos a esa parte de la memoria actúen sobre el objeto.

Dado que el mapa inicial de un proceso está muy vinculado con el archivo que contiene el programa ejecutable asociado al mismo, comenzaremos estudiando cómo se genera un archivo ejecutable y cuál es la estructura típica del mismo, cómo evoluciona el mapa a partir de ese estado inicial y qué tipos de regiones existen típicamente en el mismo identificando cuáles son sus características básicas.

3.2.1 Fases en la generación de un ejecutable

En general, una aplicación estará compuesta por un conjunto de módulos de código fuente que deberán ser procesados para obtener el ejecutable de la aplicación. Como se puede observar en la figura 3.1, este procesamiento consta de dos fases: compilación, que genera el código máquina correspondiente a cada módulo fuente de la aplicación, y enlace o montaje (*link*), que genera un ejecutable agrupando todos los archivos objeto y resolviendo las referencias entre módulos (ver Carretero Pérez 2001).

Además de referencias entre módulos, pueden existir referencias a símbolos definidos en otros archivos objeto previamente compilados, agrupados normalmente en bibliotecas (*libraries*).

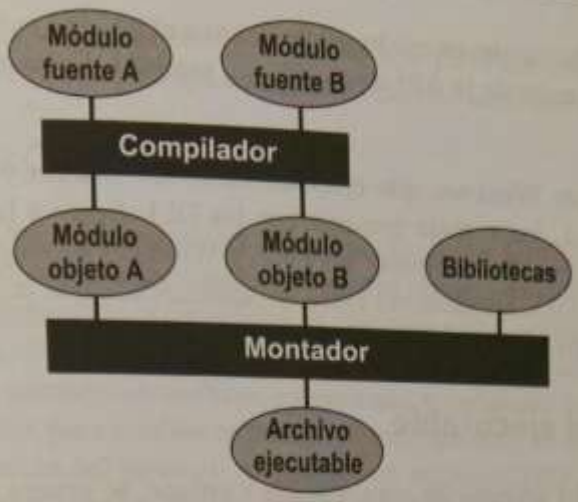


Fig. 3.1 – Fases en la generación de un ejecutable.

> Una biblioteca es una colección de objetos tales como subrutinas (o clases en la Programación Orientada a Objetos), normalmente relacionados entre sí, y se usa para desarrollar software.

La manera de generar el ejecutable comentado hasta ahora consiste en compilar los módulos fuente de la aplicación y enlazar los módulos objeto resultantes junto con los extraídos de las bibliotecas correspondientes. De esta forma, se podría decir que el ejecutable es autocontenido: incluye todo el código que necesita el programa para poder ejecutarse y es lo que se conoce como enlazarlo de manera **estática**. Existe, sin embargo, una alternativa que presenta numerosas ventajas: las bibliotecas dinámicas.

Con este nuevo mecanismo, el proceso de enlace de una biblioteca de este tipo se difiere y, en lugar de realizarlo en la fase de enlace, se realiza durante la ejecución del programa. Cuando en la fase de enlace el *linker* procesa una biblioteca dinámica, no incluye en el ejecutable código extraído de la misma, sino que simplemente anota en el ejecutable el nombre de la biblioteca para que esta sea cargada y enlazada durante la ejecución. El uso de bibliotecas dinámicas presenta múltiples ventajas: se reduce el tamaño de los ejecutables, se favorece el compartimiento de información y se facilita la actualización de la biblioteca. Pero note que esta acción requiere de operaciones de entrada y salida, que demoran la ejecución del proceso (ver tabla 3.2).

La forma habitual de usar las bibliotecas dinámicas consiste en especificar al momento de enlace (*link time*) qué bibliotecas se deben usar, pero la carga y el enlace se pospone hasta el momento de ejecución (*run time*). Sin embargo, también es posible especificar al momento de ejecución qué biblioteca dinámica se necesita y solicitar explícitamente su enlace y carga (a esta técnica se la suele llamar ***carga explícita de bibliotecas dinámicas***).

> **DLL** (***Dynamic-Link Library***). Es un conjunto de subrutinas invocables por un proceso y enlazadas juntas en un archivo binario, que puede ser cargado dinámicamente por aplicaciones que usen esas subrutinas. Por ejemplo, `msvcrt.dll` es la biblioteca del lenguaje C que se invoca durante la ejecución de un proceso.

cuyo programa fue escrito en ese lenguaje, y `kernel32.dll` es una de las bibliotecas del susbsistema de la API de Windows, segúnM. E. Russinovich y D. A. Solomon 2004.

Los componentes de Windows que ejecutan en modo usuario y las aplicaciones usan extensivamente las DLL. La ventaja que proveen las DLL frente a las bibliotecas estáticas es que las aplicaciones pueden compartir las DLL y Windows se asegura que haya sólo una copia del código de la DLL en memoria entre las aplicaciones que la referencian. (relacione con código reentrante en 3.3.9)

3.2.2 Formato del ejecutable

Como parte final del proceso de compilación y enlace, se genera un archivo ejecutable que contiene el código máquina del programa. Como se puede observar en la figura 3.2, un ejecutable está estructurado como una cabecera y un conjunto de secciones.

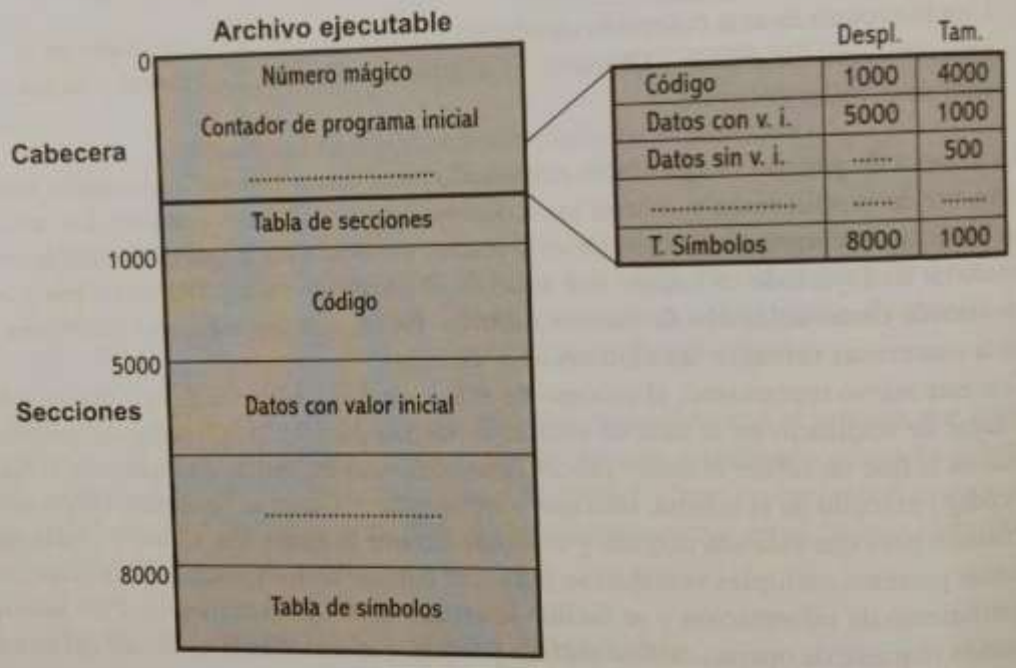


Fig. 3.2 – Formato simplificado de un ejecutable.

La cabecera contiene información de control que permite interpretar el contenido del ejecutable. En cuanto a las secciones, cada ejecutable tiene un conjunto de secciones; típicamente, aparecen al menos las tres siguientes:

- Código (texto): contiene el código del programa.
- Datos con valor inicial: almacena el valor inicial de todas las variables globales a las que se les ha asignado un valor inicial en el programa.
- Datos sin valor inicial: Se corresponde con todas las variables globales a las que no se les ha dado un valor inicial. Como se muestra en la figura, este apartado aparece

descripto en la tabla de secciones de la cabecera, pero, sin embargo, no se almacena normalmente en el ejecutable, ya que el contenido de la misma es irrelevante.

3.2.2.1 El formato COFF

COFF (*Common Object File Format*) es la especificación del formato para archivos objeto, ejecutables y bibliotecas compartidas que se usó en sistemas UNIX a partir de System V Release 3 (SVR3), agregando la capacidad de contener varias secciones, característica que faltaba en el formato `a.out` original de UNIX. Se le puede agregar información para depurar el programa, es decir, nombres simbólicos a funciones y variables, información del número de línea –que son útiles para establecer puntos de detención de la ejecución (*breakpoint*)– o ver la traza de ejecución del proceso. Al generar un archivo COFF no se establece en qué parte de la memoria será cargado. La dirección virtual donde el primer *byte* del archivo se cargará se llama dirección base de la imágen. Este formato es la base de los formatos ELF y PE utilizados por Linux y Windows, respectivamente.

3.2.2.2 El formato ELF

ELF (*Executable and Linking Format*) es el formato de archivos objeto, ejecutables, bibliotecas compartidas y volcados de memoria (*core dump*). Es muy flexible y extensible y no está limitado a un procesador o arquitectura en particular. Ha reemplazado a los formatos `a.out` y COFF y se utiliza en todos los sistemas operativos tipo UNIX modernos, en particular en Linux. También es utilizados en las consolas de juego Playstation Portable, Playstation 2, Playstation 3 y Wii.

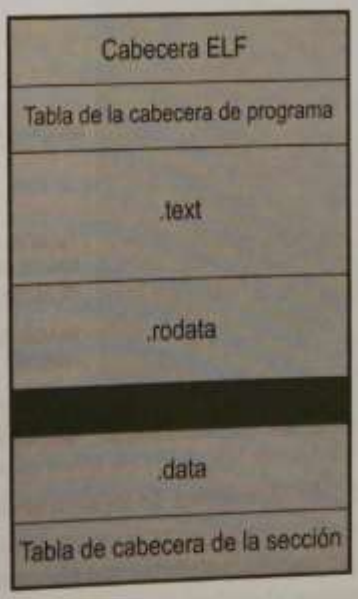


Fig. 3.3 – Estructura de un archivo ELF.

Su estructura consiste de una cabecera seguida por datos del archivo, que a su vez incluyen:

- Una tabla de cabecera del programa que describe cero o más segmentos.
- Una tabla de cabecera de sección que describe cero o más secciones.
- Datos referenciados por las entradas en la tabla de cabecera del programa o en la tabla de cabecera de la sección.

Los segmentos contienen información necesaria para la ejecución del archivo y las secciones contienen datos importantes para el enlace y la reubicación. Normalmente, un segmento incluye una o más secciones (ver figura 3.3).

3.2.2.3 El formato PE

Microsoft introdujo el formato de archivo PE (*Portable Executable*) como parte de la especificación original Win32. El archivo PE deriva del COFF, que utilizaba el sistema operativo VAX/VMS, lo cual se comprende porque los arquitectos originales del equipo de desarrollo del Windows NT provenían de Digital Equipment Corporation. El término fue elegido para indicar un común denominador con todas las versiones de los sistemas operativos de Microsoft y para todas las CPU soportadas. Se lo utiliza en los ejecutables, archivos objeto y bibliotecas de enlace dinámico (DLL).

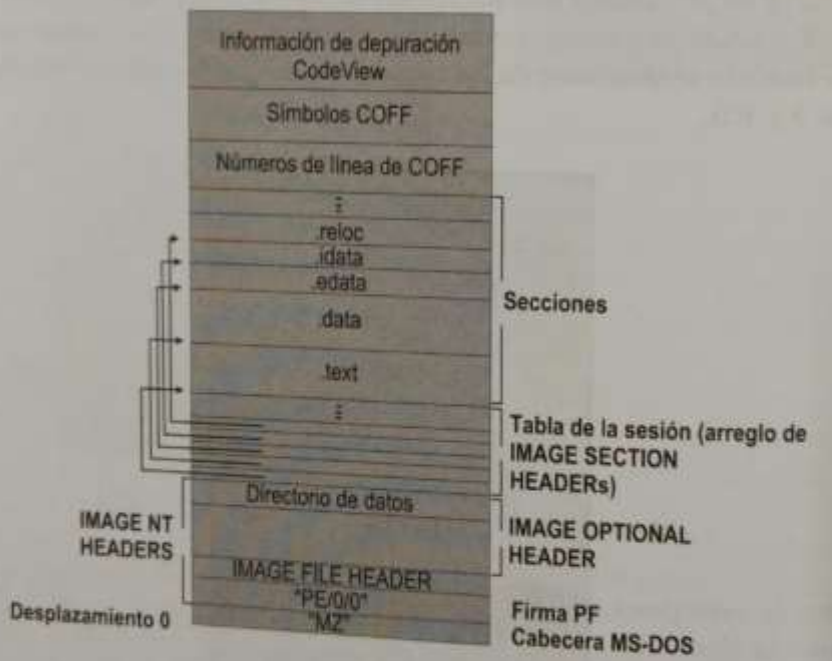


Fig. 3.4 – Estructura de un archivo PE.

Es una estructura de datos que encapsula la información necesaria para que el cargador del sistema operativo Windows gestione el código ejecutable envuelto dentro de la misma. Cada región puede requerir distinta protección en memoria, de manera que el comienzo de cada

sección deberá estar alineada con un límite de una página. Por ejemplo, la sección `.text` que contiene el código del programa está marcado como de ejecución y de sólo lectura, mientras que la sección `.data` que tiene variables globales está marcada como de no ejecución y de lectura y escritura. (Ver Figura 3.4)

A diferencia de ELF, que usa código independiente de la posición (*position-independent code*), PE se compila a una dirección base preferida (*preferred base address*) y si no puede ser cargado en esa dirección preferida, el sistema operativo tiene que recalcular la base, lo que implica recalcular todas las direcciones y modificar el código para usar los nuevos valores, afirmaPietrek 1994.

3.2.3 Mapa de memoria de un proceso

El mapa de memoria de un proceso no es algo homogéneo, sino que está formado por distintas regiones o segmentos. Cuando se activa la ejecución de un programa, se crean varias regiones dentro del mapa a partir de la información del ejecutable. Las regiones iniciales del proceso se van a corresponder básicamente con las distintas secciones del ejecutable.

Cada región es una zona contigua que está caracterizada por la dirección dentro del mapa del proceso donde comienza y por su tamaño. Además, tendrá asociada una serie de propiedades y características específicas, tales como las siguientes:

- Soporte de la región, donde está almacenado el contenido inicial de la región. Se presentan normalmente dos posibilidades:
  - Soporte en archivo: está almacenado en un archivo o en parte del mismo.
  - Sin soporte: no tiene un contenido inicial.
- Tipo de uso compartido:
  - Privado: el contenido de la región sólo es accesible al proceso que la contiene. Las modificaciones sobre la región no se reflejan permanentemente.
  - Compartido: el contenido de la región puede ser compartido por varios procesos. Las modificaciones en el contenido de la región se reflejan permanentemente.
- Protección: tipo de acceso a la región permitido. Típicamente, se proporcionan tres tipos:
  - Lectura: se permiten accesos de lectura de operandos de instrucciones.
  - Ejecución: se permiten accesos de lectura de instrucciones.
  - Escritura: se permiten accesos de escritura.
- Tamaño fijo o variable: en el caso de regiones de tamaño variable, se suele distinguir si la región crece hacia direcciones de memoria menores o mayores.

Las regiones que presenta el mapa de memoria inicial del proceso se corresponden básicamente con las secciones del ejecutable más la pila inicial del proceso, a saber:

- Código (texto): se trata de una región compartida de lectura/ejecución. Es de tamaño fijo. El soporte de esta región está en el apartado correspondiente del ejecutable. Se la suele llamar `.text`

- Datos con valor inicial: se trata de una región privada, ya que cada proceso que ejecuta un determinado programa necesita una copia propia de las variables del mismo. Es de lectura/escritura y de tamaño fijo. El soporte de esta región está en el apartado correspondiente del ejecutable. Por ejemplo, la declaración «`int i=7`»; en lenguaje C estaría en esta región. Se la suele llamar `.data`
- Datos sin valor inicial: se trata de una región privada, de lectura/escritura y de tamaño fijo (el indicado en la cabecera del ejecutable). Como se comentó previamente, esta región no tiene soporte en el ejecutable, ya que su contenido inicial es irrelevante. Por ejemplo, la declaración «`int i[10]`»; en lenguaje C estaría en esta región. Se le suele llamar `.bss`
- Pila: esta región es privada y de lectura/escritura. Servirá de soporte para almacenar los registros de activación de las llamadas a funciones (las variables locales, parámetros, direcciones de retorno, etc.) Se trata, por lo tanto, de una región de tamaño variable que crecerá cuando se produzcan llamadas a funciones y decrecerá cuando se retorne de las mismas. Típicamente, esta región crece hacia las direcciones más bajas del mapa de memoria. En el mapa inicial existe ya esta región que contiene los argumentos especificados en la invocación del programa.

Los sistemas operativos modernos ofrecen un modelo de memoria dinámico, en el que el mapa de un proceso está formado por un número variable de regiones que pueden añadirse o eliminarse durante la ejecución del mismo. Además de las regiones iniciales ya analizadas, durante la ejecución del proceso pueden crearse nuevas regiones relacionadas con otros aspectos, tales como los siguientes:

- *Heap* (montículo): la mayoría de los lenguajes de alto nivel ofrecen la posibilidad de reservar espacio durante su ejecución. En el caso del lenguaje C, se usa la función «malloc» para ello. Esta región sirve de soporte para la memoria dinámica que reserva un programa durante su ejecución. Comienza justo después de la región de datos sin valor inicial (de hecho, en algunos sistemas se considera parte de la misma) y crece en sentido contrario a la pila (hacia direcciones crecientes). Se trata de una región privada de lectura/escritura, sin soporte (se rellena inicialmente con ceros), que crece según el programa vaya reservando memoria dinámica y decrece según la vaya liberando.
- Archivos proyectados: cuando se proyecta un archivo, se crea una región asociada al mismo.
- Memoria compartida: cuando se crea una zona de memoria compartida y se proyecta, se crea una región asociada a la misma. Se trata, evidentemente, de una región de carácter compartido, cuya protección la especifica el programa a la hora de proyectarla.
- Pilas de *threads*: cada *thread* necesita una pila propia que, normalmente, se corresponde con una nueva región en el mapa. Este tipo de región tiene las mismas características que la región correspondiente a la pila del proceso. Recuerde que una pila es una estructura de datos que se accede de modo LIFO.

En la figura 3.5 se muestra un hipotético mapa de memoria que contiene algunos de los tipos de regiones comentados en este apartado.

Como puede apreciarse en la figura anterior, la carga de una biblioteca dinámica implicará la creación de un conjunto de regiones asociadas a la misma, que contendrán las distintas

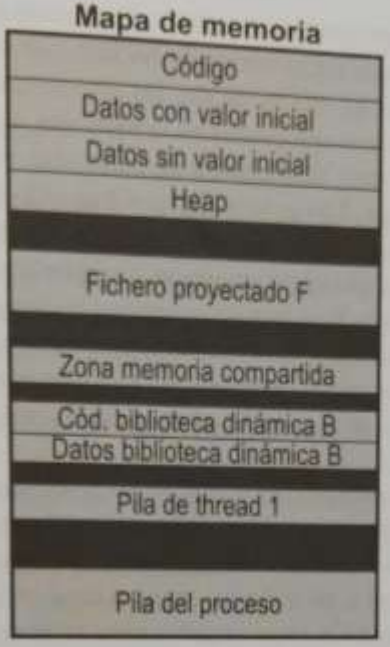


Fig. 3.5 – Mapa de memoria de un proceso hipotético.

secciones de la biblioteca (código y datos globales). Para visualizar estos conceptos pase a la práctica de la subsección 3.5.1.4 .

3.3 Diferentes esquemas de administración

La memoria principal debe dar cabida tanto al sistema operativo como a los diversos procesos de usuario. En los esquemas de **asignación contigua**, la memoria se divide en dos particiones, una para el sistema operativo residente y otra para los procesos de usuario. Es posible colocar el sistema operativo en memoria baja o alta. El factor principal que afecta esta decisión es la ubicación del vector de interrupciones. Puesto que el vector de interrupciones suele estar en la memoria baja, es más común colocar el sistema operativo en esa misma área.

3.3.1 Mono programación

3.3.1.1 Partición absoluta única

El esquema de administración de memoria más simple no requiere soporte de hardware. El espacio de memoria está dividido por convención en dos particiones: una partición está asignada al sistema operativo y la otra a un proceso en ejecución. Los así llamados programas de «buen comportamiento» se limitan a acceder las ubicaciones de memoria en la partición del proceso. Sin soporte de hardware, sin embargo, no hay nada que prevenga que los programas corrompan el sistema operativo. Los primeros sistemas DOS usaban una variante de este

esquema, que permitían que los programas de usuarios con «mal comportamiento» pudieran quebrar el sistema completo (lo que frecuentemente hacían).

Cuando un programa se carga en una partición, las direcciones en el código cargado deben corresponder con las direcciones apropiadas en la partición. Las direcciones en el código pueden estar **limitadas** a direcciones de memoria en particular, ya sea al momento de compilación o al momento de la carga. Los programas que han sido limitados a ubicaciones de memoria al momento de compilación se dice que contienen **código absoluto**. En cambio, si esta ligazón (*binding*) ocurre cuando el programa se carga en memoria, se dice que el programa contiene **código reubicable**. Para que un programa contenga código reubicable, debe existir algún mecanismo para identificar los *bytes* en el programa que se refieren a direcciones de memoria. A pesar de que esto agrega complejidad a los procesos de compilación y carga, tiene la ventaja obvia de liberar al programa de ser cargado en cualquier ubicación disponible de la memoria.

Se le puede incorporar protección a un esquema de partición absoluta agregándole al hardware un **registro base**. El sistema operativo carga el registro base con la dirección más baja accesible por un programa de usuario. El hardware luego compara cada dirección generada por el programa de usuario con el contenido del registro base (ver fig. 3.6). Las direcciones menores que la dirección almacenada en el registro base provocan una trampa de fallo de memoria (*memory-fault trap*) en el sistema operativo.

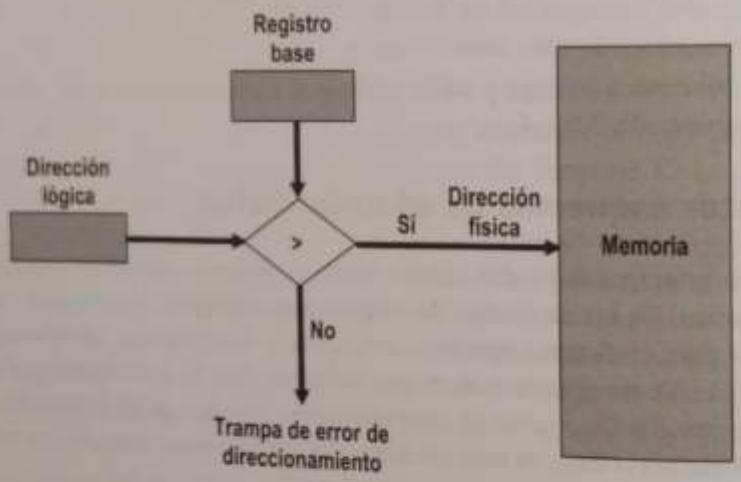


Fig. 3.6 – Direccionamiento con registro base.

3.3.1.2 Partición reubicable única

Mucho más útiles que los sistemas de registro base son los sistemas que contienen un **registro de reubicación** (*relocation register*). Como con el registro base, el registro de reubicación es cargado por el sistema operativo con la dirección de comienzo del proceso. Pero en lugar de comparar el contenido del registro de reubicación con la dirección generada por el proceso, sus contenidos se suman. El programa ahora opera en un **espacio de direcciones lógico**. Es compilado como si fuera a ser asignado a la memoria que comienza en la ubica-

ción 0. El hardware de administración de memoria convierte las direcciones lógicas en las **direcciones físicas** verdaderas.

3.3.1.3 Superposiciones

En todos los esquemas previos de administración de la memoria, la cantidad de memoria disponible a un proceso está limitado por la cantidad de memoria física. El problema de tener programas con requerimientos de memoria superiores a la memoria del computador data desde el comienzo de las computadoras. En aquel momento se optó por dividir el programa en partes a las que se llamó superposiciones (*overlays*), y está muy bien presentado en el trabajo de Pankhurst (1968).

Una superposición mantiene en memoria sólo aquellas instrucciones y datos necesarios en un momento dado. Se definen previamente a la ejecución varias superposiciones, que estarán en diferentes momentos en memoria, reemplazando a la anterior, cuando el proceso la requiera. Comienza a ejecutarse la superposición 0, al terminar se descarga y se carga la 1, y así sucesivamente cargando y descargando.

El sistema operativo se encargaba de la carga y descarga pero el trabajo del programador era tedioso, pues debía descomponer su programa en módulos pequeños para crear las superposiciones.

En Silberschatz y Galvin (1997) está el ejemplo del cuadro 3.3 bastante claro. Supongamos un ensamblador de dos pasadas. Durante la pasada 1, el ensamblador construye una tabla de símbolos; después, durante la pasada 2, genera código en lenguaje de máquina. Quizá podríamos dividir semejante ensamblador en código de la pasada 1, código de la pasada 2, tabla de símbolos y rutinas de soporte comunes utilizadas tanto por la pasada 1 como por la 2.

Código del paso 1	70k
Código del paso 2	80k
Tabla de símbolos	20k
Rutinas comunes a ambos pasos	30k
En total	200k

Tabla 3.3 – Superposiciones (*overlays*).

Para cargar todo a la vez, necesitaríamos 200K de memoria. Si sólo contamos con 150K, no podremos ejecutar nuestro proceso. Pero la pasada 1 y la 2 no tienen que estar en la memoria al mismo tiempo. Definimos dos superposiciones: la A contiene la tabla de símbolos, las rutinas comunes y la pasada 1, y la superposición B contiene la tabla de símbolos, las rutinas comunes y la pasada 2.

No debemos olvidar cargar un manejador (*driver*) de superposiciones antes de cargar la primera. Una vez ejecutada esta superposición, será el manejador el encargado de leer y cargar la segunda superposición a memoria, sobrescribiendo la primera.

Hay que considerar que esta técnica aporta entrada/salida adicional en la lectura de las superposiciones pues estas residen en disco como imágenes absolutas de memoria. Son implementadas por el programador sin soporte especial por parte del sistema operativo.

Esta técnica de programación suele utilizarse actualmente en los sistemas integrados (*embedded systems*) por sus limitaciones de memoria física, que es una memoria interna dentro de un «sistema en chip» o SoC (*system on chip*), y porque no tienen memoria virtual.

> Un sistema integrado (*embedded system*) es una computadora diseñada para realizar unas pocas funciones dedicadas, a menudo con limitaciones de tiempo real, en contraste con una computadora personal, que está diseñada para ser flexible en sus funciones.

Por ejemplo, podemos considerar que un dispositivo ADSL (*Asymmetric Digital Subscriber Line*) enrutador inalámbrico es un sistema integrado, como también puede serlo una consola de videojuegos. Esto puede deducirse si observa que normalmente no hay actividad de entrada/salida en un videojuego mientras se permanece en un determinado nivel y, al superarlo, se descarga de memoria para cargar el siguiente.

La mayor desventaja de las superposiciones es el esfuerzo que le demanda al programador. Identificar segmentos que se han de superponer no es tarea fácil ni confiable. Una solución muy superior es la utilización de un esquema de administración de la memoria que libere al usuario de las limitaciones de la memoria física. A tales sistemas se los denomina **memoria virtual**.

3.3.2 Multiprogramación

En un ambiente de multiprogramación hay varios procesos compartiendo la memoria, por lo tanto, la protección se debe intensificar, protegiendo los espacios de los procesos entre sí y con respecto a la memoria del sistema.

Los esquemas de partición única limitan a una computadora a ejecutar un programa a la vez. Tales sistemas de mono programación en la actualidad son raros de encontrar (excepto las consolas de juegos), pero eran comunes en las primeras computadoras. Para cargar múltiples procesos, el sistema operativo debe dividir la memoria en múltiples particiones para esos procesos.

En un esquema de partición múltiple, se crean varias particiones para permitir que múltiples procesos de usuario queden residentes en la memoria en forma simultánea. Un segundo registro de hardware, para usarlo en conjunto con el registro de reubicación, marca el final de una partición. Este registro puede contener ya sea el tamaño de la partición o la última dirección en la partición. Típicamente, el registro contiene el tamaño de la partición y se lo llama **registro de tamaño** o **registro límite** (ver figura 3.7).

3.3.2.1 Múltiples particiones fijas

Si todas las particiones son del mismo tamaño, el sistema operativo sólo necesita llevar la cuenta de cuáles particiones están asignadas a cada proceso. La tabla de particiones de memoria almacena o bien la dirección de comienzo para cada proceso o el número de la

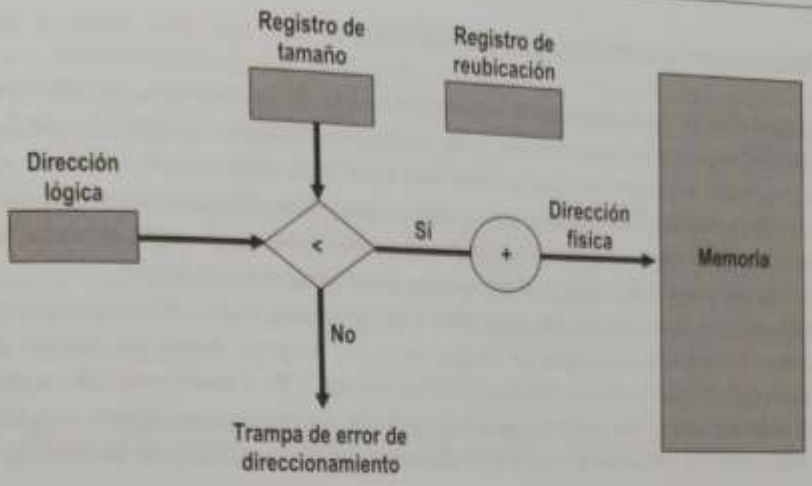


Fig. 3.7 – Registros de tamaño y reubicación.

partición asignada a cada proceso. Si está almacenado el número de partición y el tamaño de la partición es potencia de 2, la dirección de comienzo puede ser generada al concatenar el número apropiado de ceros al número de partición. El registro límite se establece al momento del arranque y contiene el tamaño de la partición. Cada vez que a un proceso se le asigna control de la CPU, el sistema operativo debe restablecer el registro de reubicación. El espacio al final de una partición que no es usado por un proceso, se desperdicia.



Al espacio desperdiciado dentro de una partición se le llama **fragmentación interna**.

Un método para reducir la fragmentación interna es usar particiones de diferentes tamaños[1]. Si se hace esto, el sistema operativo también debe restablecer el registro límite cada vez que se le asigna un proceso a la CPU. El tamaño de la partición puede estar almacenado en la tabla de particiones de la memoria, o bien puede deducirse a partir del número de partición.

El tener particiones de diferente tamaño complica la administración de procesos. Cada vez que a un proceso diferente se le da control de la CPU, el sistema operativo debe restablecer el registro de tamaño además del registro de reubicación. El sistema operativo también debe tomar decisiones acerca de en cuál partición debería asignar a un proceso. Obviamente, un proceso no puede ser asignado a una partición menor que el tamaño del proceso. Pero, ¿debería un proceso más pequeño ser asignado a una partición más grande si cabe en una partición más pequeña, aun si la partición más pequeña actualmente está ocupada por otro proceso?

1. El grado de multiprogramación queda definido por la cantidad de particiones.

3.3.2.2 Múltiples particiones variables

En lugar de dividir la memoria en un conjunto fijo de particiones, un sistema operativo puede elegir ubicar procesos en cualquier ubicación de memoria que esté sin usar. La cantidad de espacio asignado a un proceso es la cantidad exacta de espacio que requiere, eliminando la fragmentación interna. Sin embargo, el sistema operativo debe manejar más datos. Se debe almacenar la ubicación exacta de comienzo y finalización de cada proceso y se deben mantener los datos acerca de cuáles ubicaciones de memoria están libres.

A medida que los procesos se van creando y terminando, el uso de la memoria evoluciona hacia secciones alternadas de espacio asignado y sin asignar, como un tablero de ajedrez (*checkerboarding*), según lo llama Harris (2002) (ver fig. 3.8). Como resultado, a pesar de que puede haber más memoria sin asignar que el tamaño de un proceso en espera, esta memoria no puede ser usada por un proceso porque está dispersa entre un número de **huecos** de memoria.

> A este espacio desperdiciado no asignado a ninguna partición se le llama **fragmentación externa**.

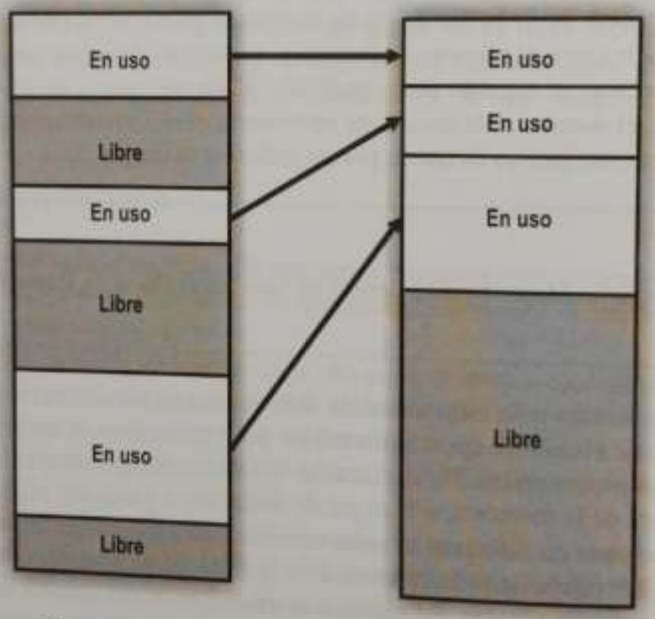


Fig. 3.8 – Fragmentación externa y compactación.

Se puede usar un mapa de bits para mantener un control de qué memoria ha sido asignada. La memoria está dividida en unidades de asignación y cada bit en el mapa indica si está asignada o no la unidad correspondiente. El incrementar el tamaño de la unidad de asignación hace decrecer el tamaño del mapa de bits, pero incrementa la cantidad de memoria desperdiciada cuando el tamaño del proceso no es un múltiplo del tamaño de la unidad de asignación.

También se puede usar una lista enlazada para mantener un control de la memoria libre. Cada agujero contendrá una entrada indicando el tamaño del agujero y un puntero al próximo agujero en la lista. El sistema operativo sólo necesita un puntero al primer agujero en la lista. Esta lista puede mantenerse por orden de memoria, orden de tamaño o en ningún orden en particular.

Se puede emplear la **compactación** para hacer un uso más eficiente de la memoria. Al mover procesos en memoria, los huecos en la misma se pueden recolectar en una sección única de espacio sin asignar. La cuestión que se nos plantea es si este movimiento de procesos se justifica o no, porque el intentar mejorar el espacio disponible de memoria implica una sobrecarga extra de trabajo.

3.3.2.2.1 Algoritmos de selección de la partición:

En situaciones en las que múltiples agujeros de memoria son suficientemente grandes como para contener un proceso, el sistema operativo debe usar un algoritmo para seleccionar en qué agujero se cargará el proceso. Se han estudiado y propuesto una variedad de algoritmos:

- **Primer ajuste** (*first fit*) El sistema operativo revisa en todas las secciones de memoria libre. El proceso es asignado al primer hueco encontrado que sea mayor que el tamaño del proceso. A menos que el tamaño del proceso coincida con el tamaño del hueco, el agujero continúa existiendo, reducido por el tamaño del proceso.
- **Próximo ajuste** (*next fit*) En el «primer ajuste», ya que todas las búsquedas empiezan al comienzo de la memoria, siempre se ocupan más frecuentemente los huecos que están al comienzo de la memoria que los que están al final. El «próximo ajuste» intenta mejorar el desempeño al distribuir sus búsquedas más uniformemente sobre todo el espacio de memoria. Hace esto manteniendo el registro de qué hueco fue el último en ser asignado. La próxima búsqueda comienza en el último hueco asignado, no en el comienzo de la memoria.
- **Mejor ajuste** (*best fit*) El algoritmo del «mejor ajuste» revisa la lista completa de huecos para encontrar el hueco más pequeño, cuyo tamaño es mayor o igual que el tamaño del proceso.
- **Peor ajuste** (*worst fit*) En situaciones en las que el mejor ajuste encuentra una coincidencia casi perfecta, el hueco que queda es virtualmente inútil porque es demasiado pequeño. Para prevenir la creación de estos huecos inútiles, el algoritmo del «peor ajuste» trabaja de manera opuesta al del «mejor ajuste»; siempre elige el que deje el hueco remanente más grande.

First fit es la solución más rápida, lo que a los fines prácticos podría considerarse lo mejor y más sencillo. *First fit* y *Best fit* hacen un mejor aprovechamiento de la memoria que *Worst fit*.

El otro punto para tener en cuenta es cómo se organizan los procesos para ser asignados. El encolamiento se puede definir entre estos dos modelos:

3.3.2.2.2 Cola de procesos por partición:

Se encolan los procesos de acuerdo con la partición asignada, normalmente por tamaño. La ventaja es el aprovechamiento de los espacios de memoria. La desventaja es que puede haber encolamiento para una partición y que el resto de las particiones estén ociosas.

3.3.2.2.3 Cola única:

Los procesos a la espera de ser cargados se encolan en una cola única y un módulo administra el uso de todas las particiones. Si hay una partición justa para ese proceso, se asigna ese proceso a la partición, si no hay de su tamaño pero hay mayor, se asigna la mayor. También puede influir en la decisión de la carga la prioridad del proceso.

3.3.2.3 Sistema de compañeras

El sistema de compañeras (*buddy system*) desarrollado por D. Knuth (1997a), es un compromiso entre la asignación de tamaño fijo y la de tamaño variable . La memoria se asigna en unidades que son potencia de 2. Inicialmente, hay una única unidad de asignación que comprende a toda la memoria. Cuando se le debe asignar memoria a un proceso, se le asigna una unidad de memoria, cuyo tamaño es la menor potencia de 2 mayor que el tamaño del proceso. Por ejemplo, a un proceso de 50K se lo ubicará en una unidad de asignación de 64K. Si no existe una unidad de asignación de ese tamaño, la asignación disponible más chica mayor que el proceso se dividirá en dos unidades «compañeras» de la mitad del tamaño que la original. La división continúa con una de las unidades compañeras hasta que se crea una unidad de asignación del tamaño apropiado. Cuando un proceso libera memoria provoca que se liberen dos unidades compañeras, las unidades se combinan para formar una unidad dos veces más grande, tratando de ese modo de minimizar la fragmentación. La tabla 3.4 muestra cómo las unidades de asignación se dividirán y se juntarán a medida que la memoria es asignada y liberada.

Dado un tamaño de memoria de 2^N, un sistema de compañeras mantiene un máximo de N listas de bloques libres, una para cada tamaño (dado un tamaño de memoria de 2^8, son posibles unidades de asignación de tamaño 2^8 hasta 2^1). Con una lista separada de bloques disponibles para cada tamaño, la asignación o liberación de memoria puede ser procesada de forma eficiente. Este algoritmo tiende a ser un poco más eficiente en tiempo, sin embargo, el sistema de compañeras no usa la memoria de una forma eficiente a causa de tener tanto fragmentación interna como externa provocado por su enfoque del mejor ajuste.

3.3.2.3.1 Slab Allocator

Este algoritmo, con el llamativo nombre de «asignador de baldosas», fue desarrollado por Bonwick (1994) para el sistema operativo SunOS. Dentro del núcleo de un sistema operativo se asigna una considerable cantidad de memoria para un conjunto finito de objetos tales como descriptores de archivos y otras estructuras comunes. Bonwick observó que la cantidad de tiempo requerido para inicializar un objeto ordinario en el núcleo excedía la cantidad de tiempo requerido para asignarlo y liberarlo. Por lo tanto, en lugar de retornar la memoria

Acción	Memoria					
Comienzo	1M					
A solicita 150 K	A	256K			512K	
B solicita 100 K	A	B	128K		512K	
C solicita 50 K	A	B	C	64K	512K	
B libera	A	128K	C	64K	512K	
D solicita 200K	A	128K	C	64K	D	256K
E solicita 60K	A	128K	C	E	D	256K
C libera	A	128K	64K	E	D	256K
A libera	256K	128K	64K	E	D	256K
E libera	512K				D	256K
D libera	1M					

Tabla 3.4 – Sistema de compañeras.

liberada al contenedor global, propuso que la memoria siguiera inicializada para el propósito previsto, reteniendo su estado entre usos y eliminando así la fragmentación provocada por la asignación y liberación de memoria. Este algoritmo ha sido usado por distintos sistemas operativos, incluyendo a Linux hasta la versión 2.6.23.

3.3.2.4 Reubicación

Al movernos de un sistema de monoprogramación a otro de multiprogramación, es necesario tener en cuenta que en este último no se puede conocer con antelación la posición de memoria que ocupará un programa cuando se cargue en memoria para proceder a su ejecución, puesto que dependerá del estado de ocupación de la memoria, pudiendo variar, por lo tanto, en sucesivas ejecuciones del mismo. En un sistema con multiprogramación, es necesario realizar un proceso de traducción o **reubicación** de las direcciones de memoria a las que hacen referencia las instrucciones de un programa, para que se correspondan con las direcciones de memoria principal asignadas al mismo.

Las **direcciones físicas** son los números binarios que apuntan a posiciones en la memoria física (normalmente en RAM), es decir, a la circuitería electrónica de las celdas de memoria.

Las **direcciones lógicas** son las que utiliza el programa. El hardware de administración de memoria convierte las direcciones lógicas en direcciones físicas.

Las **direcciones relativas** son un caso particular de las direcciones lógicas, en las cuales las direcciones se expresan como una posición relativa a algún punto conocido, normalmente el principio del programa.

De esta manera, los programas pueden cargarse y descargarse de memoria a lo largo de la ejecución. Este proceso de traducción crea un **espacio lógico independiente** para cada proceso, proyectándolo sobre la parte correspondiente de la memoria principal de acuerdo con una función de traducción. Esta función la lleva a cabo un módulo específico del procesador llamado MMU (*Memory Management Unit*). El programa se carga en memoria desde el ejecutable sin modificaciones y durante su ejecución se traducen las direcciones generadas. El sistema operativo almacena asociado a cada proceso cuál es la función de traducción que le corresponde al mismo y en cada cambio de proceso debería indicar al procesador qué función debe usar para el nuevo proceso activo.

> La Unidad de Administración de Memoria, en adelante MMU (*Memory Management Unit*), es un componente hardware de la computadora, responsable de manejar los accesos a memoria solicitados por la CPU. Las funciones que realiza son las de traducción de direcciones virtuales a direcciones físicas, protección de la memoria, control de la caché, arbitraje del acceso al bus y, originariamente, cambio de banco.

3.3.3 Paginación simple

Una solución al problema de la fragmentación externa es permitir que el espacio de direcciones lógico de un proceso sea **no contiguo**. Así, se puede asignar memoria física a un proceso siempre que haya alguna disponible. Una forma de implementar esta solución es adoptar un esquema de **paginación**. Con este esquema de administración ya no es necesario tener el proceso completo cargado en memoria. El concepto de paginación puede acreditarse a los diseñadores del sistema Atlas, que ha sido descripto en los trabajos de Kilburn y col. (1961) y Howarth, Payne y Sumner (1961).

En un sistema que utiliza paginación simple, la memoria se divide en bloques de tamaño fijo llamados **marcos** (*page frames*). A los procesos se los divide en bloques llamados **páginas** (*pages*), los cuales tienen el mismo tamaño que los marcos. El tamaño de un marco (y, por ende, de una página) está determinado por el hardware. Cuando se carga un proceso en la memoria, el sistema operativo carga cada página en un marco sin usar, los cuales no necesariamente deben estar contiguos.

Una **tabla de páginas** almacena el número de marco de página asignado para cada página. Así, el número del marco en el cual la página N fue cargada es almacenado en la $N-$ésima entrada en la tabla de páginas. El tamaño de la página es una potencia de 2. Así, si una dirección lógica contiene L bits, y el tamaño de página es de 2^P *bytes*, P bits de una dirección especifican el desplazamiento dentro de una página y los restantes $L-P$ bits especifican el número de página. El hardware genera la dirección física añadiendo el desplazamiento de página al número de marco extraído de la tabla de páginas (fig. 3.9). Es totalmente transparente al proceso la traducción de la dirección lógica a una dirección física, incluyendo la división de la dirección lógica en el número de página y el desplazamiento.

En un sistema paginado se puede usar un registro de tamaño para atrapar las direcciones fuera de rango. Pero más frecuentemente se estila que la entrada de la tabla de páginas contenga un bit que indique «válido» o «inválido». Para el caso de un proceso que sólo tenga n páginas, sólo las primeras n entradas de la tabla de páginas estarán marcadas como válidas.

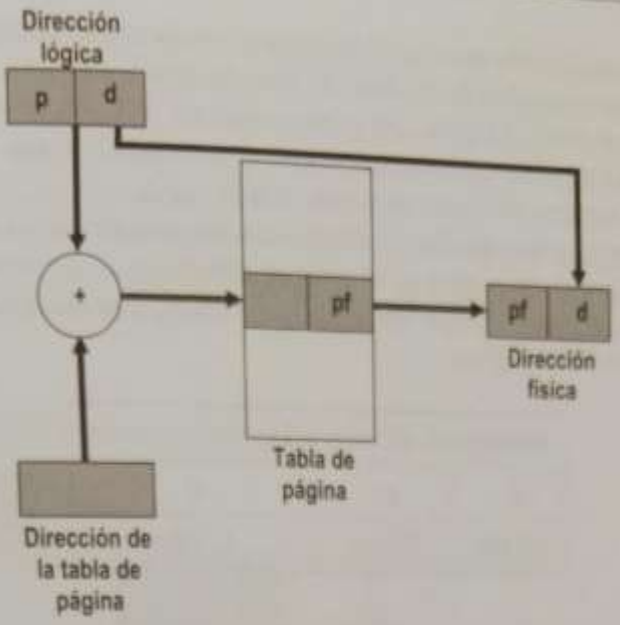


Fig. 3.9 – Paginación simple.

En algunos sistemas se pueden incluir también uno o más bits de protección en una entrada de la tabla de páginas. Sin embargo, las capacidades de protección están más comúnmente asociadas con los sistemas con segmentación.

En los sistemas en los que los procesos pueden tener un gran número de páginas se puede usar un sistema multinivel (fig. 3.10).

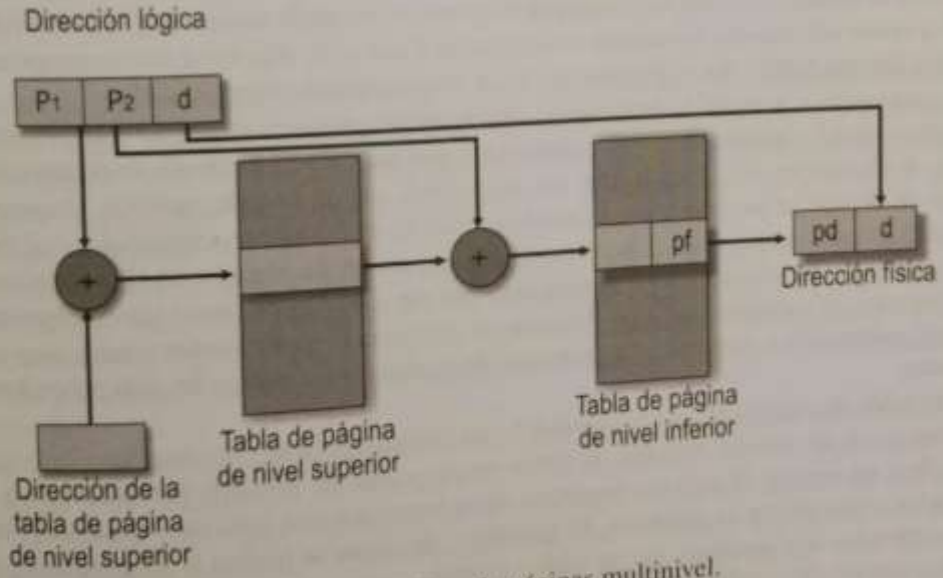


Fig. 3.10 – Tablas de páginas multinivel.

El número de página puede estar partido en dos o más componentes. El primer componente sirve como índice a la tabla de páginas de máximo nivel. La entrada en esa tabla apunta a la tabla del próximo nivel. El próximo componente en el número de página sirve como índice en esa tabla. El proceso continúa hasta que es accedida la tabla del último nivel. Esa entrada contendrá el número de marco asociado con la página.

Veamos el esquema en dos niveles. Supongamos una computadora con 32 bits de direcciones, 20 para el número de página y 12 para el desplazamiento. En un esquema de dos niveles, el número de página puede dividirse en 10 bits para el número de página y 10 para el desplazamiento dentro de la página.

Número de página		Desplazamiento
p_1	p_2	d
10 bits	10 bits	12 bits

- p_1 es el índice en la tabla de página inicial o más externa (*the outer page table*).
- p_2 es el desplazamiento dentro de la tabla externa (*outer table*).

Esto nos permite no asignar toda la tabla de página de manera contigua en la memoria principal.

3.3.4 Segmentación simple

Un aspecto importante de la gestión de memoria, que se hizo inevitable al surgir la paginación, es la separación entre la visión que el usuario tiene de la memoria y la memoria física real. La visión del usuario no es igual a la memoria física real; sólo tiene una correspondencia con ella. Dennis (1965) fue el primero en tratar el concepto de segmentación.

La segmentación, como la paginación, divide un programa en un número de bloques más pequeños llamados **segmentos** y cada uno de ellos se asigna a la memoria de forma independiente. A diferencia de la paginación, los segmentos son de tamaño variable. La responsabilidad de dividir un programa en segmentos recae en el usuario (o compilador), el sistema operativo no está involucrado. El hardware establece un límite superior en el tamaño de cualquier segmento. El programa objeto generado por un compilador puede tener diferenciados los segmentos de variables globales, de la pila o *stack* para procedimientos o funciones (almacenando parámetros y direcciones de retorno), de la porción de código de cada procedimiento y función.

Una tabla de segmentos es muy similar a una tabla de páginas. No obstante, dado que los segmentos son de tamaño variable, la memoria no puede ser dividida previamente en nada parecido a un marco. El sistema operativo debe mantener una lista de segmentos libres y asignarlos a los huecos en memoria. El problema de cómo se realiza la asignación es igual los encontrados en el esquema de partición variable.

Una entrada de la tabla de segmentos debe tener campos para almacenar la dirección de comienzo del segmento en memoria principal y el tamaño del mismo. Si el tamaño máximo del segmento es m bits, los últimos m bits de la dirección lógica especifican el desplazamien-

to del segmento. Los bits restantes especifican el número del segmento. La dirección lógica se traduce a una dirección física, extrayendo el número de segmento y el desplazamiento a partir de la dirección lógica. El número de segmento se usa como índice a la tabla de segmentos. El desplazamiento se compara con el tamaño del segmento. Si el desplazamiento es mayor que el tamaño del segmento, se genera un fallo por dirección inválida, provocando que se aborte el programa. Caso contrario, el desplazamiento es agregado a la dirección de comienzo del segmento para generar la dirección física (fig. 3.11). Para incrementar la velocidad, la comprobación del tamaño y la generación de la dirección física se pueden realizar concurrentemente.

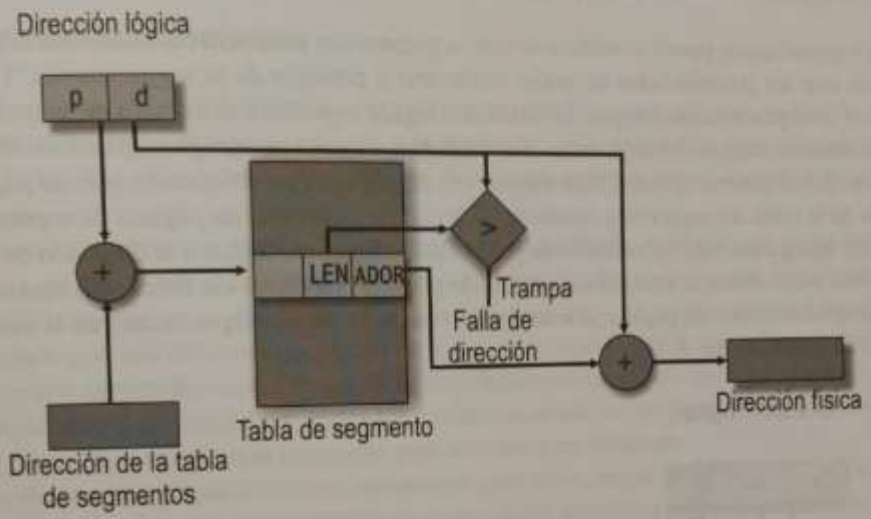


Fig. 3.11 – Segmentación simple.

Dado que los segmentos están definidos por el usuario (por el compilador, más bien), es posible definir que ciertos segmentos sean de sólo lectura. Agregando un bit que indique «sólo lectura» en una entrada de la tabla de segmentos, el sistema de administración puede comprobar las operaciones de escritura a segmentos de sólo lectura y generar un fallo si detecta tales operaciones.

Si existe el soporte para segmentos de sólo lectura, también puede hacerse posible que varios procesos compartan los segmentos de sólo lectura y que, por lo tanto, disminuyan el uso de la memoria. Típicamente, esto ocurre cuando dos procesos están ejecutando el mismo programa y el código para ese programa está diseñado para estar en uno o más segmentos de sólo lectura.

Los segmentos compartidos también pueden usarlos los procesos que ejecutan diferentes programas, pero que usan las mismas subrutinas de biblioteca. En esta situación, se debe tener cuidado que las direcciones dentro de un segmento compartido funcionen con ambos programas. Para las direcciones de ubicaciones dentro del segmento, la solución más simple es usar direccionamiento relativo. Este usa el desplazamiento a partir del valor actual del contador de programa para generar la dirección de destino. Para todas las direcciones también es una posibilidad utilizar direccionamiento indirecto a partir de un registro que apunte al segmento apropiado. El direccionamiento absoluto, en el cual se especifica el segmento y

desplazamiento de la dirección de destino, es posible solamente si todos los programas usan el mismo número de segmento.

Una ventaja importante de la segmentación es la posibilidad de implementar mecanismos de protección de una manera relativamente fácil. Por ejemplo, si definiéramos únicamente segmentos de instrucciones o datos, los de instrucciones (es decir, rutinas, procedimientos o funciones) pueden especificarse como de sólo lectura (*read only*) o sólo ejecución (*execute only*). Otra ventaja es la posibilidad de compartir segmentos (código o datos de sólo lectura).

3.3.5 Segmentación con paginación

La segmentación puede combinarse con la paginación para proveer la eficiencia de la paginación con las posibilidades de poder compartir y proteger de la segmentación. Tal como pasa con la segmentación simple, la dirección lógica especifica el número de segmento y el desplazamiento dentro del segmento. Sin embargo, cuando se agrega paginación, el desplazamiento del segmento se subdivide en número de página y un desplazamiento de página. La entrada de la tabla de segmentos contiene la dirección de la tabla de páginas de segmentos. El hardware agrega los bits del número de página de la dirección lógica a la dirección de la tabla de páginas para ubicar la entrada de la tabla de páginas. La dirección física se forma añadiendo el desplazamiento de página al número de marco de página especificado en la entrada de la tabla de páginas (fig. 3.12).

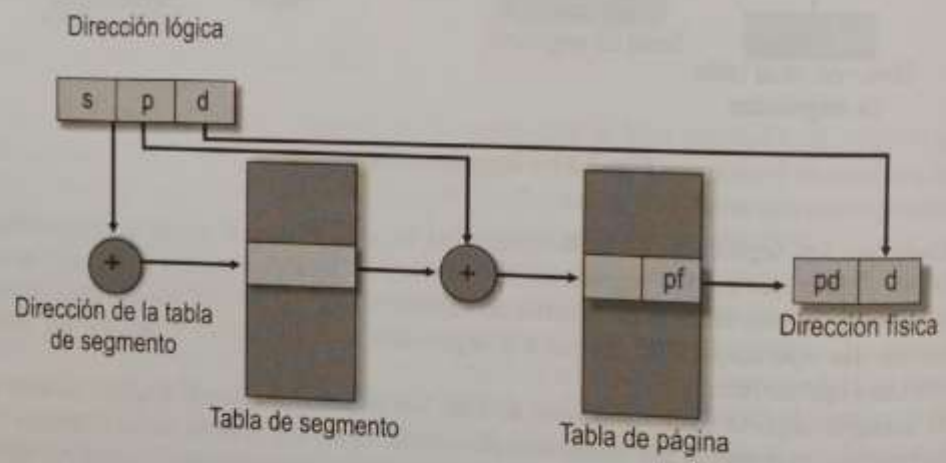


Fig. 3.12 – Segmentación con paginación.

El primer sistema que manejó la segmentación paginada fue el GE 645, en el que se implementó originalmente MULTICS y está publicado en la obra de Organick (1972).

3.3.6 Tablas de páginas y de segmentos

Cada proceso tiene sus propias tablas de páginas y/o segmentos almacenados en memoria por el sistema operativo. En algunos sistemas, un registro de administración de la memoria apunta a la tabla de segmento o de página para el actual proceso en ejecución. Cuando se le

otorga control de la CPU a un proceso nuevo, sólo se necesita reiniciar un único registro para conmutar al espacio de direcciones lógicas del nuevo proceso.

Si las tablas de páginas son suficientemente pequeñas, la unidad de administración de la memoria puede tener un conjunto de registros dedicados para mantener la tabla de páginas. Los registros, por lo general, son más rápidos que la memoria convencional y por esto se acelera el proceso de traducción de las direcciones, para lo cual se utilizan instrucciones privilegiadas.

El puntero a esa tabla, como a otros datos propios del proceso, son almacenados en el bloque de control del proceso (*Process Control Block* o PCB).

3.3.7 Memoria asociativa

En los sistemas que tienen tablas de páginas extremadamente grandes se puede usar la **memoria asociativa** (*Translation Lookaside Buffer* o TLB) que es una memoria ultrarápida (caché) de la CPU administrada por la MMU. Se la suele traducir como «*buffer* de traducción anticipada».

Cada elemento en una memoria asociativa está identificado mediante un valor índice. A diferencia de la memoria convencional –que se basa en el conocimiento exacto de la dirección de memoria en la que se encuentra la información–, la memoria asociativa no está referenciada por una dirección, sino con un valor de búsqueda (se le suele llamar «direccionamiento por contenido»); se revisan todos los elementos buscando uno con un valor índice correspondiente. La búsqueda de todos los elementos se realiza en paralelo, de manera que el tiempo de acceso es el tiempo necesario para acceder a un elemento.

Los sistemas de paginación usan memorias asociativas pequeñas y de alta velocidad para mejorar el rendimiento. Una entrada en la memoria asociativa debería ser idéntica a una entrada de la tabla de páginas, excepto que debería contener también un campo para el número de página. Cuando se especifica una página se busca la memoria asociativa al mismo tiempo que se accede a la tabla de páginas. Por esta forma de buscar se la denomina en inglés *lookaside*, que podríamos traducir como «mirando de reojo».

Si se encuentra una entrada en la memoria asociativa (*cache hit* o *TLB hit*) con el número de página coincidente, se aborta el acceso a la tabla de páginas y se usa la entrada de la memoria asociativa. Si no se encuentra una coincidencia en la memoria asociativa, se usa la entrada de la tabla de páginas (*page walk*), lo que tardará varios ciclos más, sobre todo si la página que contiene la dirección buscada no está en memoria primaria. Si en la tabla de páginas no se encuentra la dirección buscada, se provoca un fallo de página (fig. 3.13) que deberá manejar el sistema operativo.

A la misma vez que está siendo traducida la dirección lógica, la memoria asociativa reemplaza una de sus páginas, la entrada menos usada recientemente, con la entrada de la tabla de páginas para la páginas que está siendo accedida actualmente. De esta manera, la memoria asociativa mantiene una copia de las entradas de las páginas más usadas recientemente.

Cuando se le da el control de la CPU a un proceso nuevo (cambio de contexto), la unidad de administración de la memoria no debe usar entradas de la memoria asociativa de un proceso previo. La manera más fácil de prevenir esto se logra si el sistema operativo marca cada entrada en la memoria asociativa como inválida. Como alternativa, la memoria asociativa podría agregar un campo para contener un número de identificación del proceso. De manera tal

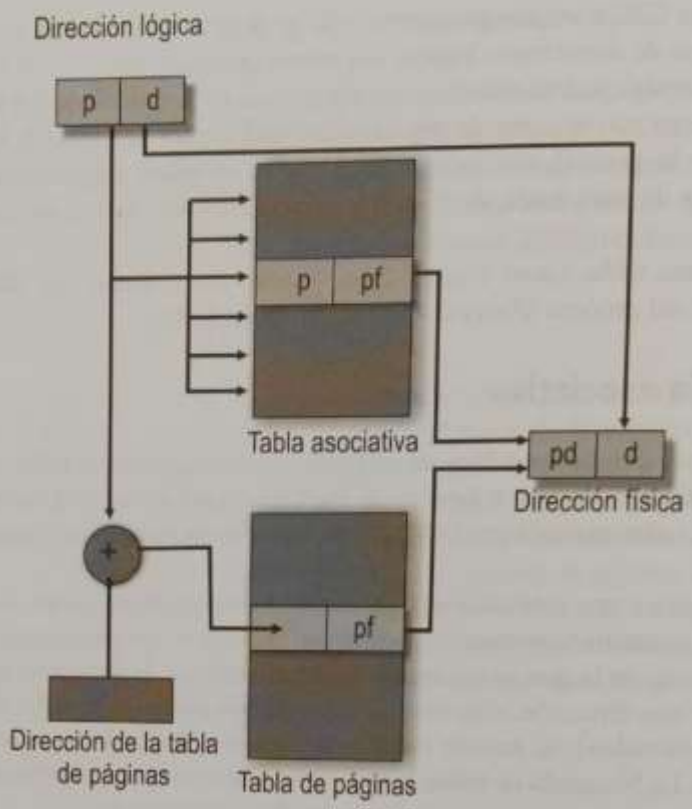


Fig. 3.13 – Memoria asociativa - TLB.

que, cada vez que se intenta un acceso, una coincidencia también implica una coincidencia entre el número de identificación de proceso con el valor en un registro especial que contenga al número de identificación del proceso actual.

La mayor mejora en el desempeño se logra cuando la memoria asociativa es significativamente más rápida que la búsqueda en la tabla de páginas normal y la **tasa de aciertos** (*hit ratio*) es alta.

> La tasa de aciertos es el cociente entre los accesos que encuentran una coincidencia en la memoria asociativa (los exitosos) y aquellos que no lo encuentran.

3.3.8 Tabla de páginas invertida

En el esquema visto hasta ahora tenemos una tabla de página por proceso, indexada por el número de página. El sistema operativo traslada la referencia virtual a una física. Debe calcular cómo acceder a ella.

En los sistemas paginados que tienen un gran espacio de direcciones virtual que usan una memoria asociativa, se puede usar una tabla de páginas invertida para reducir el tamaño de la tabla de páginas. En lugar de contar con una tabla de páginas que contenga una entrada para cada una de un número muy grande de páginas, la tabla contiene una entrada para cada

marco de página; por lo tanto, habrá tantas entradas en la tabla como marcos. La entrada almacena el número de página asignado a ese marco. Cuando una dirección no puede ser resuelta mediante una entrada en la memoria asociativa, se busca en la tabla invertida una entrada que contenga el número de página. Si bien se soluciona el problema de ocupar mucha memoria para las tablas de página, la búsqueda en las tablas invertidas puede llevar mucho tiempo, pues está ordenada por marco y puede hacer que se deba recorrer toda la tabla. Para acelerar la búsqueda se usan técnicas de *hashing*.

> *Hash* se refiere a una función o método para generar claves o llaves –que representen de manera casi unívoca a un documento, registro, archivo, etc.– y resumir o identificar un dato a través de la probabilidad, utilizando una «función *hash*» o «algoritmo *hash*». Un *hash* es el resultado de dicha función o algoritmo.

3.3.9 Ventaja adicional del paginado: las páginas compartidas

Sea un ambiente de tiempo compartido (*time sharing*). Tiene 15 usuarios de desarrollo que están usando el editor para escribir programas. El editor ocupa 150k y 50k de espacio de datos. Trabajando los 15 usuarios a la vez necesitaríamos (150k * 15) + (50K * 15) de memoria real para el uso simultáneo del editor. Si el **código** del editor es **reentrante** o **puro**, es decir, no se automodifica, puede compartirse. Supongamos tres procesos p_1, p_2 y p_3 que usan el editor. El editor consta de tres páginas, cada una de ellas de 50 K.

Al ser código reentrante uno o más procesos pueden ejecutarlo al mismo tiempo. Cada proceso tiene sus propios registros y sus datos. Así, sólo una copia del editor está en memoria. Ahora, para 15 usuarios necesitaremos (150K) + (50K * 15). Puede compartirse todo proceso que sea reentrante y, principalmente, el de uso común: compiladores, sistemas de bases de datos, etcétera. No es posible implementar páginas compartidas en un esquema de tabla de páginas invertida.

> **Código reentrante**: una función reentrante es una función que puede ser usada por más de una tarea sin temor a la corrupción de los datos. Una función reentrante puede ser interrumpida en cualquier momento y retomada en cualquier momento posterior sin pérdida de datos. Las funciones reentrantes usan variables locales (es decir, registros de la CPU o variables en la pila) o datos protegidos cuando se usan variables globales.

Un ejemplo de una función reentrante es la siguiente:

```
void strcpy(char *dest, char *src)
{
    while (*dest++ = *src++){
        ;
    }
    *dest = NUL;
}
```

A continuación se muestra un ejemplo de una función no reentrante (`var1` está declarada global). Dado que la función accede a una variable global, no es reentrante.

```
void foo(void)
{
  ...
  ...
  var1 += 23;
  ...
  ...
}
```

Sin embargo, la función se puede hacer reentrante, protegiendo la sección crítica del código (ver sección 5.2.1).

```
void foo(void)
{
  ...
  deshabilitar_interrupciones();
  var1 += 23;
  habilitar_interrupciones();
  ...
}
```

Por ejemplo, los compiladores específicamente diseñados para software integrado (*embedded*) típicamente proveen de bibliotecas reentrantes.

> Según M. E. Russinovich y D. A. Solomon (2004), Windows NT y posteriores son completamente reentrantes, partes significativas de Windows 95 no eran reentrantes, principalmente el código de 16 bits tomado de Windows 3.1. Este código no reentrante incluía a la mayoría de las funciones gráficas y de administración de la ventana. Cuando una aplicación de 32 bits sobre Windows 95 intentaba llamar a un servicio de sistema implementado en código de 16 bits no reentrante, debía obtener primero un candado global (*system wide lock*) o *mutex*, para evitar que otros hilos ingresaran a la base de código no reentrante. Peor aún, una aplicación de 16 bits mantenía a este candado mientras duraba su ejecución. Como resultado, a pesar de que el núcleo de Windows 95 contenía un planificador multi hilado de 32 bits desalojable, las aplicaciones a menudo ejecutaban de manera mono hilada porque una buena parte del sistema estaba todavía implementado en código no reentrante.

3.3.10 Intercambio

Puede ocurrir que el sistema operativo decida que un proceso en memoria salga temporariamente, porque necesita más espacio, para incrementar el número de procesos que comparten la CPU o por que llega un proceso de mayor prioridad.

> La técnica de llevar temporariamente un proceso a memoria secundaria se llama intercambio o *swapping* (ver Corbató, Merwin-Daggett y Daley 1962). Y al espacio en disco donde se almacena, se le suele llamar *backing store*.

Cuando se saca de memoria, se hace *swap-out*; cuando se trae nuevamente, *swap-in*.[2]

2. Como esta técnica ahora se usa en combinación con el paginado, los términos actualmente son *page-in* y *page-out*, respectivamente.

En el caso de un algoritmo de planificación de CPU basado en prioridades (ver 4.3.3), el manejador de memoria puede hacer *swap-out* a un proceso de menor prioridad para poder cargar a uno de mayor prioridad que llegue. Cuando termina, se vuelve a cargar el de menor prioridad. A esta variedad se la llama *roll-out/roll-in*.

El lugar donde se volverá a ubicar el proceso descargado depende del tipo de ligazón (*binding*) que haga: sólo puede cambiar de lugar si tiene *binding* en el momento de ejecución. *Backing storage* era normalmente un disco rápido (distinto del que almacenaba archivos de los usuarios), y con suficiente capacidad para mantener las imágenes de los procesos de los usuarios, y debía proveer acceso directo a estas imágenes. Actualmente, el mismo disco que almacena archivos de usuario también se usa para intercambio.

El sistema mantiene una cola de listos (ver 4.1.5) con aquellos procesos cuyas imágenes están listas para ejecutar en *backing store* o en memoria. Según veremos, cuando la CPU quiere ejecutar un proceso llama al *dispatcher*, que chequea si el próximo proceso en la cola está en memoria. Si no está y no hay memoria disponible, el *dispatcher* decidirá hacer *swap-out* de un proceso de memoria para hacer *swap-in* del proceso deseado. Luego actúa como siempre, transfiriendo el control al nuevo proceso.

El proceso de intercambio, que involucra cambios de contextos (*context switch*), debe ser muy rápido. El tiempo de ejecución de un proceso debe ser mucho mayor que el tiempo que toma hacer el intercambio. En un sistema con algoritmo de turno circular (*round-robin*) de CPU, el *quantum* deber ser mucho mayor que el tiempo de intercambio.

El tiempo de intercambio involucra el tiempo de transferencia hacia/desde el disco y, por lo tanto, es directamente proporcional a la cantidad de memoria que se transfiere.

Para intercambiar un proceso éste debe estar ocioso.

Otro tema es qué hacer con los procesos con entrada/salida pendiente, pues cuando termine la operación de entrada/salida, o *Input/Output* (I/O), puede ser que el proceso ya no esté en memoria y se quiera ubicar la información en un espacio que ahora pertenece a otro proceso. Las dos soluciones probables son:

- No intercambiar procesos con entradas o salidas pendientes.
- Utilizar para las entradas y salidas *buffers* del sistema operativo para la información y pasarla a memoria de usuario cuando el proceso vuelva a ser cargado en memoria.

En pocos sistemas se usa el *swapping* estándar, se usa más bien combinado con otras técnicas. La decisión acerca de qué proceso o procesos son intercambiados hacia/desde memoria principal le corresponde al planificador del sistema operativo.

3.4 Memoria virtual

Las alternativas de administración de memoria vistas cargan todo el proceso de manera contigua. Esto exige mayores espacios de memoria con secciones del proceso que no se utilizarán enseguida, pues la ejecución de un proceso es secuencial y se adapta al modelo de localidad. Esto significa que en un período de tiempo se usarán direcciones de memoria «cercanas». ¿Por qué no tener en memoria sólo la parte del proceso que se está ejecutando? Estas partes serían de menor tamaño y, por lo tanto, se podría aprovechar mejor la memoria. Es más: la medida del proceso o de la suma de los procesos listos puede ser mayor que la memoria disponible, pues no la ocupan toda a la vez.

Llegados a este punto de nuestro estudio, coincidente con la evolución que tuvieron los sistemas operativos y el hardware, se desprende el siguiente avance a partir de dos características de la paginación y la segmentación. Primero, que todas las referencias a la memoria dentro de un proceso son direcciones lógicas que se traducen dinámicamente a direcciones físicas durante la ejecución, de tal forma que un proceso puede cargarse y descargarse de la memoria principal ocupando regiones diferentes en distintos momentos de su ejecución. Y, segundo, que un proceso puede dividirse en varias partes y no es necesario que esas partes se encuentren contiguas en la memoria principal durante la ejecución.

Fotheringham 1961 creó esta técnica de administración para la computadora Atlas en la Universidad de Manchester y la llamó memoria virtual. La idea básica es que el tamaño del programa, los datos y la pila combinados pueden ser mayores que la memoria (real) disponible. El programador se desentiende de la cantidad de memoria real que necesitará su proceso, ya que el proceso hace uso de su propio **espacio de direcciones virtuales**. El sistema operativo guarda aquellas partes del programa de uso corriente en la memoria principal y el resto permanece en disco (ver fig. 3.14).

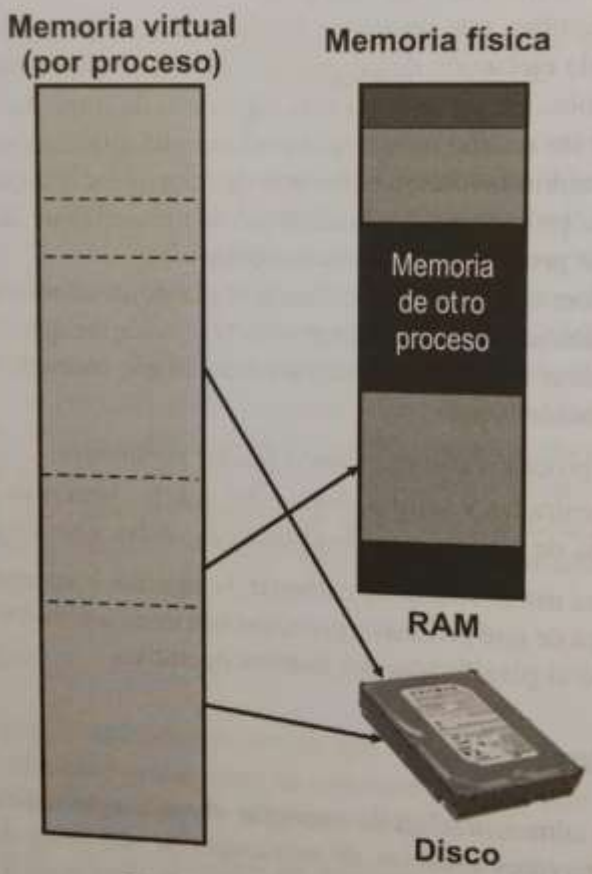


Fig. 3.14 – Memoria virtual.

La memoria virtual y la multiprogramación se corresponden bien entre sí, pues, por ejemplo, si tenemos 8 programas de 1 MB podemos asignar una partición de 256 KB en una memoria de 2 MB. Cada programa trabaja como si tuviera una máquina privada de 256 KB.

Además, la memoria virtual es útil, pues mientras un programa hace *swapping*, otro puede tener el procesador y, de esa manera, tenemos menos CPU ociosa.

En un sistema de memoria virtual, el espacio de memoria lógica disponible para un programa es totalmente independiente del espacio de memoria física. Un programa que necesite 2^{20} *bytes* de memoria puede ejecutar en un sistema con memoria virtual con solamente 2^{16} *bytes* de memoria real. La memoria virtual libera a los programas de las limitaciones de la memoria física.

3.4.1 Paginación por demanda

La paginación por demanda combina las características de la paginación simple y las superposiciones para implementar memoria virtual. En un sistema con paginación por demanda, cada página de un programa se almacena en forma contigua en el espacio de intercambio de paginación en almacenamiento secundario. A medida que se hace referencia a ubicaciones en páginas, estas páginas se copian en los marcos de página de memoria. Una vez que la página está en la memoria, se accede a ella como en la paginación simple.

Cada entrada en la tabla de páginas tiene como mínimo dos campos: el marco de la página y el bit dentro/fuera (*in/out bit*). Cuando se genera una dirección virtual, el hardware de administración de memoria extrae el número de página de la dirección y se accede la entrada correspondiente en la tabla de páginas. Se comprueba el bit dentro/fuera y, si la página está en memoria, se genera la dirección física, añadiendo el desplazamiento de página al número de marco de página. Si la página no está en memoria, se produce una trampa (*trap*) o excepción por **fallo de página** (*page fault*), transfiriendo el control a la rutina de fallos de página del sistema operativo.

Cuando se produce un fallo de página, el sistema operativo comprueba si están libres algunos marcos de página en memoria. Si no lo están, selecciona una página para eliminarla. La página marcada para eliminación se copia a almacenamiento secundario y el bit dentro/fuera en su entrada en la tabla de páginas se cambia a «fuera». Si hay un marco de página libre, el sistema operativo copia la página en el marco libre. La entrada de la tabla de páginas de la página ingresada en memoria se modifica indicando el número de marco y que la página está ahora en memoria. A continuación, se completa la rutina de tratamiento del fallo de página y el hardware vuelve a ejecutar la instrucción que generó la trampa.

Se le puede agregar un **bit sucio** (*dirty bit*) a cada entrada de la tabla de páginas. El hardware de administración de la memoria cambia el bit sucio cuando hay una referencia de escritura a esa página. Cuando se carga en memoria una página por primera vez, se «limpia» el bit sucio. Si el bit sucio de una página seleccionada para reemplazo está limpio, significa que la página no ha sido modificada desde que fue cargada en memoria. Sólo las páginas reemplazadas que han sido modificadas necesitan ser escritas nuevamente al espacio de intercambio.

Cuando no hay memoria virtual, la dirección generada por el proceso se coloca directamente en el bus de memoria. Cuando hay memoria virtual, la dirección pasa a la MMU.

En un esquema de paginado por demanda puro, el proceso comienza haciendo fallos de página con la primera instrucción que se ha de ejecutar, y así continúa, con una alta tasa de fallos, hasta que se estabiliza al tener todas las páginas necesarias en memoria. El momento en que se produce el fallo de página dentro del ciclo de instrucción es también un punto para

considerar, pues la mayoría de las veces se hace necesario volver a ejecutar la instrucción. Si ocurre en la captura o *fetch*, habrá que realizarlo nuevamente. Si ocurre al tratar de resolver un operando, se deberá rehacer el *fetch* de la instrucción, decodificar de nuevo el operando y, entonces, hacer el *fetch* del operando.

3.4.1.0.1 Ejemplo

Supongamos que tenemos una computadora de 32KB de memoria física y que puede generar direcciones de 16 bits, de 0 a 64K. Estas serían las direcciones virtuales. Las páginas y los marcos deben ser de igual tamaño. En nuestro ejemplo serán de 4K. Por lo tanto, tenemos 16 páginas virtuales y 8 cuadros de página.

Tabla páginas	
0-4K	2
4K-8K	1
12	6
16	0
20	4
24	3
28	
32	
36	
40	5
44	
48	7
52	
56	
60	
64	

Marcos mem real
0-4K
4-8
8
12
16
20
24
28

El programa quiere acceder a la dirección 0 mediante una instrucción, por ejemplo, `move reg,0`.

La dirección 0 se manda a la MMU, que observa que la dirección virtual 0 queda en la página 0 (0-4095) y que el cuadro de página es 2 (8192-12387).

Por lo tanto, transforma la dirección en 8192 y la pone en el bus. La memoria advierte que hay una solicitud de memoria a la dirección 8192. La instrucción `move reg, 8192` se transforma en `move reg,24576`, pues 8192 está en la página 2 (virtual) que se transforma en la 24576 porque apunta al cuadro de página 6. En un tercer ejemplo, la dirección virtual

20500 tiene 20 *bytes* desde el inicio de la página virtual 5 (20480-24575), por lo tanto, se transforma en 12288+20=12308 (la página virtual 5 apunta al cuadro de página 3).

Veamos cómo se transforma una dirección virtual de 16 bits en una física. Supongamos la dirección 8196:

0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0
N de pag virt				Desplazamiento											

En la tabla de páginas de la MMU se accede a la página 2 y de ahí se toma el valor, que es 6 (110), y se forma la dirección:

0	1	1	0	0	0	0	0	0	0	0	0	0	1	0	0

Esto ocurre si la página está en memoria, si no, es un *page fault*.

Los 4 primeros bits de la dirección virtual indican la página y los 12 bits restantes, desplazamiento.

De allí podemos deducir que puede haber 2^4 páginas y que el tamaño de las páginas será de 2^{12} *bytes*. La medida de la página la determina el HW, siendo siempre potencia de 2. Si la medida del espacio lógico de direcciones es una potencia m de 2 y la medida de la página en unidades de direccionamiento es una potencia n de 2, los bits de mayor orden m-n de la dirección lógica designan el número de página y los n de bajo orden designan el desplazamiento.

3.4.2 Localidad de referencia

Cuando hay referencias a páginas ya cargadas en memoria, la traducción de direcciones es realizada totalmente por el hardware, rindiendo un desempeño comparable a sistemas sin memoria virtual. Pero la referencia a páginas intercambiadas a disco provocan trampas de fallo de página, retardando el tiempo de acceso por más de un orden de magnitud (ver tabla 3.1)

Representamos el tiempo de acceso efectivo a memoria, T_{ae}, con la siguiente fórmula:

$T_{ae} = t_{am} \star (1-p) + t_{pf} \star p$

donde

t_{am}: tiempo acceso a memoria

p: probabilidad de *page fault*

t_{pf}: tiempo que dura el *page fault*.

De acuerdo con la fórmula enunciada arriba, el tiempo de acceso efectivo será directamente proporcional al tiempo de servicio de un *page fault*.

El rendimiento global sería inaceptable si las referencias a memoria fueran aleatorias. Sin embargo, las referencias tienden a estar localizadas en un pequeño conjunto de páginas. Esta **localidad** da como resultado que la tasa de aciertos de páginas en memoria, respecto de las que no lo están, sea suficientemente alta como para tiempos de acceso promedio aceptables. A la lista ordenada de números de páginas accedida por un programa se le llama su **cadena de referencias** (*reference string*).

Podemos ver dos formas de localidad. Por una parte, cuando se ha hecho referencia a una ubicación en memoria, hay mucha posibilidad de que sea referenciada otra vez a corto tiempo, a esto le llamamos **localidad temporal** (*temporal locality*); y, por otra, la **localidad**

espacial (*spatial locality*) se refiere al hecho de que si es accedida una ubicación en memoria, es probable que sea accedida una ubicación cercana a ella en la próxima instrucción.

Los experimentos han mostrado que, durante la ejecución de un proceso, las referencias tienden a agruparse en un número de ciertas localidades (Hatfield 1972). Por ejemplo, un programa que ejecuta un lazo accederá a un conjunto de páginas que contengan las instrucciones y datos referenciados en ese lazo. Las llamadas a funciones tenderán a incrementar los números de páginas en la localidad. Cuando el programa salga del lazo, se moverá a otra sección del código, cuya localidad de páginas es virtualmente distinta de la localidad previa.

3.4.3 Traba de páginas

El sistema de paginación debe ser capaz de trabar páginas en memoria de manera tal que no puedan ser intercambiadas a almacenamiento secundario. Gran parte del sistema operativo debe estar «encerrado con llave» en memoria. Y lo más importante es que el código que selecciona la próxima página que debe ser intercambiada a memoria nunca debería estar intercambiada a almacenamiento secundario, porque nunca podría ejecutarse a sí misma para ser intercambiada de regreso a la memoria.

> ***Aclaración:*** en la bibliografía en inglés se utilizan los términos *lock* y *block*, ambos traducidos al español como «bloqueo». En español utilizamos el término «bloquear» en el sentido de «interceptar», «obstruir» o «impedir», pero el término *lock* se refiere a colocar una llave o candado a algo, situación que puedo revertir por mí mismo. En cambio, si me encuentro impedido u obstruido, no es tan sencillo salir por mí mismo de la situación, porque normalmente obedece a factores externos. Por ejemplo, puedo salir de casa por mí mismo si abro la cerradura (*unlock*), pero no puedo salir por mí mismo de un atasco de tránsito.

El diseño del sistema de entrada y salida puede requerir también que permanezcan en memoria algunas de las páginas del proceso. Si las operaciones de entrada y salida de un dispositivo pueden leer o escribir datos directamente a una ubicación de memoria del proceso, las páginas que contengan esas ubicaciones de memoria no deberían estar intercambiadas a almacenamiento secundario. Si bien pueden no estar actualmente referenciadas por el proceso, están siendo referenciadas por el dispositivo de entrada/salida. Al colocar un bit cerrojo (*lock bit*) en una entrada de una tabla de páginas, prevenimos que la página sea intercambiada a almacenamiento secundario.

A pesar de que se debe tener cuidado con evitar que algunas páginas sean intercambiadas a almacenamiento secundario, no es necesario el bit cerrojo en todos los casos. El sistema debe estar diseñado de manera tal que ninguna página del sistema operativo sea intercambiada a almacenamiento secundario. El sistema de entrada y salida podría estar diseñado de manera tal que no permita la entrada/salida al espacio de direcciones del proceso directamente, se les debería requerir a todos los dispositivos que transfieran los datos desde y hacia un *buffer* del sistema operativo. En un sistema como éste, el intercambio se limita a las páginas de procesos de usuario y el sistema puede diseñarse de manera que cualquier página de usuario pueda ser intercambiada en cualquier momento.

3.4.4 Tamaño de la página

Un número de factores afectan la determinación del tamaño de la página:

- El tamaño de la página siempre es una potencia de 2, con rangos que van desde los 512 *bytes* hasta los 16K.
- Un tamaño pequeño de página reduce la fragmentación interna.
- Un tamaño grande de página reduce el número de páginas que se necesitan, reduciendo el tamaño de la tabla de páginas. Una tabla de páginas más pequeña requiere menos memoria (al espacio perdido para almacenamiento de la tabla de páginas se le conoce como **fragmentación de la tabla**). La tabla de páginas debe cargarse en registros de páginas, el tiempo de carga se reduce al tener tablas más pequeñas.
- Un tamaño grande de página reduce la sobrecarga involucrada en el intercambio de páginas desde y hacia memoria. Además del tiempo de procesamiento requerido para manejar un fallo de página, la operación de entrada/salida incluye el tiempo requerido para mover la cabeza de lectura/escritura del disco a la pista apropiada (tiempo de búsqueda o *seek time*), el tiempo requerido para que el sector gire hasta quedar debajo de la cabeza de lectura/escritura (tiempo de latencia o *latency time*) y el tiempo para leer la página (tiempo de transferencia o *transfer time*). Dado que los tiempos de búsqueda y latencia, con frecuencia, representan más del 90% del tiempo total, la lectura de dos bloques de 1K puede llevar casi el doble de tiempo que la lectura de un bloque de 2K.
- Un tamaño de página más pequeño, con su resolución más fina, está en mejores condiciones de acertarle a la localidad de referencias del proceso. Esto reduce la cantidad de información no utilizada, copiada de ida y vuelta entre la memoria y el almacenamiento de intercambio. También reduce la cantidad de información no utilizada almacenada en memoria principal, dejando más memoria disponible para fines útiles.

3.4.5 Algoritmos de reemplazo de páginas

El algoritmo que selecciona la página para ser intercambiada a almacenamiento de intercambio se llama **algoritmo de reemplazo de páginas**. El algoritmo de reemplazo de páginas óptimo selecciona para remover a la página que no será referenciada de nuevo en lapso más largo de instrucciones ejecutadas. Desafortunadamente, este algoritmo es solamente teórico, el implementarlo requeriría conocimientos de lo que ocurrirá en el futuro. Por lo tanto, su uso está limitado a servir como punto de referencia (*benchmark*), con el cual puedan ser comparados otros algoritmos de reemplazo de páginas.

El algoritmo de reemplazo de páginas FIFO (*First-in First-out*) selecciona a la página que ha estado en memoria por más tiempo. Para implementar este algoritmo, las entradas de la tabla de páginas deben incluir un campo para el tiempo en el que fue cargada en memoria. Cuando se carga la página, el sistema operativo graba el campo con la hora actual. La página seleccionada para reemplazo será la que tenga la hora de carga más temprana. A pesar de que es fácil de implementar, FIFO no es muy eficiente. A las páginas usadas con frecuencia, aun si han estado en memoria por mucho tiempo, obviamente no se las debería intercambiar. FIFO no toma en consideración la cantidad de veces que se han usado y las intercambia de todas formas.

El algoritmo del **Menos Usado Recientemente** (*Least Recently Used* o LRU) mantiene el registro de la última vez que fue usada cada página, no cuando fue cargada en memoria. El hardware de administración de la memoria usa un contador que se incrementa durante cada referencia a memoria. Cada entrada de la tabla de páginas tiene un campo que almacena el valor del contador. Cuando se hace referencia a una página, se almacena el valor del contador en la correspondiente entrada de la tabla de páginas. El algoritmo LRU busca en la tabla una entrada con el valor de contador más bajo y selecciona esa página para reemplazo.

En un sistema de n marcos de página, una matriz n x n provee un método alternativo para implementar el algoritmo LRU. A la matriz se la inicializa con ceros, cuando se accede el marco de página k todos los bits de la fila k se ponen en 1, luego todos los bits en la columna k se ponen en cero. Si la fila i contiene el valor binario más bajo, entonces el marco de página i es el menos usado recientemente.

El algoritmo **No Usado Recientemente** (*Not Recently Used* o NRU) es una aproximación a LRU. Además del bit sucio, cada entrada de la tabla de páginas contiene el **bit de referencia**. Cuando se carga una página, el sistema operativo pone el bit de referencia en cero. Cuando una página es accedida, el hardware pone el bit en 1. Además, a intervalos periódicos, el sistema operativo pone todos los bits de referencia en la tabla de páginas de vuelta en cero. Un valor de cero en el bit de referencia significa que no ha sido referenciado «recientemente» y un 1 significa que sí.

Las entradas pueden dividirse en conjuntos dependiendo del valor en el bit de referencia y el bit sucio. La primera prioridad es seleccionar una página con su bit de referencia en cero. La segunda prioridad es elegir una página cuyo bit sucio sea cero, ya que intercambiar a almacenamiento una página sucia consume más tiempo que intercambiar una página limpia. El sistema operativo es libre de elegir cualquier página a partir de varias páginas, con los mismos valores en el bit de referencia o sucio.

Otra aproximación a LRU llamada **envejecimiento** (*aging*) se logra al agregar un *byte* de referencia a las entradas de la tabla de páginas. Cuando se accede una página, el hardware pone en 1 el bit más significativo en el *byte* de referencia. Como en NRU, se usan interrupciones de temporizador (*timer interrupts*) para activar una rutina del sistema operativo, a pesar de que el intervalo de interrupción es menor en este algoritmo que en NRU. El sistema operativo desplaza todos los bits a la derecha un lugar en los *bytes* de referencia. Se selecciona para eliminar a una página con el valor binario más bajo en su *byte* de referencia. Tal como en NRU, el sistema operativo puede elegir cualquier página entre todas las páginas con el valor más bajo.

El algoritmo de reemplazo de la **Segunda Oportunidad** (*Second Chance*) es una combinación de los algoritmos FIFO y NRU. Selecciona a la página más vieja como candidata para remover y la remueve si su bit de referencia es cero. Sin embargo, si su bit de referencia es 1, su hora de carga se restablece a la hora actual y su bit de referencia se pone en cero. El algoritmo luego se repite con la segunda página más vieja, siendo ahora la más vieja. El proceso continúa hasta que es seleccionada una página.

3.4.6 El rendimiento de los algoritmos

El rendimiento de los diferentes algoritmos se puede comparar dadas diferentes secuencias de referencias de memoria. Tal como vimos, a la lista de páginas accedidas por un pro-

ceso se la llama **cadena de referencia** (*reference string*). Dados un algoritmo, una cadena de referencia y el número de marcos de página, se puede determinar el número de fallos de página disparados y éste se puede usar como la base para comparar los algoritmos.

También se puede comparar el rendimiento de un algoritmo en particular dados diferentes números de marcos de página asignados a un proceso. Uno podría esperar que al incrementar el número de marcos de página debería provocar la misma o menor fallos de página. Pero éste no es siempre el caso; bajo ciertas circunstancias ocurre una situación llamada **la anomalía de Belady**, en la que a mayor marcos de página ocurren más fallos de página y está documentado en el trabajo de Belady, Nelson y Shedler (1969). El algoritmo óptimo no sufre de esta anomalía; el LRU tampoco. En cambio, puede ocurrir en el FIFO.

3.4.7 Políticas de asignación

El hardware de la máquina es el que dicta el número mínimo absoluto de marcos que se le pueden asignar a cualquier proceso. Cuando ocurre un fallo de página, se carga la página necesaria en memoria y la instrucción se ejecuta de nuevo. Si la instrucción hace referencia a dos direcciones pero el proceso es asignado solo un único marco, ocurrirá un ciclo infinito de fallos de página. En las máquinas con direccionamiento indirecto y con instrucciones con operandos múltiples, una única instrucción puede hacer referencia a direcciones en varios marcos de página.

Por motivos de rendimiento, el número mínimo de marcos por proceso generalmente es mayor que el mínimo absoluto que dicta el hardware. No obstante la anomalía de Belady, por lo general el rendimiento mejora a medida que se incrementa el número de marcos de página asignados. Cualquiera que sea el mínimo, se puede lograr sólo mediante la limitación del número de procesos asignados a marcos de página. Esta es la tarea que realiza el planificador de largo plazo, ya que limita el número de procesos que comparten marcos de página, de manera tal que la frecuencia de fallos de página dan un nivel aceptable de rendimiento.

Es el sistema operativo el que debe decidir cuántos marcos de página asignar a cada proceso. Una estrategia llamada **distribución equitativa** (*equal allocation*) divide a los marcos de página disponibles en partes iguales entre los procesos.

Dados m marcos y p procesos, la distribución equitativa divide los m marcos por los p procesos, es decir $\frac{m}{p}$ marcos para cada proceso. Si la división no fuera exacta, podrían utilizarse los marcos del resto como depósito de marcos libres.

Pero la limitación radica en si están compitiendo dos procesos, de los cuales uno es chico y le bastan pocos marcos y al otro proceso se le hacen necesarios muchos, no se justifica darle a ambos la misma cantidad de marcos: el chico tendrá los suyos desocupados, mientras el otro tiene una alta tasa de fallas de páginas.

La **distribución proporcional** (*proportional allocation*) divide a los marcos en proporción a los tamaños de los procesos. En este algoritmo, la memoria disponible se asigna proporcionalmente a lo que el proceso necesita.

Sea v_i el espacio virtual del proceso p_i.

Sea V la suma de los espacios virtuales de los procesos.

O sea: $V = \Sigma v_i$

Sea m el números de marcos disponibles.

Entonces la asignación de marcos a_i para el proceso p_i sería:

$a_i = \frac{v_i}{V * m}$

a_i debe ser entero, mayor al número mínimo de marcos y no debe exceder los marcos disponibles.

En cualquiera de ambas asignaciones, se depende del nivel de multiprogramación. Al aumentar el grado de multiprogramación se deberán ceder marcos libres para el nuevo proceso. Si decrece, los procesos pueden utilizar los marcos liberados por el proceso saliente.

Otro punto para considerar es la prioridad de los procesos que compiten por marcos. Es decir, podrían otorgársele más marcos a un proceso de mayor prioridad para que finalice antes. En algunos sistemas, se plantea la asignación proporcional en la que la variable es la prioridad y no el tamaño del proceso, o en combinación, el tamaño y la prioridad. También un proceso de mayor prioridad podría usar para su reemplazo un marco de un proceso de prioridad menor.

3.4.7.0.2 Ejemplo

Siendo dos procesos: P_1, de 15 K y P_2, de 135 K, la cantidad de memoria disponible es de 70 K que se reparte en 70 marcos.

Por asignación equitativa: Se le otorgan 35 marcos a cada uno, aunque P_1 usará sólo 15K y el resto estarán libres pero no disponibles. A P_2 se le otorgarán 35 marcos: para los 135 que necesita, tendrá una alta tasa de fallas.

Por asignación proporcional: V=15+135; V=150

$\text{Marcos}(P_1) = \frac{15}{150 \times 70} = 7$

$\text{Marcos}(P_2) = \frac{135}{150 \times 70} = 63$

En ambos casos, a cada proceso se le asigna un número fijo de marcos. La asignación fija puede usarse con un algoritmo de reemplazo de **ámbito local** (*local scope*). Un algoritmo de reemplazo de ámbito local selecciona para remover a una página perteneciente al proceso que generó el fallo de página.

Alternativamente, un sistema operativo puede emplear un algoritmo de **ámbito global** (*global scope*) y **asignación variable** (*variable allocation*). En un sistema así, cuando el algoritmo de reemplazo busca una página para remover, selecciona la página a partir de todos los procesos en la máquina. El número de marcos asignados a un proceso variaría, ya que se miden con respecto a las páginas de otros procesos. Si el algoritmo selecciona una página para remover que haría caer al proceso por debajo de la asignación mínima, o bien se selecciona otra página o bien se intercambian a almacenamiento secundario todas las páginas del proceso.

Una estrategia de compromiso es utilizar asignación variable con ámbito local. El número de páginas asignadas a un proceso varía de acuerdo con las necesidades del proceso. Sin embargo, la página que ha de ser reemplazada se selecciona siempre a partir del proceso que está ejecutando actualmente. El número de páginas asignadas a un proceso está basado en el **conjunto de trabajo** (*working set*) del proceso.

3.4.7.1 Hiperpaginación

Aunque técnicamente es posible reducir el número de marcos asignados al mínimo, hay cierto número (mayor) de páginas que están en uso activo. Si el proceso no cuenta con este número de marcos, causará muy pronto un fallo de página. En ese momento, el proceso deberá reemplazar alguna página y, dado que todas sus páginas están en uso activo, deberá reemplazar una que volverá a necesitar de inmediato. Por lo tanto, se generará rápidamente otro fallo de página, y así sucesivamente. El proceso seguirá causando fallos, reemplazando páginas por las que entonces causará otro fallo y traerá de nuevo a la memoria.

> Al rendimiento degradado motivado por el intercambio excesivo se lo conoce como **hiperpaginación** (*thrashing*, literalmente «paliza» o «fustigamiento»). Un proceso está hiperpaginando si pasa más tiempo paginando que ejecutando.

3.4.8 Conjunto de trabajo

Denning 1968 define al conjunto de trabajo de un proceso en un determinado tiempo como al conjunto de páginas referenciadas en un cierto intervalo de tiempo precedente. Con frecuencia, se expresa el conjunto de trabajo usando la notación funcional $W(t, \Delta)$, donde W representa al conjunto de trabajo, t representa el tiempo y Δ representa el intervalo. Usualmente, es más simple para el tiempo el ser medido en términos de instrucciones ejecutadas (una unidad de tiempo de 1 significa una instrucción ejecutada).

El objetivo es elegir un valor para Δ de manera tal que el conjunto de trabajo refleje la localidad del programa. Si el valor de Δ es demasiado pequeño, el conjunto de trabajo no incluirá todas las páginas en la localidad actual. Si el valor de Δ es demasiado grande, el conjunto de trabajo incluirá páginas de una localidad previa.

A pesar de que el conocimiento de las páginas del conjunto de trabajo de un proceso es de valor insignificante para el sistema operativo, el conocimiento del tamaño del conjunto de trabajo se puede usar para determinar cuántas páginas asignar a cada proceso. El número de procesos asignados a memoria se ajusta de manera tal que a cada proceso se le asigna el número de marcos indicados por el tamaño del conjunto de trabajo.

3.4.9 Prepaginado

Si el sistema operativo conoce el conjunto de trabajo al momento de que el proceso fue intercambiado a almacenamiento secundario, puede **prepaginar** (*prepage*) todas las páginas en ese conjunto de trabajo cuando el proceso sea intercambiado nuevamente a memoria. El prepaginado previene que la referencia inicial a las páginas del conjunto de trabajo generen un fallo de página. Esto le ahorra al sistema operativo la sobrecarga extra de procesar esos fallos de página. Sin embargo, algunas de las páginas cargadas pueden nunca ser referenciadas. Prepaginar una página que no es referenciada, degrada el rendimiento de la entrada/salida y desperdicia marcos de página.

También se puede usar el prepaginado al momento de un fallo de página. Además de cargar la página que generó el fallo, también se puede cargar a la página siguiente en memoria virtual. Para aquellos programas cuya ejecución es primordialmente secuencial, el acceso a

una página puede ser un predictor de un acceso a la página siguiente. El costo en tiempo por intercambiar ambas páginas es mínimo, ya que la latencia del disco y el tiempo de búsqueda son los mismos (se hayan intercambiado una única página o dos páginas contiguas). La mayor desventaja de esta forma de prepaginado son los marcos desperdiciados por páginas que son cargadas pero nunca referenciadas. A causa de esta desventaja, esta técnica no ha mostrado ser efectiva en general.

3.4.10 Segmentación

Así como la paginación simple puede modificarse para crearse paginación por demanda, la segmentación también se puede modificar para crear segmentación por demanda. Sin embargo, la variabilidad de los tamaños de los segmentos complica muchos de los problemas encontrados en la paginación por demanda. En la decisión del intercambio, el tamaño de los segmentos se convierte en un factor importante para decidir qué segmentos intercambiar. Toda la noción del conjunto de trabajo se distorsiona por los tamaños diferentes de los segmentos. Iacobucci (1988) describió la segmentación por demanda y los detalles del OS/2.

Una forma mucho más práctica de implementar segmentación en un sistema de memoria virtual es combinarlo con paginación por demanda. O bien los segmentos siempre se consideran intercambiados en memoria, o bien el estado de intercambio de un segmento depende de si su tabla de páginas está intercambiada en memoria. Por lo tanto, la manipulación de los segmentos trabaja básicamente del mismo modo que lo hace en un sistema de paginación y segmentado simples.

> La capacidad de memoria virtual se crea por paginación por demanda de los segmentos.

3.5 Trabajos prácticos

3.5.1 Práctica con Linux

3.5.1.1 Estadísticas de la memoria virtual en Linux

El término «intercambio» (*swap*) se refiere estrictamente a guardar el proceso entero, no una parte, al disco[3]. Esta forma de intercambio es rara de encontrar, sino que se usa en combinación con la paginación.

Un *page-in* es un hecho común, normal y no es motivo de preocupación. Cuando arranca una aplicación, su imagen se va cargando en memoria haciendo *page-in*, esto es normal. Pero el *page-out* puede ser señal de problemas. Cuando el núcleo detecta que se está quedando sin memoria intenta liberarla haciendo *page-out*. Si bien esto puede ocurrir brevemente y de tanto en tanto, si esta situación se vuelve constante podemos caer en hiperpaginación.

Tanaka 2005 nos muestra cómo el programa `vmstat`, tal como su nombre sugiere, muestra estadísticas de la memoria virtual. Cuánta memoria virtual hay, cuánta está libre y la

3. Cabe aclarar en este punto que Linux normalmente usa una **partición** para intercambio, en tanto que Windows utiliza **archivos**.

actividad de paginación; se pueden observar los *page-in* y los *page-out* a medida que ocurren.

Es mejor si nos muestra la tabla actualizándola periódicamente, el retardo o *delay* se lo podemos pasar como primer parámetro y como segundo, el conteo para que no nos lo muestre indefinidamente. Por ejemplo, si colocáramos el comando:

```
vmstat 5 10
```

podríamos ver una salida similar a ésta:

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 1183880 104564 545096    0    0    56    15  344  365  3  2 92  2  0
 0  0      0 1179308 104572 549592    0    0     0     4  272  329  1  1 98  0  0
 0  0      0 1183572 104580 545112    0    0     0     7  247  266  0  0 98  1  0
 2  0      0 1183612 104592 545092    0    0     0     8  249  285  0  0 98  1  0
 0  0      0 1183644 104592 545080    0    0     0     0  214  232  0  0 99  0  0
 1  0      0 1183652 104592 545072    0    0     0    18  266  314  1  0 99  0  0
 0  0      0 1183652 104600 545108    0    0     0    14  208  265  1  0 98  1  0
 0  0      0 1183768 104600 545064    0    0     0     1  212  276  1  0 99  0  0
 0  0      0 1183644 104604 545072    0    0     0     1  190  256  1  0 98  1  0
 0  1      0 1173544 104612 553040    0    0  1589    29  539  501  3  2 90  5  0
```

En la que se nos muestran 10 líneas de estadísticas tomadas cada 5 segundos. Las columnas están explicadas en el manual (`man vmstat`) pero las que nos interesan son `free`, `si` y `so`. La columna `free` nos muestra la cantidad de memoria libre, `si` nos muestra los *page-in* y `so` los *page-out*.

Es decir que nos muestra que hay suficiente memoria libre. Como en este momento no estamos comenzando a ejecutar una aplicación nueva, no vemos *page-in*; luego tampoco vemos *page-outs*. Pero si comenzáramos a ejecutar varias aplicaciones que requirieran de mucha memoria, y colocáramos nuevamente el mismo comando, la situación cambiaría y nos mostraría una salida similar a ésta:

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0  13344   1444   1308  19692    0  168   129    42 1505  713 20 11 69
 1  0  13856   1640   1308  18524   64  516   379   129 4341  646 24 34 42
 3  0  13856   1084   1308  18316   56   64    14     0  320 1022 84  9  8
```

Note los valores no nulos de `so`, están indicando que no hay suficiente memoria física y el núcleo está generando *page-outs*. Como vimos en la práctica 2.4.1.1, podemos usar `top` y `ps` para ver cuáles son los procesos que más recursos están consumiendo. Pero también podemos usar el mismo programa `vmstat` para ver otras estadísticas interesantes. Instamos al lector a que experimente libremente una vez que haya leído las páginas del manual.

En Linux contamos con los comandos administrativos `swapon` y `swapoff` para activar y desactivar respectivamente dispositivos o archivos definidos como áreas de intercambio.

En el pseudo sistema de archivos accesible en el directorio `/proc` tenemos al archivo `/proc/swaps` que mide el espacio de intercambio y su utilización. Por ejemplo, en un sistema con una única partición de intercambio, si colocamos este comando:

```
$ cat /proc/swaps
```

la salida será similar a ésta:

```
Filename                Type            Size    Used    Priority
/dev/sda5               partition       2104472 0       -1
```

En la que podemos ver el nombre del archivo de intercambio, el tipo de espacio de intercambio, el tamaño total y la cantidad de espacio en uso, medidos en *kilobytes*. La columna de `Prioridad` es útil cuando se usan varios archivos o particiones de intercambio; a menor número mayor es la preferencia de uso.

3.5.1.2 El programa mapa.c

El siguiente programa ilustra los segmentos

```
#include <stdio.h
#include <stdlib.h
int uig;
int ig=5;
int func()
{

     return 0;

}
int main()
{

     int local;
     int *ptr;
     ptr=(int *) malloc(sizeof(int));
     printf("Una direccion del BSS:%p\n", &uig);
     printf("Una direccion del segmento de datos:%p\n", &ig);
     printf("Una direccion del segmento de codigo:%p\n", &func);
     printf("Una direccion del segmento de pila:%p\n", &local);
     printf("Una direccion del monticulo:%p \n", ptr);
     printf("Otra direccion de la pila:%p\n",%ptr);
     free(ptr);
     return 0;

}
```

3.5.1.3 La estructura de ELF

3.5.1.4 El archivo /proc/pid/maps

Basándonos en lo que vimos en la Sección 3.2, veremos ahora que el sistema operativo Linux ofrece un tipo de sistema de archivos muy especial: el sistema de archivos `proc`. Este sistema de archivos no tiene soporte en ningún dispositivo. Su objetivo es poner a disposición del usuario datos del estado del sistema en la forma de archivos. Esta idea no es original de Linux, ya que casi todos los sistemas UNIX la incluyen. Sin embargo, Linux se caracteriza por ofrecer más información del sistema que el resto de variedades de UNIX. En este sistema de archivos se puede acceder a información general sobre características y estadísticas del

sistema, así como a información sobre los distintos procesos existentes. La información relacionada con un determinado proceso se encuentra en un directorio que tiene como nombre el propio identificador del proceso (*pid*). Así, si se pretende obtener información de un proceso que tiene un identificador igual a «1234», habrá que acceder a los archivos almacenados en el directorio «/proc/1234/». Para facilitar el acceso de un proceso a su propia información, existe, además, un directorio especial, denominado «self». Realmente, se trata de un enlace simbólico al directorio correspondiente a dicho proceso. Así, por ejemplo, si el proceso con identificador igual a «2345» accede al directorio «/proc/self/», está accediendo realmente al directorio «/proc/2345/».

En el directorio correspondiente a un determinado proceso existe numerosa información sobre el mismo. Sin embargo, en esta práctica nos vamos a centrar en el archivo que contiene información sobre el mapa de memoria de un proceso: el archivo «maps». Cuando se lee este archivo, se obtiene una descripción detallada del mapa de memoria del proceso en ese instante. Como ejemplo, se incluye a continuación el contenido de este archivo para un proceso que ejecuta el programa «cat».

Coloque el comando:

```
cat /proc/self/maps
```

Y verá una salida similar a ésta:

```
08048000-0804a000 r-xp 00000000 08:01 65455 /bin/cat
0804a000-0804c000 rw-p 00001000 08:01 65455 /bin/cat
0804c000-0804e000 rwxp 00000000 00:00 0
40000000-40013000 r-xp 00000000 08:01 163581 /lib/ld-2.2.5.so
40013000-40014000 rw-p 00013000 08:01 163581 /lib/ld-2.2.5.so
40022000-40135000 r-xp 00000000 08:01 165143 /lib/libc-2.2.5.so
40135000-4013b000 rw-p 00113000 08:01 165143 /lib/libc-2.2.5.so
4013b000-4013f000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

Cada línea del archivo describe una región del mapa de memoria del proceso. Por cada región aparece la siguiente información:

- Rango de direcciones virtuales de la región (en la primera línea, por ejemplo, de la dirección «08048000» hasta «0804a000»).
- Protección de la región: típicos bits «r» (permiso de lectura), «w» (permiso de escritura) y «x» (permiso de ejecución).
- Tipo de compartimiento: «p» (privada) o «s» (compartida). Hay que resaltar que en el ejemplo todas las regiones son privadas.
- Desplazamiento de la proyección en el archivo. Por ejemplo, en la segunda línea aparece «00001000» (4096 en decimal), lo que indica que la primera página de esta región se corresponde con el segundo bloque del archivo (o sea, el *byte* 4096 del mismo).
- Los siguientes campos identifican de forma única al soporte de la región. En el caso de que sea una región con soporte, se especifica el dispositivo que contiene el archivo (en el ejemplo, «08:01») y su nodo-i (para el comando «cat, 65455»), así como el nombre absoluto del archivo. Si se trata de una región sin soporte, todos estos campos están a cero.

A partir de la información incluida en este ejemplo, se puede deducir a qué corresponde cada una de las nueve regiones presentes en el ejemplo de mapa de proceso:

- Código del programa. En este caso, el comando estándar «`cat`».
- Datos con valor inicial del programa, puesto que están vinculados con el archivo ejecutable.
- Datos sin valor inicial del programa, puesto que se trata de una región anónima que está contigua con la anterior.
- Código de la biblioteca «`ld`», encargada de realizar todo el tratamiento requerido por las bibliotecas dinámicas que use el programa.
- Datos con valor inicial de la biblioteca «`ld`».
- Código de la biblioteca dinámica «`libc`», que es la biblioteca estándar de C usada por la mayoría de los programas.
- Datos con valor inicial de la biblioteca dinámica «`libc`».
- Datos sin valor inicial de la biblioteca dinámica «`libc`».
- Pila del proceso.

Habiendo llegado a este punto, experimente libremente, investigando el mapa de memoria de otros procesos. Por ejemplo, si se cambia al directorio

```
cd /proc
```

puede ver que hay un directorio por cada PID. Experimente investigando qué es lo que contiene el mapa de memoria del proceso cuyo PID es «1», luego con el de su propio intérprete de comandos «`bash`». Puede dejar ejecutando en una consola el proceso «`top`» y desde otra ver el mapa de memoria. Tome notas.

3.5.2 Práctica con Windows

3.5.2.1 Memoria virtual en Windows

> M. E. Russinovich y D. A. Solomon 2004 afirman que, si bien cada proceso en Windows tiene su propio espacio de memoria privado, el código en modo núcleo del sistema operativo y los manejadores de dispositivos comparten un espacio de direcciones virtuales único. Cada página en la memoria virtual está rotulada con el modo de acceso en el que el procesador debe estar para leer y/o escribir la página. Las páginas en el espacio del sistema pueden accederse sólo desde el modo núcleo, mientras que todas las páginas en el espacio de direcciones del usuario son accesibles desde el modo usuario del procesador. Las páginas de sólo lectura, tales como las que contienen código ejecutable, no son escribibles desde ningún modo.

Si en Windows hacemos clic con el botón derecho del *mouse* sobre el icono «Mi PC» y seleccionamos «Propiedades», obtenemos la ventana «Propiedades de sistema»; si hacemos clic sobre la pestaña «Rendimiento», vemos que contiene un botón llamado «Memoria virtual».

Puede observar que normalmente Windows se encarga de administrar la configuración de la memoria virtual (comportamiento recomendado), pero usted puede especificar la configuración de memoria virtual y aún deshabilitarla, lo cual no se recomienda, salvo que usted sea un usuario con experiencia o administrador del sistema.

Por otra parte, el «Solucionador de problemas de memoria» dice:

> Si tiene demasiados documentos abiertos o se están ejecutando demasiados programas simultáneamente, puede que no tenga suficiente memoria libre para ejecutar otro programa.

Y más adelante agrega:

> ¿Tiene suficiente espacio libre en el disco duro para el archivo de paginación virtual? Windows usa espacio de disco duro en la forma de un archivo de paginación virtual, para simular memoria RAM.

Con toda esta información ahora responda:

- Manteniendo el tamaño de la memoria principal, una de las ventajas de introducir memoria virtual sería...
  - poder reducir el grado de multiprogramación.
  - poder aumentar el grado de multiprogramación.
  - poder mantener estable el grado de multiprogramación.
  - que el sistema operativo no tenga que gestionar la memoria.

3.5.2.2 La estructura de PE

`Trashbin` es una herramienta desarrollada por Roshan James para observar dentro de un archivo con estructura PE. Es similar a `dumpbin` de Microsoft, que se distribuye con Visual Studio Pro.

`Trashbin` le permite observar dentro de un archivo PE estándar, tal como lo hace `dumpbin`, pero no puede hacer ciertas cosas que sí puede hacerlas `dumpbin` pero, además de las funcionalidades de éste, puede mostrar información de metadatos en archivos PE administrados (es decir, `.net`). Se puede descargar `trashbin` en formato fuente y ejecutable desde el siguiente enlace `http://www.thinkingms.com/pensieve/homepage/work/trashbin/archives/trashbin.zip`

3.5.2.3 PhysMem

Con el fin de demostrar la capacidad de ver la memoria física y de ofrecer la oportunidad de navegar a través de la RAM de su computadora, Mark Russinovich escribió `PhysMem`. Es un programa de consola para Win32 que abrirá la sección de memoria física y volcará los contenidos de regiones (en hexadecimal y ASCII) que le solicite en una interfaz simple de línea de comando.

Al navegar por la memoria, algunos lugares de interés a los que les gustaría darles una mirada es en la dirección 0x1000, que es donde está ubicado NTLDR (*NT Loader* o Cargador de NT), y en el rango 0xF9000-0xFFFFF, que es donde está proyectada la ROM BIOS.

`PhysMem` usa la API nativa para abrir y proyectar `\Device\PhysicalMemory` porque ese nombre es inaccesible a través de la API Win32. También usa la API nativa (todas documentadas en el Windows NT DDK o *Driver Development Kit*) para proyectar vistas de la sección, a pesar de que esto se podría haber hecho usando Win32.

Puede descargar `PhysMem` desde: `http://download.sysinternals.com/Files/PhysMem.zip`

3.5.2.4 CacheSet

`CacheSet` es un *applet* que le permite manipular los parámetros del conjunto de trabajo de la caché de archivos del sistema. Además de darle la capacidad de controlar los tamaños mínimos y máximos del conjunto de trabajo, también le permite restablecer el conjunto de trabajo de la caché, forzándolo a crecer tanto como sea necesario desde un punto de partida mínimo.

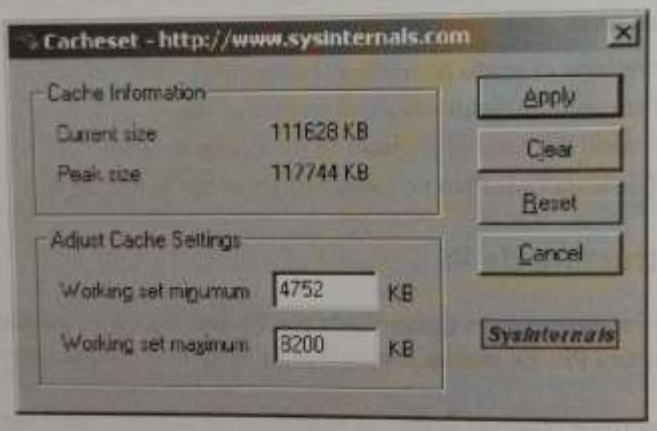


Fig. 3.15 – CacheSet.

gustaría darles una
Loader o Cargador
ROM BIOS.
alMemory porque
nativa (todas do-
vectar vistas de la

.com/Files/Ph

njunto de trabajo
olar los tamaños
r el conjunto de
punto de partida

Martín Silva

Planificación

4

4.1 Conceptos

En el escenario más simple que nos pudiéramos imaginar en una computadora tendríamos a un único proceso que, habiendo cargado en memoria, ejecuta hasta finalizar. Esto es lo que ocurría en las primeras computadoras y es lo que podemos observar en una consola de videojuegos. En sistemas posteriores un poco más complejos, de procesamiento por lotes pero uniprogramados, y que también debían ejecutar de comienzo a fin, la función del sistema operativo era la de protegerse de los programas de usuario. Asimismo, dichos sistemas debían ser capaces de darle inicio al próximo programa en el lote. Luego apareció la multiprogramación, cuyo propósito era el de solapar la ejecución en la CPU con el procesamiento de la entrada y salida, ya fuera administrando un número fijo o un número variable de tareas, como por ejemplo el IBM OS/MFT y el IBM OS/MVT, respectivamente. Estos sistemas contaban con un procesador dedicado a asignar un lote en tarjetas a la memoria y, de manera similar, asignar la impresora al proceso que requería efectuar la salida descargándolo de la memoria. Pero luego, con la aparición del tiempo compartido, fue necesario conmutar rápidamente entre procesos decidiendo no solo cuándo conmutar sino a qué proceso. El código del sistema operativo que se ha hecho cargo de esta tarea, como ya dijimos, es el **planificador**.

Sin embargo, para no imaginar la historia de las ciencias de la computación de manera totalmente lineal y siempre evolutiva, tengamos en cuenta que los sistemas operativos más populares de fines de la década de los 80 y comienzos de la de los 90 –es decir, DOS y Mac– tampoco tenían algoritmos de planificación de la CPU sofisticados. Poseían un único hilo de ejecución y corrían de a un proceso por vez hasta que el usuario les solicitaba ejecutar otro.

Tal como decíamos en el apartado 2.1.3, el planificador es el encargado de tomar la decisión de qué proceso ejecuta, habiendo varios procesos que están en condiciones de hacerlo, de la manera más justa y equitativa, y basándose en alguna política (si bien no es tan fácil

definir qué es «justo y equitativo», como en muchos órdenes de la vida). Como vemos, la función del planificador es muy importante porque tiene un gran efecto en la utilización de los recursos y en el rendimiento global del sistema. Es tan importante, que los libros acerca de los sistemas operativos le dedican un considerable espacio, incluso un capítulo entero, como en este caso, pero irónicamente el trozo de código que cumple esta función en el núcleo del sistema operativo es bastante corto.

4.1.1 Supuestos subyacentes

Los supuestos básicos por detrás de la mayoría de los algoritmos de planificación son:

- Hay un grupo de procesos ejecutables que compiten por la CPU.
- Los procesos son independientes y compiten por los recursos.
- El trabajo del planificador es el de distribuir el preciado recurso de la CPU a los diferentes procesos «de manera justa» (de acuerdo con cierta definición de justicia) y de una manera que optimice algún criterio de rendimiento.

En general, estos supuestos básicos han comenzado a romperse, como señala Rinard (1997). Primero que todo, las CPU realmente no son tan escasas; casi todas las computadoras actuales tienen varias (o tienen varios núcleos o *cores*) y muy pronto la gente podrá costearse computadoras con muchas CPU. Segundo, muchas aplicaciones están comenzando a estructurarse en la forma de muchos procesos cooperativos. Entonces, el enfoque del planificador como mediador entre entidades en competencia puede estar obsoleto. Pensemos por un momento que desde la década del sesenta hasta la fecha son muchos los aportes que se han efectuado en esta materia y estos supuestos básicos se han dejado traslucir en las publicaciones científicas. De hecho, las computadoras eran costosísimas, de manera que no aprovecharlas adecuadamente constituía un lujo injustificable. Además, cuando los usuarios debían pagar por utilizarlas, se esperaba obtener el mayor provecho por el dinero invertido. De los programas podemos decir que casi no tenían interacción con el usuario y había más determinismo y predictibilidad en su ejecución[1]. Comparemos este escenario con el que vemos actualmente, en el que las computadoras son muchísimo más potentes y baratas que aquellas; es habitual que tengamos más de una, que las tengamos encendidas sin usarlas y que los programas sean altamente interactivos, con poco determinismo y mucha cooperación y concurrencia en su ejecución.

4.1.2 Características

Los procesos e hilos no son todos iguales, pero es típico que ejecuten durante un cierto tiempo, luego realicen alguna operación de entrada o salida y luego ejecuten por un cierto tiempo más, y así hasta finalizar (si es que lo hacen, porque hay procesos en la computadora que corren todo el tiempo que esté encendida). Los procesos se caracterizan por tener **ráfagas** (*bursts*) de CPU intercaladas con ráfagas de entrada y salida, como se aprecia en la figura 4.1.

De los procesos que realizan muchos cálculos y muy poca entrada o salida, con ráfagas largas de CPU, diremos que son **orientados a la CPU** (*CPU bound*), mientras que de los

1. Queremos decir que si ejecutábamos varias veces el mismo programa con la misma entrada obteníamos la misma salida.

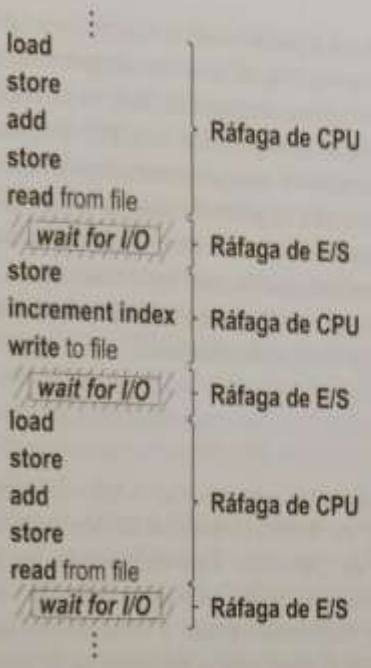


Fig. 4.1 – Ráfagas de CPU y de entrada y salida.

que realizan muchas operaciones de entrada y salida con ráfagas cortas de CPU diremos que son **orientados a la entrada y salida** (*I/O bound*). Entonces una de las típicas cosas que hará un planificador es permutar la CPU a otro proceso cuando uno esté efectuando entrada o salida, porque estas operaciones toman mucho tiempo (recordemos la tabla 3.2) y no queremos dejar a la CPU ociosa mientras espera que finalicen estas operaciones. Esto era tan cierto tanto en los años sesenta, en los que estas operaciones tardaban varios minutos, como en la actualidad, en las que generalmente son imperceptibles al cerebro humano. Un proceso orientado a la entrada y salida puede ser tal porque interactúa con un humano o no. Por ejemplo, el proceso de un editor de textos estará orientado a la entrada y salida. Aun operado por el humano con los dedos más veloces del mundo, el proceso tendrá que esperar mucho entre una pulsación de tecla y la otra. Es importante darle prioridad a los procesos que interactúan con humanos porque la falta de velocidad y respuesta en este tipo de aplicaciones es percibida inmediatamente por nosotros. En cambio, nos importa menos que demore un par de minutos más un programa orientado a la CPU, como podría ser la compilación de un programa o uno que secuencie el ADN.

Un proceso orientado a la entrada y salida que no interactúa con un humano puede ser uno que tenga muchos accesos al disco; por ejemplo, el motor de una base de datos. No hay forma de decirle al planificador qué actividad de entrada y salida interactúa con un humano y cuál no, es decir, el planificador no sabe si el proceso está bloqueado esperando una pulsación de teclado o bloqueado esperando datos desde el disco rígido.

Así lo explicamos desde el punto de vista de los procesos e hilos, pero también podríamos formarnos una imágen desde el punto de vista de la CPU, que atiende (ejecuta) trozos de código pertenecientes a distintos procesos o hilos así como a trozos de código del propio sistema

operativo, más específicamente del planificador. Como lo expresa Aas (2005), normalmente el planificador corre en su propio hilo, el cual es despertado por una interrupción del temporizador (*timer interrupt*). En otras circunstancias, es invocado a través de una llamada a sistema u otro hilo del núcleo que desea ceder la CPU. Por un cierto tiempo se le permitirá ejecutar a un cierto hilo, luego vendrá una conmutación al hilo del planificador, seguido por otra conmutación al hilo elegido por el planificador, y así continúa este ciclo.

Entonces, desde el punto de vista de los procesos, éstos corren como si tuvieran toda la CPU para ellos. El usuario, que interactúa con procesos, tiene la misma sensación. Desde el punto de vista de la CPU, ésta pareciera estar permanentemente atendiendo a una interminable cola de trozos de procesos y trozos del núcleo.

4.1.3 Contexto

Cuando el núcleo decide ejecutar un proceso o hilo diferente, ordena al hardware almacenar el **contexto** del mismo, es decir, el estado de los registros de la CPU, en el área de almacenamiento del contexto del proceso. Una vez que se ha llevado a cabo esta operación, se carga el contexto de otro proceso desde su propia área de almacenamiento, se limpian las cachés, se cambia el mapa de memoria virtual y se reanuda la ejecución del código de este proceso (o hilo) recién cargado. A este procedimiento se le llama cambio de contexto[2]. Esto agrega un gasto (*overhead*) extra a los procesos y cuantos más registros tenga la CPU, mayor será la sobrecarga. El tiempo requerido para realizar un cambio de contexto está determinado por cuántos registros de la CPU necesitan ser preservados y restablecidos. Esta conmutación depende fuertemente del hardware y realmente no es parte del planificador. El tiempo del cambio de contexto varía, por lo tanto, según el hardware; es decir, la velocidad de la memoria, el número de registros y si hay o no instrucciones especiales son variables que inciden en ese tiempo.

4.1.4 Desalojo

Antes de comenzar a analizar a los distintos algoritmos de planificación es importante que retomemos el tema del desalojo o requisa (*preemption*) introducido en la sección 2.2.3.4, y que significa que el sistema operativo mueve un proceso del estado «ejecutando» al estado «listo», sin que el proceso o hilo lo haya solicitado, expulsándolo para retormarlo con posterioridad. Sin desalojo, diremos que el sistema implementa la política de «ejecutar hasta finalizar».

Por el contrario, puede que el proceso ceda voluntariamente la CPU, bien porque ha finalizado o bien porque está esperando un evento externo tal como la transferencia de datos desde el disco, que se presione una tecla o que llegue un paquete de la red; en definitiva una operación de entrada o salida, como se muestra en la figura 4.1. Para implementar el desalojo, el sistema operativo necesita de una interrupción de reloj o equivalente.

El sistema operativo posee un «reloj de interrupción» o «temporizador de intervalos» para generar esta interrupción, en algún tiempo futuro específico o después de un transcurso

2. El cambio de contexto puede realizarlo tanto el software como el hardware. Los procesadores a partir del Intel 80386 cuentan con instrucciones específicas para ello.

de tiempo en el futuro; la CPU es entonces despachada hacia el siguiente proceso. Si el reloj interrumpe un proceso de usuario, esta interrupción provoca la ejecución de código del sistema operativo, el que decide cuál será el proceso que obtendrá la CPU.

El reloj de interrupción ayuda a garantizar tiempos de respuesta razonables a usuarios interactivos, ya que evita que el sistema se «cuelgue» a un solo proceso en un ciclo infinito y permite que los procesos respondan a «eventos dependientes del tiempo». Asimismo, los procesos que necesitan una ejecución periódica dependen del reloj de interrupción.

Al desalojo lo encontramos en todos los sistemas operativos modernos de propósito general y es necesario para garantizar la equidad.

Un proceso retiene el control de la CPU hasta que la libera voluntariamente, el reloj lo interrumpe o alguna otra interrupción atrae la atención de la CPU. Los sistemas operativos cuyo planificador no utiliza desalojo necesitan que cada proceso haga algo para ceder explícitamente el control de la CPU. Para mantener la ilusión de la concurrencia, este procedimiento se debería efectuar con frecuencia.

A la planificación sin desalojo se la suele llamar también «multitarea cooperativa» porque las tareas cooperan unas con otras para cederse el control de la CPU. Los sistemas operativos que no utilizan desalojo son mucho más simples de diseñar que los que sí lo utilizan. Una de las ventajas que tienen es que la latencia de las interrupciones es baja. Otra ventaja es que pueden utilizar código no reentrante sin peligro de ser dañado por otro proceso. Esto es así porque cada proceso corre hasta completar antes de que abandone la CPU. Sin embargo, a las funciones reentrantes no se les debería permitir el control de la CPU. Otra ventaja es la menor necesidad de resguardar datos compartidos mediante el uso de semáforos que protegen estas variables compartidas, ya que cada proceso posee la CPU sin la preocupación de ser desalojado. No obstante, bajo ciertas circunstancias es necesario el uso de semáforos, como en el caso que tengamos dispositivos de entrada o salida compartidos que pueden requerir el uso de semáforos de exclusión mutua, por ejemplo, si un proceso necesita acceso exclusivo a una impresora.

Por ello, si se utilizan algoritmos con desalojo deberemos considerar si el resto del sistema operativo está preparado para desplazar en cualquier momento a un proceso. Por ejemplo, si ese proceso en ese momento, a causa de una llamada al sistema (*system call*) está haciendo que se modifiquen estructuras del núcleo y se lo interrumpe, esas estructuras pueden quedar en un estado inconsistente. Por ello, sería razonable que el sistema operativo espere a que se termine la llamada al sistema para efectuar el desalojo.

No obstante, una interrupción puede desalojar un proceso ya que tiene prioridad sobre éste, si bien siempre se vuelve al proceso interrumpido. La desventaja más importante de un sistema sin desalojo es su falta de «sensibilidad», ya que si tuviéramos a un proceso de más prioridad listo para ejecutar podría tener que esperar bastante porque el proceso que está ejecutando en ese momento cederá la CPU sólo cuando deba hacerlo. Los núcleos de tiempo real son expropiativos a causa de estas demoras. Los núcleos no expropiativos no son determinísticos; nunca se sabe cuándo un proceso de mayor prioridad obtendrá la CPU.

En cambio, cuando deseamos un buen tiempo de respuesta usamos un planificador con desalojo. Sin embargo, aunque los sistemas con desalojo permiten darle el control a una tarea con mayor prioridad, son mucho más complicados de diseñar y no deberían hacer uso de funciones con código no reentrante a menos que se aseguren el uso exclusivo mediante

semáforos de exclusión mutua; si no, se corre el riesgo de corrupción de datos si el proceso usa la misma función no reentrante que utilizó el proceso desalojado.

Normalmente, cuando hablamos de una planificación con desalojo suponemos implícitamente que nos estamos refiriendo –por simplicidad– a procesos de usuario administrados por un núcleo de sistema operativo que no se desaloja a sí mismo, es decir, que desaloja al código de un proceso si está operando con el procesador en modo usuario, no en modo núcleo, lo cual implicaría estar desalojando código del núcleo mismo. Pero al decir que en los sistemas operativos modernos el núcleo mismo puede ser desalojado, queremos significar que desalojamos pequeños trozos o instrucciones que en ese instante se están ejecutando con el procesador en modo núcleo. Obviamente que no se lo desaloja durante la ejecución de una sección crítica o de un *spin lock*, pero se lo suele usar dentro del código de un manejador de dispositivo.

> Un *spinlock* es una traba sobre la cual el hilo espera dando vueltas en un bucle hasta que se libere.

Los núcleos expropiativos (*preemptive kernels*, en inglés) son necesarios con aplicaciones de tiempo real que deben cumplir con restricciones de tiempo muy estrictas. Por ejemplo, el tiempo necesario para completar una llamada `malloc()` depende de cuán fragmentado se encuentre el montículo (o *heap*), el tamaño del bloque solicitado y la carga del sistema; entonces, una llamada puede demorar unos 10 milisegundos en completarse mientras que otra puede tardar 50 milisegundos. Si una tarea debe terminarse dentro de los 40 milisegundos, podría no finalizar a tiempo si otro proceso llama a `malloc()` en el medio. Por el contrario, el planificador de un núcleo con desalojo podría expulsar la llamada a `malloc()` y habilitar a la tarea de tiempo real para que finalice a tiempo.

4.1.5 Colas

En ciencias de la computación, una cola es una lista lineal en la que se hacen todas las inserciones y supresiones (y normalmente todos los accesos) en un extremo, según define D. Knuth (1997a). El sistema operativo organiza colas para administrar las solicitudes a los recursos compartidos. Hay colas para la planificación del uso de la CPU y también colas para el uso de dispositivos (o *device queues*), que son utilizadas para ordenar los requerimientos que varios procesos pueden tener sobre un dispositivo específico.

Al ser creados, los procesos se colocan en la cola de trabajos (*task queue*), formada por los procesos que aún no residen en la memoria principal pero que están listos para su ejecución; es la forma que el núcleo tiene de diferir el procesamiento.

La residencia en memoria es imprescindible para la ejecución de un proceso; por ello, la cola de trabajos listos (*ready queue*) es aquélla formada por los procesos que están listos para la ejecución en memoria principal; son los procesos que compiten directamente por la CPU.

Los procesos, así como cambian dinámicamente de estado, también migran entre las distintas colas a medida que ejecutan. Un proceso que está en la cola de listos no puede estar en las colas de dispositivo, pues en este último caso estaría esperando a que se completara una operación de entrada o salida; por lo tanto, no puede estar compitiendo por la CPU.

Si bien hablamos de procesos por simplicidad, para poder representarnos mentalmente lo que ocurre, los elementos de cualquiera de estas colas no son los procesos en sí, sino sus PCB que están en memoria.

La cola de procesos listos (*ready queue*) se almacena como una lista enlazada donde la cabecera (*header*) de la cola de listos tiene punteros al primer y al último PCB de los procesos de la lista; cada PCB apunta al próximo en la lista.

4.1.6 Criterios

Existen distintos criterios que se deben tener en cuenta a la hora de seleccionar un algoritmo de planificación y debemos tener en claro bajo cuáles nos interesa fijar las pautas, dependiendo del objetivo que sigamos. Por ejemplo:

- Maximizar el uso de la CPU, para que se mantenga ocupada el mayor tiempo posible.
- Maximizar el rendimiento (*throughput*), es decir, el número de procesos que se ejecutan por unidad de tiempo.
- Minimizar el tiempo de retorno (*turnaround time*), que es el tiempo transcurrido desde que el proceso ingresa hasta que termina, sumando espera para entrar en memoria, en cola de procesos listos, ejecutándose en CPU y haciendo entrada o salida.
- Minimizar el tiempo de espera (*waiting time*), que es el tiempo que el proceso está en la cola de listos (pero no ejecutando).
- Minimizar el tiempo de respuesta (*response time*), que es el tiempo que transcurre desde que se presenta una solicitud hasta que se tiene respuesta. Es el tiempo que tarda en comenzar a responder, sin incluir el tiempo de la respuesta en sí.

Evidentemente, no podemos considerar los procesos interactivos de la misma manera en que consideramos los procesos por lotes (o poco interactivos). En los sistemas por lotes queremos buen rendimiento o tiempo de retorno; por otro lado, en los sistemas interactivos, si bien ambos tiempos son importantes, el de respuesta es primordial. De hecho, para algunos sistemas el rendimiento o el tiempo de respuesta no es ciertamente relevante, porque algunos procesos ejecutan todo el tiempo que la computadora esté encendida.

Otros criterios que deberíamos tener en cuenta son:

- La equidad, si consideramos que todos los procesos tienen que ser tratados de igual forma y a todos se les debe dar la oportunidad de ejecutarse, para que ninguno sufra inanición.
- Recursos equilibrados: la política de planificación que se elija debe mantener ocupados los recursos del sistema, favoreciendo a aquellos procesos que no abusan de los recursos asignados ni sobrecargan el sistema y, consecuentemente, bajando el rendimiento general.

En la práctica, estos objetivos a menudo tienen conflictos entre sí; por ejemplo, para incrementar la utilización del procesador debemos incrementar el número de procesos activos pero al hacerlo disminuimos el tiempo de respuesta, por lo que el planificador debe tomar una decisión de compromiso.

En términos de planificación, no hay una definición única de desempeño que satisfaga las necesidades de todos. No hablamos de lo mismo si nos referimos al desempeño según lo

percibe un usuario de computadora de escritorio o al desempeño con servidores ni al desempeño en computación de alto rendimiento (HPC, por sus siglas en inglés: *High Performance Computing*).

4.1.7 Notación de Landau

Está fuera del alcance de esta obra el tratamiento extenso de este tema, que pertenece a la Teoría de la Complejidad Computacional. Por ello, daremos algunas aproximaciones como para poder entender las clases de complejidad en las que recaen los algoritmos de planificación. Según Cormen y col. (2001), en análisis de algoritmos, una cota superior asintótica es una función que sirve de cota superior de otra función cuando el argumento tiende a infinito y se utiliza la notación de Landau $O(g(x))$, coloquialmente llamada notación O grande (*Big O Notation*, en inglés) para referirse a las funciones acotadas superiormente por la función $g(x)$. Esta notación nos da una idea acerca de cuánto tiempo usará un algoritmo; a saber, el tiempo para un algoritmo de complejidad $O(n)$ dependerá de la entrada, es decir, será lineal con n, mientras que uno de complejidad $O(n^2)$ será cuadrático a la entrada y otro de orden $O(1)$ será independiente de la entrada y operará de manera constante en el tiempo.

4.2 Tipos de planificación

Tal como vimos en la sección 2.1 y en la sección 2.1.3, el planificador está muy relacionado con los distintos estados de un proceso, como podemos apreciar en la figura 4.2.

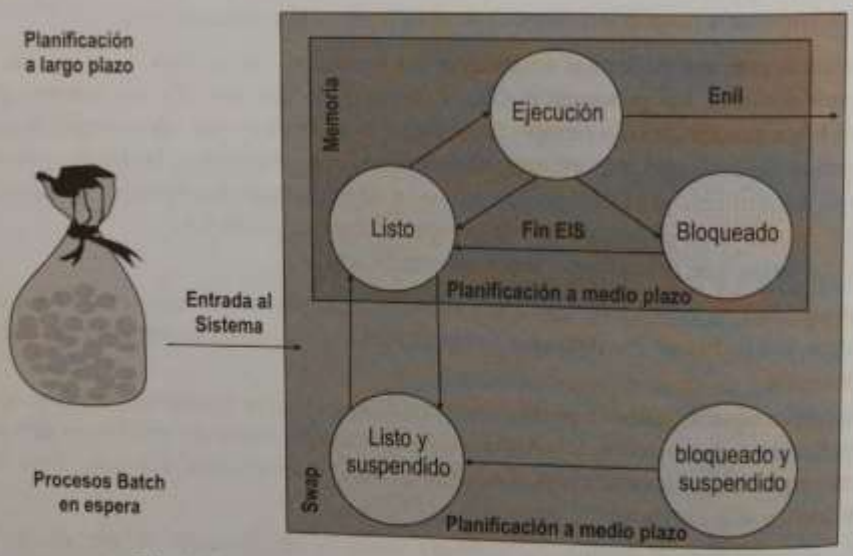


Fig. 4.2 – El planificador y las transiciones de estado.

El planificador a largo plazo determina qué procesos o hilos se admiten en la cola de listos y cuáles pueden competir por los recursos. Al efectuar la llamada al sistema para ejecutar un determinado programa, este planificador autoriza o no la admisión al conjunto de procesos

que se están ejecutando en este momento; es decir, aumenta el grado de multiprogramación y por lo tanto el grado de concurrencia que habrá en un momento determinado. A medida que tengamos más y más procesos en el sistema, será menor el tiempo de CPU que se le pueda asignar a cada uno de ellos. Si bien estaríamos manteniendo alto el grado de utilización de la CPU, degradaríamos los otros tiempos.

El planificador a mediano plazo se ocupa de la suspensión de procesos y es el que decide qué procesos en estado «bloqueado» dejan de estar en memoria principal para pasar a estar en memoria secundaria, intercambiando procesos entre una y otra, como ya dijimos en la sección 3.3.10. Al añadir o eliminar procesos de memoria principal, modifica el grado de multiprogramación.

El planificador a corto plazo es el que decide, de entre todos los procesos en estado «listo», cuál es el que ejecutará a continuación y durante cuánto tiempo luego del que está ejecutando en ese momento, cuando éste realice alguna operación de entrada o salida o sea interrumpido. Es decir que, de los tres, es el que más frecuentemente interviene y, por la función que cumple, se le suele llamar también planificador de la CPU. Por todo esto, el código que lo implementa debe ser altamente eficiente, para que la sobrecarga que impone en el rendimiento global no sea notable. Por ello, no es de extrañarse que, tanto en el pasado como en el presente, se lo haya codificado en lenguaje ensamblador en lugar de hacerlo en un lenguaje de alto nivel[3] como C o C++. El tiempo que este planificador le asigne al próximo proceso que se va a ejecutar –y para lograr el objetivo de alguna política de diseño previamente definida– es lo que calificará de cierta manera al sistema operativo, ya sea como de «propósito general», «tiempo compartido» o de «tiempo real».

Decíamos en la sección 2.1.3 que el activador (*dispatcher*) pone en ejecución el proceso seleccionado por el planificador de corto plazo, lo cual implica que es el encargado de efectuar el cambio de contexto –como lo describimos en la sección 2.1.4–, conmutar la CPU a modo usuario y «saltar» a la posición adecuada del proceso de usuario para que continúe ejecutando desde allí.

Madnick y Donovan 1974 caricaturizaron los roles de los planificadores –en aquella época se le llamaba planificador de trabajos al de largo plazo y planificador de procesos al de corto plazo– expresando que el planificador de largo plazo es como el coordinador de un concurso: decide quién entra a la competencia pero no quién la gana; el planificador de corto plazo, por otro lado, decide qué participante es el que gana, considerando que en un medio ambiente no multiprogramado, solo se permite la entrada de un concursante.

¿Cuándo se toman las decisiones de planificación? ¿Cuándo el planificador elige a qué proceso ejecutar? Hay una variedad de posibilidades:

- Cuando un proceso cambia de estado «ejecutando» a «esperando». Puede ser a causa de un pedido de entrada o salida, porque espera que termine un hijo o porque espera que se complete una operación de sincronización, tal como la adquisición de una traba.
- Cuando un proceso cambia de estado «ejecutando» a «listo»: por ejemplo al completar el manejo de una interrupción. Un caso común de manejo de una interrupción es el del temporizador (*timer interrupt*) en los sistemas interactivos. Si el planificador conmuta procesos en este caso, es porque ha desalojado un proceso que estaba ejecutando. Otro

3. Aun si todo el código del sistema operativo estuviera escrito en C, es posible «embutirle» instrucciones en lenguaje ensamblador.

tipo de manejo de interrupción es el del manejador (*handler*) de finalización de entrada y salida.

- Cuando los procesos cambian de estado «esperando» a «listo»: por ejemplo al completar la entrada o salida, o la adquisición de una traba.
- Cuando un proceso finaliza.

En el caso de que no hubiera hilos o procesos en estado «listo» para ejecutar, algunos sistemas operativos proporcionan uno que básicamente no hace nada pero que es elegido por el planificador porque siempre está en estado «listo». Este hilo no debería ser elegido nunca si hay otro en estado «listo» que merezca ejecutar. En lugar de no hacer nada se lo puede implementar para que efectúe ahorro de energía: por ejemplo efectuando un lazo o bucle que contenga la instrucción `HLT` del procesador, lo que provoca que se apaguen varios componentes internos hasta que llegue alguna interrupción. Este hilo podría estar implementado ya sea como un hilo del núcleo o bien como un proceso. En este caso, un usuario puede visualizarlo y contabilizarlo como tiempo ocioso de la CPU; así es como lo implementa Windows y lo llama «proceso inactivo del sistema» o *system idle thread*, en inglés.

4.3 Algoritmos de planificación

Habiendo expuesto los criterios o políticas que se deben tener en cuenta para la planificación, veremos algunos algoritmos que pueden implementarlas. Estos algoritmos tratan sobre la decisión de elegir a cuál de los procesos que están en la cola de listos se le asignará la CPU. Hay muchos algoritmos de planificación para muchos tipos diferentes de sistemas, pero solo examinaremos algunos de los más comunes.

4.3.1 FCFS (First-Come, First-Serve)

El algoritmo «primero en llegar, primero en ser atendido» es el más simple de entender y de implementar. Se le asigna la CPU al proceso que la requirió primero, a medida que los procesos o hilos van arribando. Se organiza mediante una cola FIFO (*First In First Out*), tal que, cuando un proceso requiere CPU, su PCB se coloca al final de la cola. Cuando se debe elegir un proceso para asignarle CPU se elige el proceso cuyo PCB está primero en la cola y se mantiene en la CPU hasta que haya finalizado; es decir, éste es un algoritmo sin expulsión. El tiempo promedio de espera puede ser largo y, como se podrá suponer, no es muy eficiente, ya que los procesos o hilos cortos se quedan atascados detrás de los largos.

4.3.1.1 El efecto convoy

Según Silberschatz y Galvin (1997), una situación indeseable que puede ocurrir cuando se mezcla un proceso orientado a la CPU con procesos orientados a la entrada y salida es el efecto convoy. Estos últimos ejecutan por poco tiempo y luego se bloquean esperando que se complete la operación de entrada o salida. El proceso orientado a la CPU en el estado «listo»

no tendrá que esperar mucho tiempo antes de pasar a «ejecutando». Pronto encontraremos al sistema con todos los procesos orientados a la entrada y salida bloqueados y ejecutando solamente el proceso orientado a la CPU. A medida que se vayan finalizando las operaciones de entrada o salida, la cola de «listos» se irá llenando con procesos orientados a la entrada y salida y su tiempo de espera aumentará a medida que la utilización del dispositivo de entrada y salida cae en picada. Una vez que el proceso orientado a la CPU solicita una operación de entrada o salida, la CPU podrá volver a ejecutar a los procesos orientados a la entrada y salida que estaban listos, pero si su ejecución se completa antes de que finalice la operación de entrada o salida del proceso orientado a la CPU, ésta quedará ociosa. El resultado es un bajo rendimiento, porque la CPU está atendiendo prácticamente un solo proceso. Este efecto podría evitarse atendiendo procesos más cortos primero –si de alguna manera pudiéramos prever cuáles son éstos– aumentando así la productividad.

4.3.2 SJF (Shortest-Job-First)

El algoritmo «primero el trabajo más corto» es otro algoritmo no expulsivo que intenta subsanar el efecto convoy eligiendo, de entre los procesos que están en la cola de listos, a aquél que tenga al tiempo de procesamiento más corto. En caso de empate, se puede usar el algoritmo FCFS, un algoritmo que también fue diseñado originalmente para los primeros ambientes de procesamiento por lotes. Para implementarlo es necesario conocer el tiempo de procesamiento, algo que en aquella época se podía estimar bastante bien, pero no con los procesos interactivos de la actualidad. El SJF ha sido modificado para su uso en planificación a corto plazo en ambientes interactivos y basa su decisión en el tiempo de ráfaga de procesador esperado, el cual se puede calcular como el simple promedio de los tiempos de ráfaga previos; por esto, el algoritmo debería llamarse «primero la siguiente ráfaga de CPU más corta», pero el nombre original permanece.

4.3.2.1 SRPT (Shortest-Remaining-Processing-Time)

El algoritmo SJF puede transformarse en expropiativo. Si se está ejecutando un proceso y llega a la cola de listos otro cuya siguiente ráfaga de CPU es más corta que lo que falta por ejecutar del proceso que actualmente tiene la CPU, el algoritmo SJF expropiativo desalojaría a este proceso que se está ejecutando para darle la CPU al que acaba de llegar a la cola de listos. A esta planificación, según Schrage y Miller (1965), podríamos llamarla «primero el que tenga el menor tiempo restante», es decir *Shortest Remaining Processing Time*, o SRPT, por sus siglas en inglés. Algunos autores la llaman SRT, es decir, *Shortest Remaining Time*.

Esta política de planificación es conocida desde hace tiempo por ser óptima para minimizar el tiempo medio de respuesta. A pesar de esto, se la utiliza poco en la práctica porque se piensa que la mejora en el desempeño de SRPT sobre otros algoritmos de planificación se deriva del hecho de que penaliza injustamente a los trabajos largos favoreciendo a los cortos pero Bansal y Harchol-Balter (2001) muestran que en realidad no es tan injusto, comparándolo con otros algoritmos conocidos como «justos». También se le objeta que el basarse en el supuesto conocimiento del futuro requerimiento de tiempo es una afirmación bastante difícil de sustentar, de algunos procesos se puede decir que tienen un comportamiento predecible pero de otros no.

4.3.3 Por prioridad

La idea básica es sencilla: se le asocia un valor a cada proceso que representará su prioridad y se le asigna la CPU al proceso de la cola de listos que tenga la mayor. Los valores asociados a la prioridad pueden estar en un rango fijo, de 0 a n, según cada sistema. También lo determina cada sistema si el número más bajo es la más alta o la más baja prioridad. La prioridad puede ser fija o variable, externa o interna. Si es fija, ese valor no varía en el ciclo de vida del proceso; si es variable, significa que ese valor puede cambiar dinámicamente de manera tal que haya factores, por ejemplo el tiempo que lleva esperando en colas, que puedan «ayudar» a que haya un mejor nivel de competencia elevando la prioridad de procesos postergados y evitando la inanición. A la técnica de elevar la prioridad de un proceso de acuerdo con el tiempo que lleva en el sistema se le llama envejecimiento o *aging*. Si a la prioridad se la define internamente, será determinada mediante cantidades medibles y en función del uso de los recursos (memoria, archivos abiertos, tiempos). Si es externa, será determinada por criterios externos al sistema operativo; por ejemplo, puede decidirse darle alta prioridad a un proceso de importancia, a consideración del operador. Otro criterio por adoptar puede ser el económico, es decir en los casos en los que se paga por el procesamiento.

También podemos pensar otros algoritmos en función del concepto de prioridad; es decir, si todos los procesos tienen la misma prioridad, deberemos procesarlos en orden según FCFS. Al algoritmo SJF también lo podemos pensar como un caso especial del algoritmo general de planificación por prioridades ya que su prioridad viene dada por la inversa de la próxima ráfaga (pronosticada) de CPU, es decir, cuanto más larga sea la próxima ráfaga, menor será su prioridad y viceversa.

Este algoritmo se puede implementar con desalojo o sin desalojo. Si mientras se está ejecutando un proceso llega a la cola de listos un proceso de mayor prioridad que el que se está ejecutando, puede existir la decisión de interrumpir al que se está ejecutando o no. Si no se lo desaloja, en lugar de darle la CPU, se lo pondrá primero en la cola de listos.

4.3.4 RR (Round-Robin)

En los sistemas de tiempo compartido se utiliza mucho esta planificación que le asigna a cada proceso de la cola de listos un intervalo de tiempo de CPU llamado «rodaja de tiempo» (*time slice* en inglés) o *quantum*; los procesos van tomando la CPU por turnos (de ahí su nombre, RR o *Round-Robin* en inglés) y por el tiempo que indique este *quantum*, al cumplirse éste, el proceso es desalojado mediante una interrupción y retornado a la cola de listos, pero puede ocurrir que el proceso deje la CPU porque finalizó o requirió una operación de entrada o salida, y en este caso el planificador selecciona otro proceso.

Como se ve, es una planificación equitativa y ordenada, además de simple de implementar, pero el tiempo promedio de espera puede ser largo. Además, depende fuertemente de la elección del *quantum*, que debe ser mucho mayor que el tiempo del cambio de contexto para tener un alto rendimiento de la CPU, pues si es excesivamente grande tiende a degenerar en el algoritmo FCFS. Es mejor para procesos cortos porque no se quedan atascados tras procesos largos pero tiene un rendimiento pobre si los procesos o hilos tienen la misma duración.

4.3.5 Colas multiniveles

4.3.5.1 MLQ

La desventaja principal de los algoritmos precedentes es que suponen que todos los procesos son de la misma naturaleza, sin tomar en cuenta cuán orientados a la CPU (o a la entrada y salida) están y si son interactivos o no. Esto provoca que estos algoritmos favorezcan a cierta clase de procesos.

Por ello la planificación mediante colas multiniveles, o MLQ por sus siglas en inglés (*Multi Level Queues*), intenta diferenciar entre distintos tipos de trabajo. Con ese fin divide la cola de procesos listos en varias colas como vemos en la figura 4.3: una por cada tipo de trabajo, de modo que podamos asignar los procesos permanentemente a alguna de ellas para gestionarlos de la manera que realmente necesitan. El criterio de asignación está en función de alguna propiedad del proceso, por ejemplo el tamaño de memoria, la prioridad o el tipo de proceso. Entonces, por ejemplo, los procesos interactivos (los *foreground* de UNIX) estarán en una cola distinta de los procesos no interactivos (*background*) porque necesitan distintos tiempos de respuesta. Los interactivos podrían estar gestionados por un algoritmo RR mientras que los otros podrían depender de un algoritmo FCFS. Debe haber un criterio de planificación entre las colas, que puede ser de prioridad fija con desalojo o sin desalojo o, también puede ser de prioridad variable con desalojo o sin desalojo. El que suele implementarse es el de prioridad fija con desalojo, que consiste en que no se puede ejecutar un proceso si hay algún otro en una cola más prioritaria. Este algoritmo establece una competencia entre las colas y entre los procesos del mismo tipo entre sí.

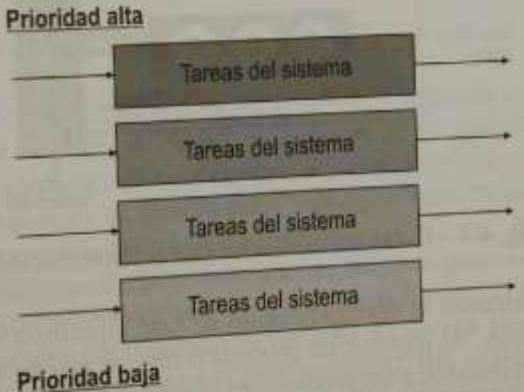


Fig. 4.3 – Colas multiniveles.

4.3.5.2 MLFQ

El algoritmo de colas multinivel presenta baja carga de planificación pero es poco flexible porque los procesos quedan asignados de forma permanente a una cola. Sin embargo, mediante un algoritmo que permita que un proceso se pueda mover de una cola a otra dependiendo de su comportamiento durante su ejecución obtendremos un algoritmo de colas multinivel

con realimentación, o MLFQ por sus siglas en inglés (*Multi Level Feedback Queues*). Estas se implementaron originalmente en el sistema CTSS, tal como lo describe Corbató, Merwin-Daggett y Daley (1962) y como lo analizó Scherr (1965). El objetivo del algoritmo del CTSS era dar buena respuesta al usuario interactivo a costa del usuario no interactivo que realizaba extensos cálculos intensivos. El algoritmo cumplía sus objetivos estableciendo ocho colas de procesos listos (del 0 al 7). Los usuarios interactivos tendían hacia las colas con números bajos y los usuarios con procesos orientados a la CPU, hacia las colas con números altos. El planificador corría primero todos los procesos en las colas con números bajos (favoreciendo a los interactivos) pero, cuando corrían, los procesos en las colas con números altos recibían un *quantum* grande de tiempo, con lo que reducían el alto tiempo de cambio de contexto para los trabajos grandes orientados a la CPU, según afirman Madnick y Donovan (1974).

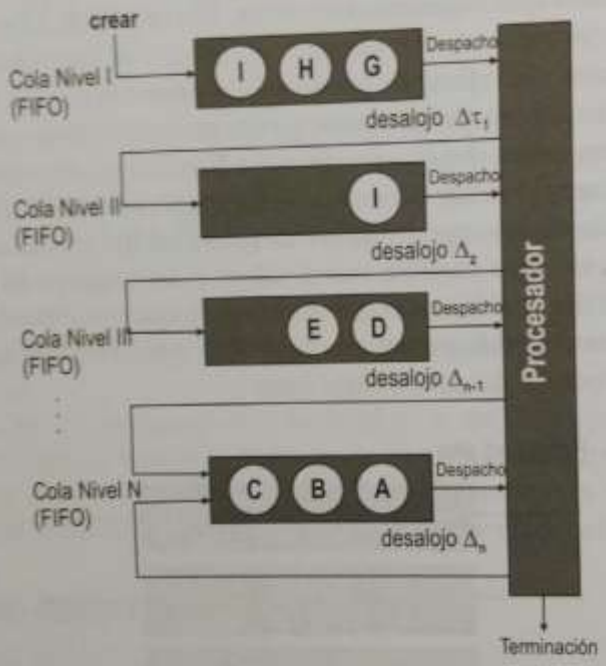


Fig. 4.4 – Colas multi-niveles con realimentación.

En la figura 4.4 ilustramos el funcionamiento de este algoritmo que implementa dos o más colas de planificación. Al ingresar un proceso o hilo se inserta en la cola de más alto nivel. Cuando son seleccionados, a los procesos en la cola se les asigna una rodaja de tiempo relativamente pequeña. Al expirar su rodaja de tiempo, al proceso se lo desplaza a la cola siguiente de menor nivel. Las rodajas de tiempo asociadas a las colas crecen a medida que decrece el nivel de la cola. Si un proceso se bloquea antes de utilizar toda su rodaja de tiempo, es movido a la próxima cola de nivel superior o bien a la cola de más alto nivel.

El proceso seleccionado por el algoritmo es el próximo proceso que está en la cola no vacía de mayor nivel. Típicamente, la selección dentro de la cola sigue el algoritmo FCFS. Si un proceso entra a una cola de alto nivel mientras un proceso de una cola de menor nivel está ejecutándose, puede ser desalojado este último.

Los procesos orientados a la entrada y salida permanecen en las colas superiores, donde reciben mayor prioridad sobre los procesos que están más orientados a la CPU en las colas de menor nivel. No obstante, a los procesos más orientados a la CPU se les da mayores rodajas de tiempo. Tal como ocurre con SJF y SRTF, es posible que un proceso orientado a la CPU sufra de inanición. Por eso, un posible remedio para esta situación es elevar un proceso a la siguiente cola de mayor nivel si ha permanecido ocioso en la cola por una cierta cantidad de tiempo.

El número de variables en un sistema con colas multiniveles con realimentación lo hace tanto flexible como complejo.

En general, un planificador MLFQ estará definido por los parámetros siguientes:

- El número de colas.
- El algoritmo de planificación de cada cola.
- El algoritmo de planificación entre las distintas colas.
- El método usado para determinar cuándo pasar un proceso a una cola de prioridad más alta.
- El método usado para determinar cuándo pasar un proceso a una cola de prioridad más baja.
- El método usado para determinar en qué cola se introducirá un proceso cuando haya que darle servicio.

Esto lo convierte en el algoritmo de planificación de la CPU más general, según Silberschatz y Galvin (1997), ya que se lo puede configurar para adaptarlo a cualquier sistema específico que se esté diseñando, pero también es el más complejo.

4.4 Planificación en POSIX

La norma POSIX, especificada por la IEEE (1996) y expuesta claramente en Carretero Pérez (2001), establece una serie de políticas de planificación, aplicables a procesos e hilos, que debe implementar cualquier sistema operativo que ofrezca esta interfaz. Cada proceso o hilo lleva asociada una política de planificación y una prioridad. Estos parámetros se pueden modificar mediante los servicios correspondientes.

Cada política de planificación lleva asociado un rango de prioridades. POSIX especifica que cada implementación debería ofrecer un rango de, al menos, 32 niveles de prioridad. El planificador elegirá siempre para ejecutar el proceso o hilo con la prioridad más alta. Las políticas de planificación disponibles en POSIX son FIFO, RR y OTHER.

El algoritmo que la norma llama FIFO es el FCFS, pero permite el desalojo; es decir que los procesos que se planifican según esta política se introducen al final de la cola de su prioridad asociada y dejan la CPU al ejecutar una llamada bloqueante o cuando aparezca en el sistema un proceso con mayor prioridad, en cuyo caso el proceso expulsado pasa a ser el primero de la cola asociada a su prioridad, pero si el proceso estuvo bloqueado y pasa a listo para ejecutar, se introduce al final de la cola asociada a su prioridad.

Un proceso puede cambiar su prioridad o su política de planificación utilizando los servicios adecuados, y al hacerlo se realiza una replanificación. Si como resultado de ésta el proceso resulta expulsado, éste se introduce al final de la cola de procesos de su prioridad.

Si el proceso o hilo sigue la política RR dispondrá de una rodaja de tiempo para ejecutar, luego de la cual será desalojado y puesto al final de la cola de procesos de su prioridad. Si es expulsado por otro de mayor prioridad, se introduce al principio de su cola pero sin restaurar su rodaja de tiempo. De esta forma, cuando el proceso continúe la ejecución, lo hará hasta que consuma el resto de la rodaja de su tiempo.

La política OTHER hace referencia a una cuyo comportamiento dependerá de cada implementación. Observe que la política de planificación se realiza por proceso y, por tanto, las tres políticas conviven en el planificador.

4.5 Planificación en Linux

El planificador de Linux ha estado evolucionando e innovando permanentemente a lo largo de su historia, y lo seguirá haciendo. Es uno de esos desafíos en los que, una vez alcanzada la meta, los desarrolladores evalúan las ventajas y desventajas de lo logrado y vuelven a lanzarse al próximo objetivo. Los primeros planificadores de Linux usaron diseños mínimos, obviamente no enfocados en arquitecturas masivas con muchos procesadores ni hiper-hilado; los últimos, en cambio, son muy sofisticados. Por lo tanto, es muy importante que tengamos en cuenta el número de versión del núcleo al referirnos a este tema.

El planificador de Linux es un estudio interesante sobre presiones en competencia ya que, por un lado, están los modelos de uso en los cuales se aplica Linux y, por otro, los avances tecnológicos.

A pesar de que Linux originalmente fue desarrollado como un experimento de sistema operativo de escritorio, lo encontramos ahora en servidores, teléfonos, pequeños dispositivos integrados, *mainframes* y supercomputadoras. No es de extrañar que la carga de planificación difiera en esta amplia variedad de dominios.

Del otro lado están los avances tecnológicos hechos en las plataformas, incluyendo arquitecturas –multiprocesamiento, multi-hilos simétricos, acceso no uniforme a la memoria o NUMA– y virtualización. También está incorporado aquí el balance entre la interactividad (sensibilidad del usuario) y la equitatividad en general. Desde esta perspectiva se aprecia qué difícil puede ser el problema de la planificación en Linux.

Según decíamos en el apartado 2.1.1, Linux –desde su primera versión y hasta la actualidad– almacena toda la información referida al proceso, incluyendo su planificación, en su PCB que está implementada como una estructura de datos en lenguaje C llamada `task_struct`.

4.5.1 Primeras versiones

4.5.1.1 Version 0.96

De acuerdo con las palabras del mismo Torvalds (1992), en una respuesta publicada en Usenet, el planificador original de Linux (la versión 0.96, al momento de contestar, ya que la versión 1.0 data del 14 de marzo de 1994) era bastante simple, como cabe suponer. Daba una buena respuesta a las tareas con muchas operaciones de entrada y salida pero sin ser injusto con las orientadas a la CPU. Era un planificador basado en prioridades dinámicas que utilizaba rodajas de tiempo y desalojo, pero no utilizaba colas; el proceso se elegía a partir de

un grupo de procesos listos, tomando al que tuviera el valor más alto de calificación y estaba en condiciones de ejecutar. Para ajustar la prioridad utilizaba un sistema, que podríamos llamar de «créditos», que el proceso iba consumiendo hasta dejarlo en cero. En el caso de que ningún proceso tuviera ya créditos para ejecutar, se efectuaba una renovación de éstos dándole a la variable contadora de créditos un valor con base en su prioridad. Esta renovación de créditos se le hacía a todos los procesos del sistema: no solo a los listos para ejecutar, sino también a los que estaban «durmiendo», esperando la finalización de una operación de entrada o salida; renovación que les daba más créditos para consumir cuando «despertaban».

Esta política se implementaba mediante distintas variables. En la estructura `task_struct` estaba la variable `counter` contadora de créditos; la variable `priority` que contenía la prioridad, y la renovación se efectuaba mediante la fórmula:

$$\texttt{counter} = \texttt{priority} + \frac{\texttt{counter}}{2}$$

Al núcleo Linux le llegaba una interrupción de reloj cada `jiffy` (100 Hz). El *jiffy* es un término coloquial para indicar un lapso de tiempo muy corto, lo que en castellano diríamos un periquete. En ese instante se decrementaba la variable contadora en el PCB del proceso (es decir, a `current->counter` de la estructura `task_struct`). Si esta variable llegaba a cero y el proceso no estaba ejecutando código con el procesador en modo núcleo, se volvía a planificar invocando a `scheduler()`. Esta era una forma de implementar rodajas pero sin llamarlas así. También se volvía a planificar cuando un proceso finalizaba su operación de entrada o salida y el valor de su contador era mayor que el del proceso que se estaba ejecutando. Se habilitaba una bandera que indicaba que se debía forzar la planificación. En ambas situaciones había un desalojo implícito.

Este esquema tan sencillo funcionaba muy bien y era muy adecuado para los procesadores 80386 de aquella época, pero obviamente hubiera estado muy restringido para operar en sistemas con más de un procesador o con múltiples núcleos. La capacidad para operar con SMP fue agregada en la serie 2.0; no obstante, no todas las distribuciones la habilitaban.

4.5.1.2 Versión 1.2

A partir de la serie 1.2 se incorporó una cola circular para la administración de las tareas listas para ejecutar, dejando atrás al conjunto sin órden de las versiones anteriores. El planificador operaba con una política de planificación por turno circular, además de las prioridades. Este planificador era eficiente para agregar y quitar procesos y se valía de un candado para proteger la estructura.

4.5.1.3 Version 2.0.33

De acuerdo con Rusling (1998), quien analizó la versión 2.0.33 de diciembre de 1997, en cada proceso se almacenaba su propia política de planificación dentro de su PCB, en su variable `policy`. Note entonces el cambio en el enfoque tradicional de planificación, en el que el algoritmo estaba definido en el núcleo y los procesos eran planificados pasivamente por este algoritmo. La política de planificación podía ser normal o de «tiempo real». Aclaremos

que Linux no es un sistema operativo de tiempo real; el término se refiere a tiempo real blando.

En la variable `priority` del PCB se guardaba –como siempre– la prioridad que el planificador le daba al proceso. Era también la cantidad de tiempo (en `jiffies`) durante la cual este proceso ejecutaba. Los procesos de tiempo real tenían una prioridad más alta que los otros procesos y si había un proceso de tiempo real listo para ejecutar, siempre ejecutaba primero que los otros.

El campo `rt_priority` del PCB le permitía al planificador dar a cada proceso de tiempo real una prioridad relativa. Las prioridades de los procesos –tanto los normales como los de tiempo real– podían ser alteradas mediante llamadas a sistema.

En la variable `counter` estaba la cantidad de tiempo (en `jiffies`) durante la cual le estaba permitido ejecutar a este proceso. Cuando el proceso comenzaba a ejecutar por primera vez se le asignaba el valor de `priority` y se lo iba decrementando con cada tick de reloj.

El planificador se ejecutaba desde distintos lugares en el núcleo: por ejemplo, luego de colocar el proceso en una cola de espera; también se lo podía ejecutar al finalizar una llamada a sistema, justo antes de que el proceso saliera de modo núcleo a modo usuario. Una de las razones por las que podía pasar esto era porque el temporizador del sistema acababa de poner a cero el contador `counter`.

4.5.1.4 Versión 2.2

De acuerdo con Bovet y Cesati (2000), la serie 2.2 de Linux introdujo el concepto de «clases» de planificación, permitiendo políticas de planificación para tareas de tiempo real, tareas no desalojables y tareas que no eran de tiempo real. El planificador de la serie 2.2 también incluyó soporte para multiprocesamiento simétrico (SMP).

4.5.1.5 Versión 2.4

Según lo expresa M. T. Jones (2009), el planificador de la serie 2.4 dividía el tiempo en períodos (*epochs*, en inglés). Dentro de un período las tareas podían ejecutar hasta agotar su rodaja de tiempo. Si una tarea no utilizaba toda su rodaja de tiempo, el planificador agregaba la mitad del tiempo restante no consumido al siguiente período para permitirle ejecutar por más tiempo la próxima vez. El planificador iteraba sobre las tareas aplicando una métrica o función de bondad (*goodness function*) para determinar qué tarea ejecutar a continuación.

Era bastante simple pero operaba en tiempo $O(n)$ ya que iteraba sobre todas las tareas durante un evento de planificación. Si el número de procesos era grande, al planificador podía demandarle una notable cantidad de tiempo de procesador para el cálculo; esto explica que su tiempo de ejecución fuera de complejidad $O(n)$, siendo n el número de procesos listos. Era también relativamente ineficiente porque carecía de prestaciones para aprovechar arquitecturas nuevas de hardware tales como procesadores multi-núcleo o multiprocesadores (SMP), como podemos apreciar en la figura 4.5.

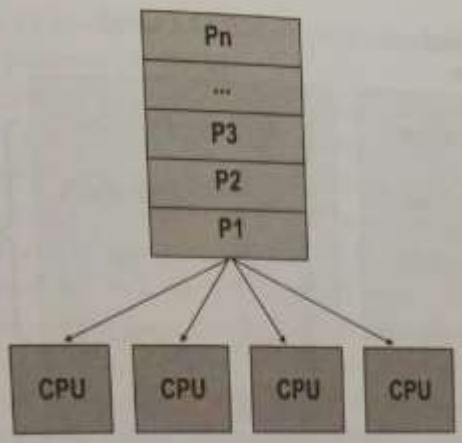


Fig. 4.5 – Lista de tareas en la serie 2.4.

4.5.1.6 Versión 2.6.8.1

Durante el período de desarrollo de la serie 2.5, uno de los cambios más significativos fue el de cambiar el planificador de la serie 2.4. El gran desafío era el de poder bajar el tiempo de cálculo, que se incrementaba de manera lineal con el número de tareas por procesar, a otro algoritmo que no dependiera de este número, es decir, que el algoritmo garantizara completar su tarea en una cierta cantidad de tiempo sin importar el número de tareas que se iban a procesar; dicho en términos más formales, pasarlo de complejidad $O(n)$ a complejidad $O(1)$. Por eso se lo conoció por este nombre, y su autor fue Ingo Molnár.

Uno de los problemas que dispararon la necesidad de un planificador nuevo, según afirma M. T. Jones (2006), fue el uso de máquinas virtuales Java™ (JVMs). El modelo de programación de Java usa muchos hilos de ejecución, lo cual sobrecargaba mucho la planificación con un algoritmo $O(n)$.

La base fundacional sobre la cual se apoya el algoritmo la dan dos estructuras de datos: una llamada «cola de ejecución» (o *runqueue* en inglés) y los arreglos de prioridad (*priority arrays*), como podemos apreciar en la figura 4.6.

La cola de ejecución sigue con atención a todas las tareas ejecutables asignadas a una CPU en particular; es decir, existe una cola de ejecución por cada CPU, a diferencia del algoritmo previo. Cada cola de ejecución contiene dos arreglos de prioridad. Cada uno es un arreglo de listas enlazadas, una para cada nivel de prioridad, y hay 140 niveles de prioridad. Estos elementos que se aprecian en la figura son atendidos por orden FIFO. Las tareas que son planificadas para ejecutar se agregan al final de un arreglo (a la lista de su nivel de prioridad), comenzando por el denominado «activo».

El trabajo del planificador consiste en elegir para ejecutar a la tarea que esté primera en este arreglo de prioridad activo, y a medida que se les van acabando sus rodajas de tiempo se las va moviendo al otro arreglo de prioridad, el «expirado». Las primeras 100 posiciones en la lista de la cola de ejecución están reservadas para las tareas de «tiempo real» y las últimas 40 para las de menor prioridad, llamadas «tareas de usuario». Durante el movimiento de una lista a otra se le calcula a la tarea una nueva rodaja de tiempo, así como su prioridad. Cuando no hay más tareas ejecutables en el arreglo de prioridad activo simplemente se intercambian

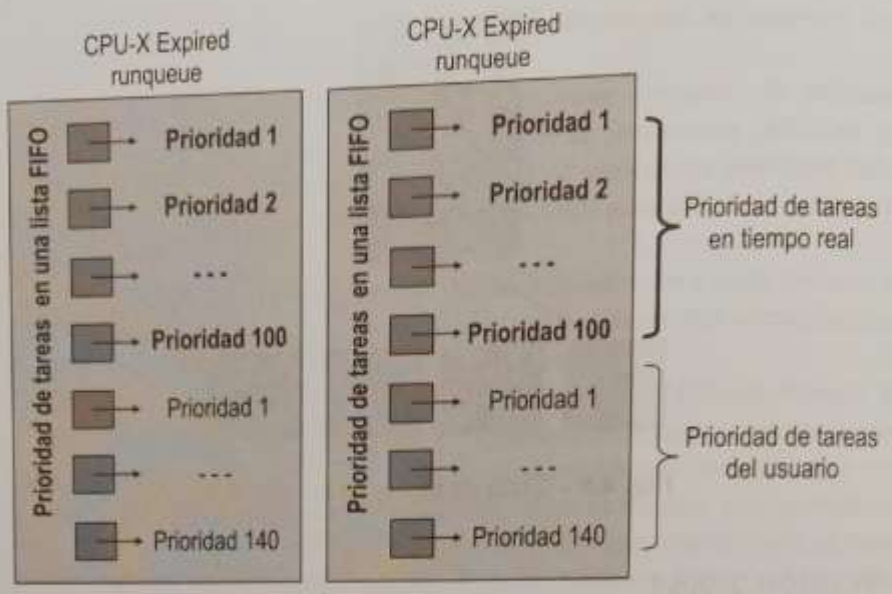


Fig. 4.6 – Estructura de cola de ejecución.

con el arreglo de prioridad expirado (lo que implica actualizar dos punteros solamente), y así el expirado pasa a ser el activo.

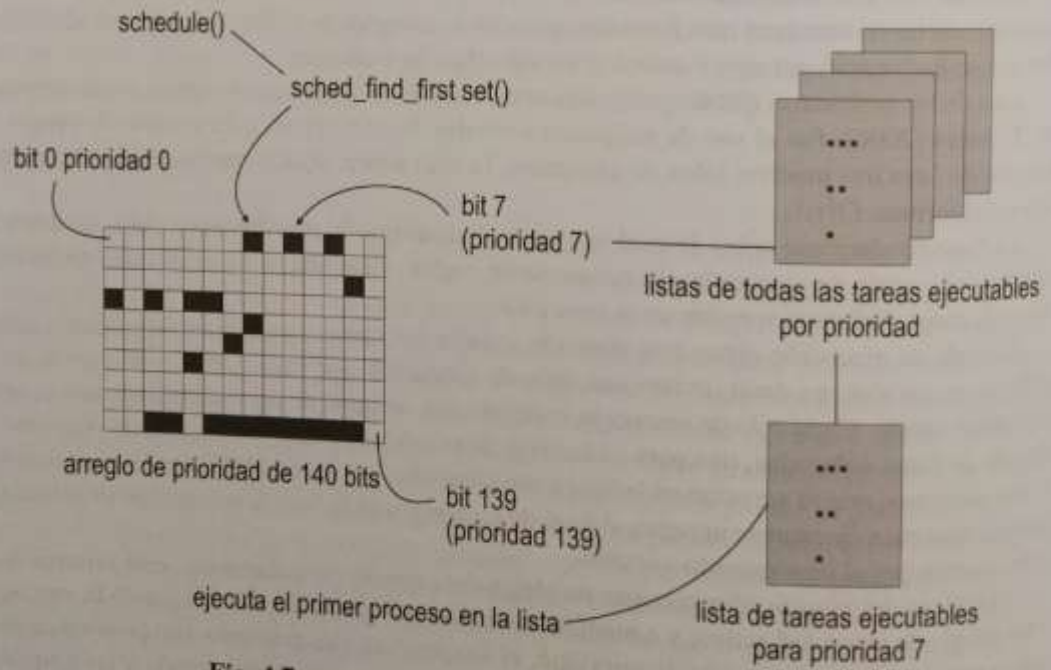


Fig. 4.7 – Mapa de bits de las tareas y sus prioridades.

Love (2010) ilustra cómo se utiliza un mapa de bits para elegir la próxima tarea por ejecutar, como el que se aprecia en la figura 4.7. En muchas arquitecturas se puede utilizar

la instrucción de procesador `find-first-bit-set` para encontrar el mayor bit de prioridad (habilitado) en una de las 5 palabras de 32 bits.

El tiempo que toma encontrar una tarea para ejecutar no depende del número de tareas activas sino del número de prioridades. Dado que encontrar el primer bit en un mapa de bits de longitud finita y encontrar el primer elemento en una lista son operaciones con un límite superior finito en cuanto a lo que puede tomar la operación, esta parte del algoritmo de planificación se desarrolla en un tiempo constante O(1), porque el tiempo que le demanda planificar es fijo y determinístico, sin tomar en cuenta el número de tareas activas.

No obstante, este mecanismo podía llevar a demoras inaceptablemente largas en la atención de las tareas porque las expiradas tenían que esperar (una cantidad de tiempo desconocida) a que las tareas activas consumieran su rodaja de tiempo. Cuando el número de tareas en el sistema se incrementaba, el peligro de inanición se incrementaba también. Para poder garantizar buena interactividad en el sistema, el planificador O(1) marcaba algunas tareas como interactivas y las reinsertaba en la cola activa (pero para prevenir la inanición esto solo se hacía si las tareas en el arreglo expirado habían ejecutado recientemente).

El planificador recompensaba a las tareas orientadas a la entrada y salida y castigaba a las orientadas a la CPU, afirma Aas (2005). Esto lo llevaba a cabo mediante métricas de interactividad con numerosas heurísticas para determinar si las tareas estaban orientadas a la entrada y salida o a la CPU, calculando cuánto tiempo pasaban las tareas en estado «durmiendo» y cuánto ejecutando; es decir, era un «estimador de interactividad».

Esta gran masa de código necesaria para calcular las heurísticas era difícil de manejar, se volvió pesada en el núcleo y, para los puristas, carecía de sustancia algorítmica.

Para solucionar este inconveniente, Con Kolivas (2007) desarrolló el algoritmo RSDL (*Rotating Staircase Deadline Scheduler*) que eliminó el código de estimación de interactividad. Está basado en el concepto de justicia (o *fairness*, en inglés) porque a las tareas se las trata con equidad dándoles la misma rodaja de tiempo. El planificador no se ocupa ni se interesa en determinar si la tarea está orientada a la entrada y salida o a la CPU. Como funcionaba bien en la serie de desarrollo 2.5, estaba por ser incorporado a la serie estable 2.6. Fue entonces cuando Ingo Molnár, el autor del algoritmo O(1), desarrolló el Planificador Completamente Justo (CFS o *Completely Fair Scheduler*, en inglés) tomando los elementos de diseño básico de la planificación justa del RSDL. No obstante, el mérito por probar que se puede lograr planificación justa sin tener conflicto con los objetivos de latencia de la interactividad pertenece a Con Kolivas, según lo reconoció el mismo Molnár (2007).

4.5.2 Versiones recientes

4.5.2.1 Versión 2.6.21

Como ya sabemos, todos los sistemas operativos modernos dividen los ciclos de CPU en forma de quantum de tiempo entre varios procesos o hilos (ambos llamados «tareas» en Linux) de acuerdo con sus políticas y prioridades. Esto es posible porque el temporizador del sistema periódicamente genera interrupciones y de esta forma permite que el planificador haga su trabajo de elegir a la próxima tarea para su ejecución. La frecuencia del temporizador está fijamente codificada (*hard-coded*) y se puede cambiar durante la fase de configuración del código fuente antes de compilarlo. Para los núcleos de la serie 2.4 esta frecuencia estaba

establecida en 100 Hz. En el caso de los núcleos de la serie 2.6, el usuario podía elegir entre 100, 250, 300 o 1000 Hz. Las frecuencias más altas implican atender más rápidamente a las interrupciones y una mejor interactividad del sistema a expensas de una mayor sobrecarga en el sistema, es decir, cuando no hay tareas en la cola de ejecución, atender miles de interrupciones del temporizador por segundo aparece como una pérdida de ciclos de CPU y de energía. En el núcleo 2.6.21 este problema fue solucionado implementando las así llamadas «tics dinámicas» (*dynamics tics*). Si el sistema estaba ocioso, algunas interrupciones de reloj se ignoran (se saltan) y se puede tener tan solo seis interrupciones por segundo. En tal estado, el sistema solamente responde a otras interrupciones de hardware.

La medición del tiempo es una de las partes más importantes del subsistema planificador y en la implementación del CFS se ha mejorado al agregar un temporizador de alta resolución. Esta clase de temporizador había sido agregada en la versión 2.6.16 del núcleo y ofreció una resolución de nanosegundos en lugar de una de milisegundos, lo que le permitió al planificador realizar su labor con una granularidad mucho más fina. En el planificador O(1) a cada tarea se le daba su porción de tiempo de CPU de longitud variable, la cual dependía de la prioridad de la tarea y su interactividad. Este quantum podía ser usado en su totalidad a menos que apareciera en el sistema una tarea nueva con una prioridad más alta o la tarea cediera voluntariamente la CPU a la espera de que finalizaran operaciones de entrada o salida ya iniciadas.

4.5.2.2 Versión 2.6.23

Molnár (2007) diseñó el planificador CFS (*Completely Fair Scheduler* o Planificador Completamente Justo), utilizado a partir de la versión 2.6.23, teniendo en mente tanto al mercado de las computadoras de escritorio como al de hardware para servidores. El cumplir con estas dos demandas tan distintas hizo de este planificador un caso de estudio muy interesante.

Según él, el 80% del diseño del planificador CFS puede resumirse en una sola oración: «CFS modela básicamente a una CPU multitarea ideal y precisa sobre hardware real». Una CPU tal (obviamente inexistente) ofrecería el 100% de su potencia (no de su tiempo) a una tarea y la dividiría en forma equitativa si tuviera más de una: 50% para cada una si tuviera dos, 25% para cada una si tuviera cuatro, y así. Si contáramos con esta CPU ideal, capaz de ejecutar de forma paralela n tareas, entonces cada una de ellas tendría la enésima parte de la potencia de la CPU, como vemos en la figura 4.8.

Pero como en la realidad una CPU solo puede ejecutar una a la vez, la situación que tenemos es la que vemos en la figura 4.9: mientras una tarea utiliza la CPU, las otras deben esperar.

Por ello, el planificador CFS emplea la noción de «tiempo de ejecución virtual» (*virtual runtime*) para poder imitar un procesador así en software y dar a cada tarea su porción justa de potencia de procesador.

Cuando se crea una tarea, se registra la hora del momento con precisión de nanosegundos y, a medida que la tarea espera su turno para ingresar a la CPU, el planificador va contabilizando la cantidad de tiempo que transcurre pero dividido por la cantidad de tareas en el sistema (y un pequeño ajuste por la prioridad), es decir, el tiempo «físico» pero ponderado, o inversamente proporcional a la cantidad de tareas en el sistema.

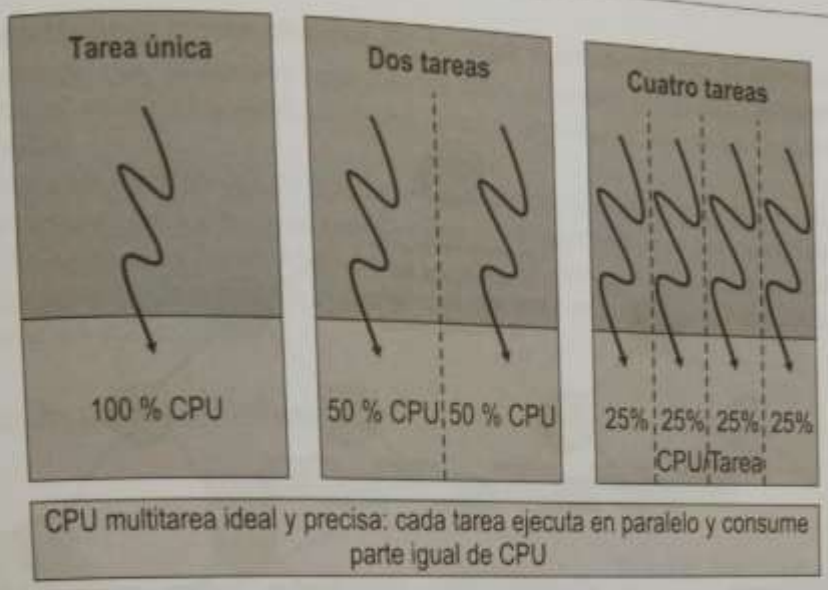


Fig. 4.8 – CPU multi-tarea, ideal y precisa.

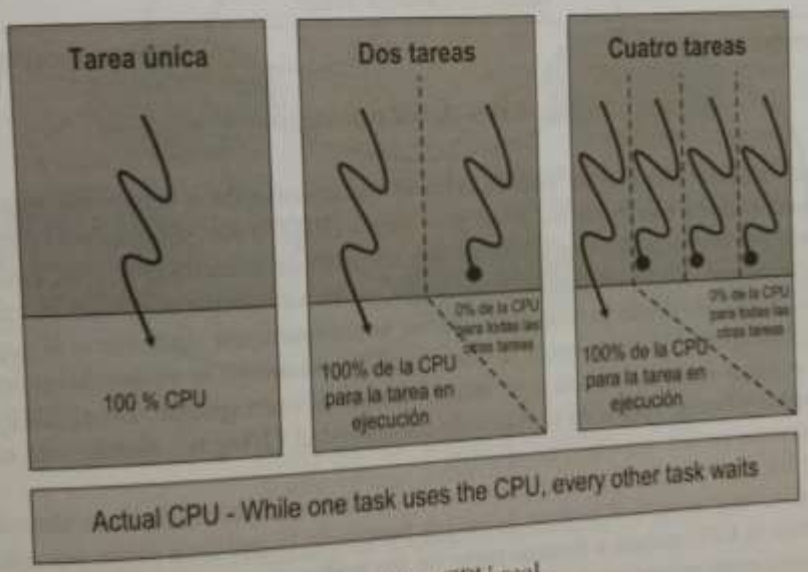


Fig. 4.9 – CPU real.

El tiempo de espera de una tarea es el que se considera que la tarea hubiera «merecido» en el modelo de equidad. A la tarea que está ejecutando en la CPU, por su parte, también se le va contabilizando su tiempo de ejecución pero en el sentido inverso, va consumiendo (decrementando) su tiempo de espera (o «merecido») mientras que crece el tiempo de las otras tareas que esperan hasta que, en un punto, a otra con mayor «necesidad» de CPU le toca el turno y se expulsa a ésta. La tarea expulsada vuelve a esperar con las otras tareas a que le llegue el turno, pero no lo hace en una cola, sino que ingresa en un árbol binario

rojo-negro, como el que vemos en la figura 4.10, tomando en consideración su tiempo de ejecución virtual.

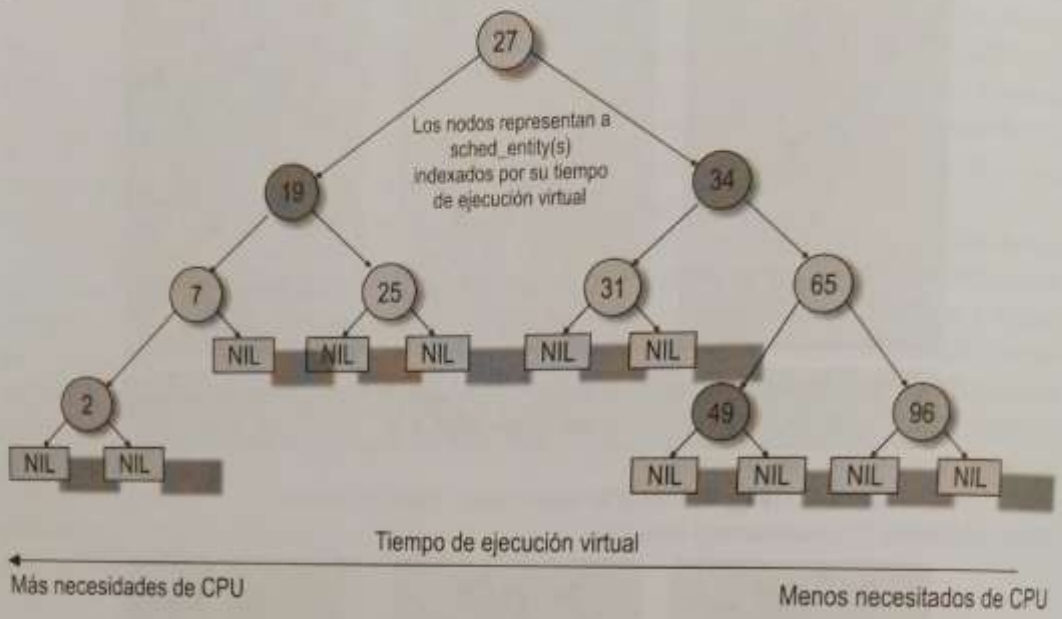


Fig. 4.10 – Árbol rojo-negro.

Está más allá del alcance de esta materia el tratamiento de a esta estructura de datos, pero solo necesitamos saber que un árbol rojo-negro (RBT o *Red-Black Tree*) es un tipo de árbol binario autobalanceado. Los nodos se van creando de acuerdo con el valor del tiempo de ejecución virtual. Dado un determinado valor, se lo va comparando con el valor de los otros nodos, partiendo desde la raíz: si es menor, la comparación sigue hacia la izquierda; si es mayor, la comparación sigue hacia la derecha. El nodo nuevo se va desplazando así hasta encontrar su lugar en la estructura y se inserta donde le corresponde. En el RBT se puede buscar, insertar y borrar nodos en tiempos de complejidad $O(\log n)$, siendo n el número de elementos (tareas en nuestro caso) en el árbol.

Otra de las ventajas de utilizar esta estructura es que está ordenada naturalmente, por lo que no es necesario que el algoritmo la revise buscando la próxima tarea por ejecutar. La tarea que dejó la CPU pasará a formar parte de los nodos que están más a la derecha en el árbol (las que tienen menor necesidad de CPU). El planificador siempre tomará el nodo que está en el extremo izquierdo, es decir, la tarea de menor tiempo de ejecución virtual o «mayor necesidad de CPU». Esto dará la sensación de que los nodos migran desde la derecha hacia la izquierda a medida que pasa el tiempo y que esperan su turno en una «cola ordenada» pero que de ninguna manera es una estructura FIFO.

Tampoco es necesaria la heurística para clasificar al proceso como «interactivo» o «no interactivo».

El CFS implementa prioridades mediante el uso de tareas ponderadas. A cada tarea se le asigna una ponderación basada en su prioridad estática. De manera que, mientras está ejecutando, la tarea con la menor ponderación (menor prioridad) verá pasar el tiempo a mayor

velocidad que aquella con una prioridad mayor. Esto significa que su variable contadora de espera se agotará más rápidamente que aquella tarea de mayor prioridad, de manera que las tareas de menor prioridad obtendrán menos tiempo de CPU comparadas con las tareas de mayor prioridad.

El planificador CFS no utiliza rodajas de tiempo en el sentido convencional; calcula la «rodaja de tiempo» para una tarea determinada justo antes de que sea planificada para su ejecución. Su longitud es variable (dinámica) y depende directamente de su prioridad estática y de la carga presente de la cola de ejecución, es decir, el número y prioridad de las tareas en la cola. Tampoco utiliza *jiffies*.

4.6 Planificación en Windows

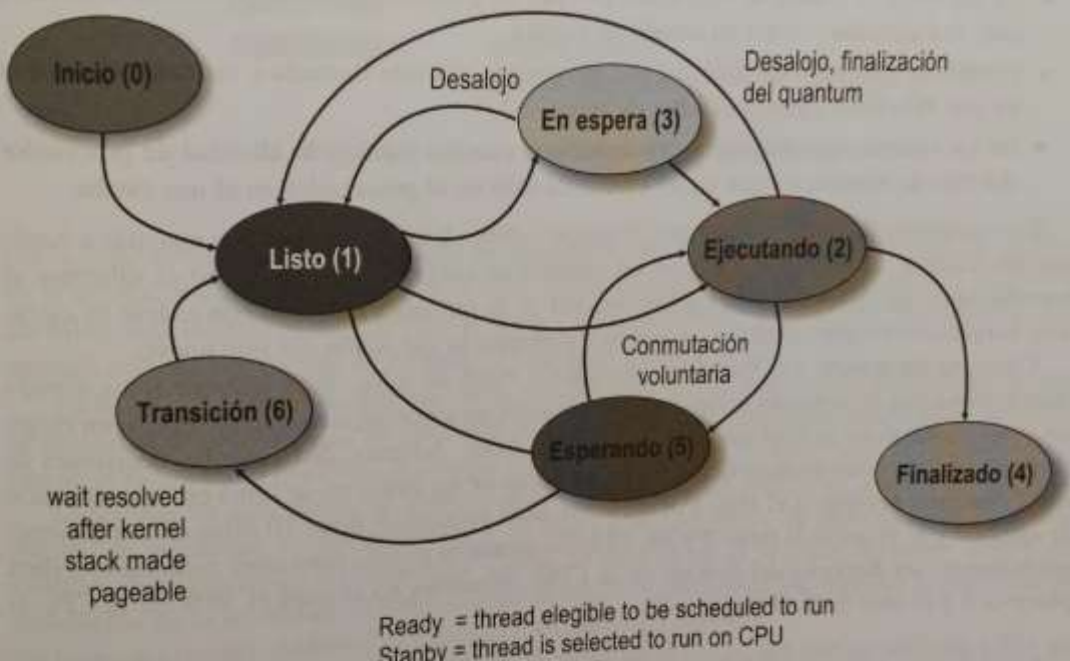


Fig. 4.11 – Diagrama de transición de estados en Windows.

Windows implementa un sistema de planificación con desalojo basado en prioridades. El hilo ejecutable (en estado listo) con la mayor prioridad es el elegido para ejecutar.

Cuando un hilo es seleccionado para ejecutar, lo hace durante una cantidad de tiempo que ya conocemos como *quantum*. Ejecutará durante ese lapso de tiempo que le está permitido ejecutar hasta que a otro hilo en el mismo nivel de prioridad se le dé turno para ejecutar, como vemos en la figura 4.11. Los valores de *quantum* pueden variar de un sistema a otro y de un proceso a otro por cualquiera de estos tres motivos: indicaciones en la configuración del sistema (largos o cortos), procesos en primer o segundo plano, o uso de objetos de trabajo (o *job objects*, en inglés) para alterar el *quantum*.

Un hilo podría no alcanzar a completar su *quantum*. Dado que Windows implementa planificación con desalojo, si otro hilo con una prioridad mayor queda listo para ejecutar, el hilo que está ejecutando en ese momento podría ser desalojado antes de que finalice su rodaja de tiempo. De hecho, un hilo puede estar seleccionado para ejecutar y ser desalojado incluso antes de que comience su *quantum*, afirman M. E. Russinovich y D. A. Solomon (2004).

El código de planificación de Windows está implementado en el núcleo, pero no hay una rutina o módulo «planificadora» sino que el código está diseminado por todo el núcleo en el cual ocurren eventos relacionados con la planificación. A las rutinas que llevan a cabo estas tareas se las denomina colectivamente «el activador del núcleo».

Los eventos siguientes pueden requerir la activación de hilos:

- Un hilo queda listo para ejecutar. Por ejemplo, acaba de ser creado o acaba de ser liberado de su estado de espera.
- Un hilo deja el estado de ejecución porque su *quantum* terminó, el hilo ha finalizado, cede la ejecución o entra en estado de espera.
- Cambia la prioridad del hilo, ya sea provocado por una llamada a sistema o porque el mismo Windows cambia el valor de prioridad.
- En un sistema con múltiples procesadores, cuando cambia la afinidad de procesador del hilo de manera tal que ya no ejecutará más en el procesador en el que estaba.

En cualquiera de estas situaciones Windows debe determinar qué hilo ejecutar a continuación. Cuando lo hace, lleva a cabo un cambio de contexto, que es, como ya sabemos, el procedimiento de salvaguardar el estado volátil de la máquina asociado con el hilo en ejecución, cargar otro estado volátil de otro hilo y reanudar la ejecución del hilo nuevo.

Como se ha notado ya, Windows planifica a nivel de hilos. Este enfoque toma sentido cuando tomamos en consideración que los procesos no ejecutan sino que solo proveen recursos y un contexto en el cual puedan ejecutar sus hilos. Además, dado que las decisiones de planificación se hacen estrictamente sobre las bases de los hilos, no se toma en consideración a qué procesos pertenece el hilo. Por ejemplo, si el proceso A tiene 10 hilos en condiciones de ejecutar y el proceso B tiene 2 y los 12 hilos tienen la misma prioridad, cada hilo recibirá teóricamente un doceavo del tiempo de la CPU. Windows no le dará el 50% de la CPU al proceso A y el otro 50% al proceso B.

4.6.1 Niveles de prioridad

Para comprender los algoritmos de planificación de hilos se debe primero entender los niveles de prioridad que utiliza Windows. Como se ilustra en la figura 4.12, internamente Windows utiliza 32 niveles de prioridad, que van desde 0 hasta 31. Estos valores se dividen de esta manera:

- 16 niveles de tiempo real (16 hasta 31).
- 15 niveles variables (desde 1 hasta el 15).
- Un nivel de sistema, el cero, reservado para el hilo de página cero (o *zero page thread*, en inglés).

Los niveles de prioridad de los hilos se asignan desde dos perspectivas diferentes: desde la API de Windows y desde el núcleo Windows. La API de Windows primero organiza procesos

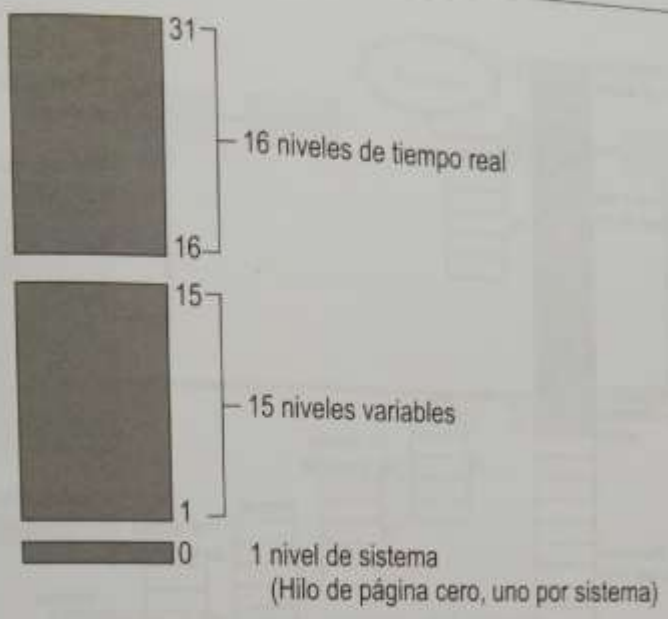


Fig. 4.12 – Niveles de prioridad de los hilos.

por la clase de prioridad a la cual son asignados en la creación (tiempo real, alta, por encima de la normal, normal, por debajo de la normal e inactivo) y luego por la prioridad relativa de los hilos individuales dentro de esos procesos (tiempo crítico, el más alto, por encima de la normal, normal, por debajo de normal, el más bajo e inactivo).

En la API de Windows cada hilo tiene una prioridad base que está en función de su clase de prioridad de proceso y de su prioridad de hilo relativa. La correspondencia entre la prioridad de Windows y la prioridad numérica interna de Windows se aprecia en la figura 4.13.

Dado que un proceso tiene solo un único valor base de prioridad, cada hilo tiene dos valores de prioridad: el del momento y el base. Las decisiones de planificación se toman basadas en la prioridad del momento. Como se explica en la sección siguiente acerca del alzamiento de la prioridad, el sistema bajo ciertas circunstancias incrementa la prioridad de los hilos en el rango dinámico (1 hasta 15) por períodos cortos.

Windows nunca ajusta la prioridad de los hilos en el rango del tiempo real (16 a 31), de manera que siempre tienen la misma prioridad base y corriente.

La prioridad inicial base del hilo se hereda de la prioridad base del proceso. Un proceso, por defecto, hereda su prioridad base a partir del proceso que lo creó. Este comportamiento puede pasarse por alto en la función `CreateProcess` o utilizando el comando `START`. También se puede cambiar la prioridad de un proceso después de creado utilizando la función `SetPriorityClass` o distintas herramientas que exponen tal función, tales como `TaskManager` y `Process Explorer` (pulsando el botón derecho del *mouse* sobre el proceso y eligiendo una clase nueva de prioridad). Por ejemplo, se puede disminuir la prioridad de un proceso que consume mucha CPU de manera tal que no interfiera con las actividades normales del sistema. Mediante el cambio de la prioridad de un proceso se aumentan y disminuyen las prioridades individuales del hilo dentro de un proceso porque, a menos que usted haya escrito el programa o tenga el código fuente, realmente no sabrá qué están haciendo los hi-

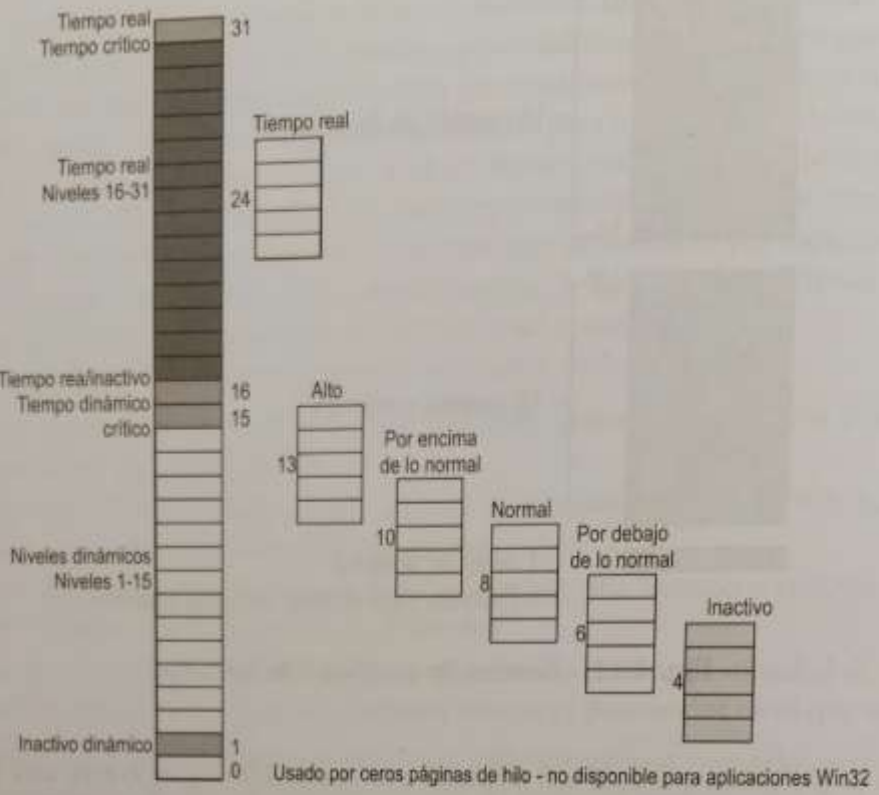


Fig. 4.13 – Correspondencia entre las prioridades del núcleo Windows y su API.

los y cambiar su importancia relativa puede provocar que el programa no se comporte de la manera pretendida.

Normalmente, la prioridad base del proceso (y, por lo tanto, la prioridad base de comienzo del hilo) asumirá un valor en el medio de cada rango de prioridad de procesos (24, 13, 10, 8, 6 o 4). Sin embargo, algunos procesos del sistema Windows (tales como el `Session Manager`, controlador de servicio y el servidor local de autenticación de seguridad) tienen una prioridad base de proceso ligeramente más alta que la establecida para la clase Normal (8). Este valor por defecto más alto asegura que los hilos en estos procesos comenzarán en una prioridad más alta que el valor 8 establecido por defecto. Estos procesos usan una llamada a sistema interna (`NTSetInformationProcess`) para indicar que su prioridad base de proceso será un valor numérico distinto del valor normal de prioridad base de comienzo.

4.6.2 Base de datos del activador

Con el objeto de llevar el registro de todos los procesos, hilos y sus propiedades y para tomar decisiones de planificación, el núcleo usa un conjunto de estructuras de datos conocidas colectivamente como la base de datos del activador (o *dispatcher database*, en inglés), ilustrada en la figura 4.14. Esta base de datos lleva el registro de qué hilos están esperando para ejecutar y qué procesadores están ejecutando qué hilos.

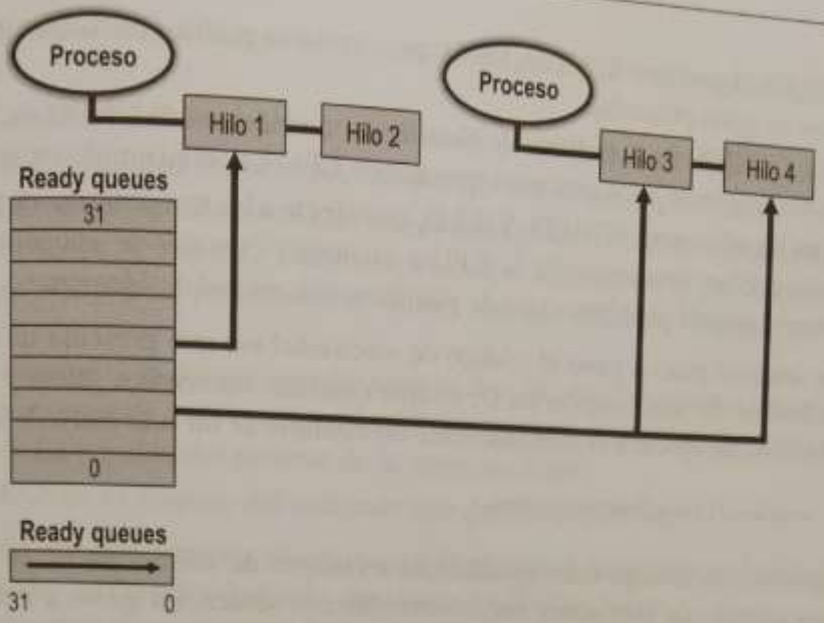


Fig. 4.14 – Base de datos del activador.

El procesador tiene asocidas un conjunto de 32 colas, llamadas «colas de listos» (o *ready queues*, en inglés), una por nivel de prioridad (desde 0 a 31) y cada una contiene punteros a los hilos en ese nivel de prioridad y que están en estado «listo» para ejecutar.

Como algunas colas pueden no contener valores porque no hay hilos con esas prioridades y para acelerar la selección del hilo a ejecutar, Windows mantiene una máscara de 32 bits llamada el resumen de listos (*ready summary*, en inglés) que indica si hay al menos un hilo en la cola de listos para un nivel de prioridad en particular (el bit 0 representa a la prioridad 0 y así sucesivamente).

Además existe otra máscara de bits llamada el resumen de inactivos (*idle summary*, en inglés) que indica qué procesadores están ociosos. El acceso a estas colas está protegido por un *spinlock* por procesador.

De manera que se puede tomar una decisión de planificación acerca de qué hilo ejecutar a continuación sobre la base de un tiempo contante.

4.7 Práctica con Linux

Como decíamos anteriormente POSIX proporciona planificación basada en prioridades. Estas prioridades pueden ser modificadas dinámicamente mediante servicios específicos.

La planificación en Linux se lleva a cabo en la función «`schedule`» del archivo `kernel/sched.c`.

Linux implementa el esquema de planificación definido en las extensiones de tiempo real de POSIX. Existen tres formas de planificar un proceso:

- SCHED_FIFO: Un proceso con este tipo de planificación tiene asociada una prioridad estática. Son más prioritarios que los procesos «normales». Una vez que han conseguido el procesador, sólo pueden ser expulsados por otro proceso más prioritario. Esto significa que no tienen un *quantum* máximo de ejecución.

- SCHED_RR: Igual que SCHED_FIFO, pero con una política de Round Robin para los procesos de igual prioridad.
- SCHED_OTHER: Es la política de planificación «clásica» de UNIX: un Round Robin con mayor *quantum* a los procesos que menos CPU han consumido recientemente.
- Existe un bit adicional, SCHED_YIELD, que afecta a las tres políticas de planificación, haciendo que un proceso ceda la CPU a cualquier otro que se encuentre preparado, únicamente para el próximo ciclo de planificación, no indefinidamente.

Podemos analizar paso a paso el código de «`schedule`», que presenta un aspecto poco ortodoxo tratándose de una función en C, ya que contiene numerosos saltos y etiquetas. La idea es que la línea de ejecución más frecuente no conlleve ni un solo salto. Véalo con:[4]

```
less /usr/src/linux/kernel/sched.c
```

Los fragmentos de código correspondientes a cuerpos de sentencias `if` se han «quitado de en medio» a modo de subrutinas implementadas con sentencias `goto` a una etiqueta. Al final de la mayoría de los fragmentos hay un salto que vuelve a la instrucción siguiente al `goto`, a través de una etiqueta de vuelta con el mismo nombre más el sufijo `_back`.

4.7.1 Cambio de contexto

Linux implementa el «sistema de archivos proc» que es como una especie de «mapa» de estructuras de datos en memoria, expresados como si fueran archivos.

Podemos con el comando:

```
cat /proc/stat
```

Ver el número de cambios de contexto en la línea `cntxt`, note que con la flecha de cursor hacia arriba se recuperan los comandos escritos, por lo tanto si recupera este comando y lo ejecuta repetidas veces notará cuantos cambios de contexto se efectúan en pocos segundos, aún cuando supuestamente el sistema «no está haciendo nada». Repase lo aprendido en la subsección 2.1.4.

Luego de repasar la subsección 2.1.2, estamos en condiciones de encarar la práctica: Comenzaremos por introducir el comando «`top`».

El comando «`top`» provee una vista de la actividad del procesador en «tiempo real». Muestra un listado de los procesos (tareas) mas intensivos en CPU en el sistema, y puede proveer una interfaz interactiva para la manipulación de procesos. Las columnas son:

- PID: el Identificador de Proceso de cada tarea.
- USER: el nombre de usuario del propietario del proceso.
- PRI: la prioridad del proceso.
- NI: El valor «`nice`» (mejorado) del proceso. Los valores negativos tienen prioridad más alta.

4. Es probable que en lugar de `linux` exista un directorio que incluya el número de versión del núcleo, por ejemplo `linux-2.4.20`, por eso cuando haya escrito la palabra `linux` presione la tecla de tabulador una o dos veces, esto invoca la función de autocompletar del intérprete `bash`.

- SIZE: El tamaño del código del proceso mas los datos mas el espacio de pila, en kilobytes.
- RSS: Cantidad total de memoria física usada por el proceso, en kilobytes.
- SHARE: La cantidad de memoria compartida usada por el proceso.
- STAT: El estado del proceso: S (*Sleeping*) durmiendo, D sueño ininterrumpible, R (*running*) corriendo, Z zombie, T (*traced*) trazado o detenido, «<» procesos con valor «nice» negativo, N para procesos con valores «nice» positivos, W para procesos en área de *swap*.
- %CPU: La porción del proceso del tiempo de CPU desde la última actualización de la pantalla, expresada como porcentaje del tiempo total de CPU por procesador.
- %MEM: La porción del proceso de la memoria física.
- COMMAND: El nombre del comando que generó el proceso.

El comando «`nice`» corre un programa con la prioridad de planificación modificada. Los rangos van desde -20 (la prioridad más alta) hasta 19 (la más baja).

El comando «`kill`» envía una señal a un proceso, generalmente la de «matar» o terminar un proceso, pero no necesariamente.

El comando «`yes`» imprime por pantalla una serie de «y» es en forma interminable. Para finalizar (abortar) el comando/proceso debe presionar simultáneamente `CTRL-C`, esto envía una «señal» desde el teclado que le indica al comando/proceso que debe finalizar.

Podemos redirigir la salida de este comando a un archivo utilizando a la derecha del comando el carácter «>» (mayor). Existe un archivo/dispositivo especial que funciona como un «agujero negro» o una papelera de reciclaje, todo lo que va a parar ahí se pierde inmediatamente; este archivo/dispositivo especial es «`/dev/null`».

Todo comando/proceso hasta tanto no finalice, no nos entrega el «prompt» del intérprete de comandos, si está ejecutando desde una consola/terminal. A esto el UNIX lo denomina «foreground». Y en contraposición existe el *background*, lo que está ejecutando «por detrás» o «tras bambalinas» en forma no asociada a una consola/terminal. Esto permite que un proceso disparado desde una consola corra permanentemente, desafectando a la consola/terminal para que el usuario pueda correr otros comandos/procesos.

Para enviar un comando al *background* debe colocar al final de la línea el carácter «&».

Abra una sesión como usuario «`ubuntu`» y ejecute el comando «`top`».

Abra otra sesión como usuario «`root`» de esta manera:

```
sudo su - root
```

en otra consola/terminal y ejecute el comando:

```
yes > /dev/null &
```

Verá a continuación una respuesta del sistema como ésta:

```
[1] 829
```

El «829» es el número PID otorgado al proceso en ejecución, el «`[1]`» indica el número de tarea otorgado por el intérprete de comandos `bash` a la tarea que está ejecutando en *background*. El comando ha quedado ejecutando enviando las «y» es a «`/dev/null`» en donde

se pierden sin remedio. A pesar de que el proceso aún no finaliza nos ha devuelto el *prompt* para seguir colocando comandos (es decir, quedó corriendo en *background*).

Vuelva a la sesión del usuario «ubuntu» y verá corriendo el proceso «yes» más o menos entre los primeros lugares.

Observe que el orden NO es permanente ¿Puede explicar por qué?

Vuelva a la sesión del usuario «root» y coloque ahora el comando

```
nice -n -15 yes > /dev/null &
```

Es decir, el mismo comando que recién pero ahora «mejorado» con una prioridad «-15». El sistema le contestará por ejemplo:

```
[2] 830
```

El significado es similar al primer caso. Vuelva a la sesión del usuario «ubuntu» y observe ahora el comportamiento de «top».

Observe y compare los valores de las columnas PID, PRI, NI, %CPU, STAT.

¿Qué conclusiones saca? Tome notas de todo lo observado

Vuelva ahora a la sesión del usuario «root» y coloque el comando:

```
kill%2
```

Apretando la tecla «enter» dos veces, el sistema le comunicará que el proceso ha sido terminado. Puede recuperar los anteriores comandos con las flechas de dirección hacia arriba (o hacia abajo) y modificar su contenido con las flechas de dirección a izquierda y derecha.

Experimente colocando distintos valores de «nice», en lugar de «-15» pruebe con «15» por ejemplo. Experimente sin trabajar con el superusuario «root», sólo con el usuario no-privilegiado «ubuntu». ¿Pueden enviarse señales entre procesos con el mismo UID o es necesario que los envíe el superusuario «root» con UID=0?

Observe y compare los resultados.

4.7.2 Una analogía

Podemos hacer una analogía si nos imaginamos lo que ocurre en un Banco frente a la ventanilla de una caja. El cajero es el procesador, los clientes son los procesos que llegan para ser atendidos. Algunos clientes traen pocas facturas para pagar y rápidamente son despachados, otros traen muchas y tardan mucho más. El Banco podría tener la política de atender al cliente hasta que haya terminado por más que se demore mucho, o bien podría tener la política de no aceptar más de cierta cantidad de boletas o cierto tiempo de atención, en cuyo caso el cliente es «expulsado» para dar paso al siguiente. Este cliente puede volver a ponerse en la cola y repetir el proceso hasta que haya hecho todos sus trámites. Para esto puede haber un policía encargado de «desalojar» al cliente de la ventanilla. El Banco también puede tener la política de darle prioridad a los ancianos y a las embarazadas, de manera que cuando llegan estos «procesos» a la cola se los atiende primero. Puede haber una sola cola para varios cajeros o una cola para cada ventanilla.

Sistemas operativos

MARTÍN SILVA



Alfaomega

5 Sincronización

5.1 Antecedentes

Decíamos en la sección 2.3.3 que en un sistema de computación concurren múltiples hilos que acceden en mayor o menor grado a los mismos recursos, y que es necesario hacerlos acordar organizadamente en el tiempo, es decir, sincronizar. En Ciencias de la Computación, el término se refiere más bien a las **relaciones** entre fenómenos o **eventos** –cualquier número de eventos y cualquier clase de relación, tal como antes, durante o después–, explica Downey (2005).

Los programadores a menudo se enfrentan a restricciones de sincronización que son requerimientos propios del orden de los eventos, tales como: el evento A debe ocurrir antes que el evento B o, los eventos A y B no deben ocurrir a la vez.

En la vida cotidiana utilizamos un reloj para forzar la sincronización; en la electrónica se utiliza una señal de reloj para sincronizar eventos y, por lo tanto, en las computadoras también, pero en los sistemas de computación no siempre podemos valernos de un reloj para lograrla, ya sea porque no existe un reloj universal –como en el caso de los sistemas distribuidos– o porque no podemos conocer con la suficiente resolución (mínimo error del instrumento de medición) cuándo ocurren los eventos. Por estos motivos debemos valernos de técnicas en software para forzar la sincronización.

Idealmente, si tuvieramos una computadora capaz de cargar un único programa y ejecutarlo de principio a fin, no tendríamos necesidad alguna de sincronización. Pero como tenemos a muchos programas, de distinta naturaleza, ejecutando simultáneamente y compitiendo por los mismos recursos, es necesario que exista una sincronización para que no desemboquen en un estado inesperado. Es decir, deberían poder completar su tarea como si estuvieran ejecutando solos en esa computadora ideal.

	Hilo 1	Hilo 2
1	LOAD X R1	LOAD X R1
2	ADD R1 1	ADD R1 1
3	STORE R1 X	STORE R1 X

Tabla 5.1

5.2 Alternativas de sincronización

Esta labor de sincronización se lleva a cabo en todos los ámbitos: en el hardware hay instrucciones especiales para ello, y también en el software, tanto en el núcleo del sistema operativo como en bibliotecas y también en las aplicaciones, ya que algunos lenguajes de programación permiten especificar sincronización de eventos.

Nos hemos referido a distintos modelos de ejecución en las computadoras. En el más simple, la computadora ejecuta una instrucción luego de otra en clara secuencia. Así, la sincronización es muy sencilla ya que podemos observar que el orden de los eventos viene dado por el orden de las instrucciones del mismo.

Básicamente, hay dos formas de complicar a este modelo. Una manera es que la computadora sea paralela, es decir, que tenga varios procesadores ejecutando hilos (o procesos) a la vez. En este modelo no nos será sencillo saber si una instrucción en un procesador se ejecuta antes que otra en otro procesador. Por ello, diferiremos su tratamiento hasta el capítulo 8.

La otra forma es considerar un modelo con un único procesador pero ejecutando varios hilos «a la vez», conmutando en fracciones de segundo de uno a otro, dándonos así la ilusión de simultaneidad. Estas decisiones de conmutación no las toman las aplicaciones –como ya vimos– sino el planificador del sistema operativo. Por lo tanto el programador no puede saber cuándo serán ejecutadas las instrucciones de los diferentes hilos. Pero si han de compartir recursos, lo que sí puede hacer el programador es diseñar cuidadosamente su aplicación teniendo en cuenta las distintas alternativas de sincronización que veremos.

Por ejemplo, consideremos un programa en el que dos hilos se ejecutan concurrentemente para sumar 1 a la variable compartida x.

Tengamos en cuenta que lo que realmente se ejecuta concurrentemente son las líneas de código generadas por el compilador. En nuestro caso, supongamos que una instrucción de incremento de una variable contadora como $x:=x+1$ da lugar a tres instrucciones de un lenguaje ensamblador cualquiera, que es lo que realmente va a ejecutar el procesador. Tendríamos los siguientes pasos:

1. Cargar desde memoria el valor de x en un registro.
2. Incrementar el valor del registro.
3. Almacenar el contenido del registro en la posición de memoria de x.

Así pues, lo que realmente se va a ejecutar de forma concurrente dentro de cada bucle es:

Cada una de estas instrucciones es atómica o indivisible, es decir, se va a ejecutar en un ciclo de reloj del procesador sin poder ser interrumpidas. Puesto que hemos dicho que en pro-

gramación concurrente existe un orden parcial, cualquier intercalado entre estas instrucciones es válido.

x	0	0	0	0	1	1	1
H1	1	2			3		
H2			1	2		3	

Tabla 5.2 – Traza de una posible ejecución concurrente.

En el Cuadro 5.2 puede verse una traza para un intercalado particular de estas seis líneas de código en el que el valor final de la variable *x* no es el esperado. Se ha perdido un incremento.

El problema estriba en que dos hilos distintos están accediendo al mismo tiempo a una variable compartida entre los dos para actualizarla.

> Una situación como ésta, en la que varios procesos o hilos acceden a, y manipulan, los mismos datos de forma concurrente, y el resultado de la ejecución depende del orden en que haya ocurrido el acceso, se denomina **condición de carrera** (o **competencia**) (del inglés *race condition*).

Nos hubiese interesado que las tres líneas de cada hilo se hubiesen ejecutado en un solo paso, sin ningún tipo de intercalado con las otras líneas del otro hilo.

5.2.1 Sección crítica

Dado que todos los hilos de un proceso comparten un espacio de direcciones en común, se deben usar algunos mecanismos para prevenir que más de un hilo intente acceder a los mismos datos en forma simultánea. En el ejemplo mostrado, si estos dos hilos necesitan incrementar la misma variable global –y para que eso ocurra en forma segura–, cada uno de ellos deberá asegurar que tiene acceso exclusivo a esta variable durante algún período de tiempo.

> Denominaremos **sección crítica** al segmento de código en el cual un hilo está accediendo en exclusividad a un **recurso compartido** (una variable, una estructura de datos, un dispositivo) y que no debe ser accedido concurrentemente por otro.

Así, la ejecución de secciones críticas de los procesos o hilos es **mutuamente exclusiva** en el tiempo. El problema de la sección crítica consiste en diseñar un **protocolo** que los hilos puedan usar para cooperar.

Cada hilo debe solicitar permiso para ingresar en su sección crítica. La sección de código que implementa esta solicitud es la sección de ingreso. La sección crítica puede ir seguida de una sección de egreso. El código que queda es la sección restante.

El problema de la sección crítica se podría resolver fácilmente en un entorno uniprocesador si pudiéramos prohibir las interrupciones mientras se está modificando una variable compartida. Así, podríamos tener certeza de que la secuencia de instrucciones actual se ejecutará en orden sin expropiación. Ninguna otra instrucción se ejecutaría, y sabríamos que la variable compartida no puede sufrir modificaciones inesperadas. Desafortunadamente, esta solución no es factible en un entorno multiprocesador.

El método más común para evitar que dos procesos o hilos accedan al mismo tiempo a un recurso compartido es el de la exclusión mutua.

5.2.2 Exclusión mutua

Entonces, para prevenir que varios hilos accedan a los mismos datos en forma simultánea (concurrente), es suficiente con asegurar que, cuando un hilo está ejecutando en una sección crítica, a ningún otro hilo se le permita ejecutar en una sección crítica en la que se acceden a los mismos recursos. Es decir, las ejecuciones de secciones críticas en las cuales los mismos recursos son accedidos por hilos, deben ser **mutuamente excluyentes** en el tiempo, una luego de la otra pero en un orden arbitrario. Cuando un hilo se está ejecutando en su sección crítica ningún otro hilo puede ejecutarse en esa sección crítica al mismo tiempo, para evitar las condiciones de carrera. El problema de la exclusión mutua fue identificado por primera vez por Dijkstra (1965b).

Obviamente, este problema puede presentarse tanto en aplicaciones como en el núcleo del sistema operativo. Consideremos el ejemplo presentado por Carretero Pérez (2001) para la generación hipotética de un número identificador de proceso (PID) en un sistema tipo Unix ejecutando en un multiprocesador y que se ilustra en la figura 5.1; si el código del núcleo estuviera mal escrito, se perdería un incremento y dos procesos tendrían el mismo PID. Si bien esta figura ilustra lo que podría ocurrir en un sistema con varios procesadores, el mismo error podría suceder en un sistema con un único procesador si el sistema operativo tuviera que realizar un cambio de contexto a un segundo hilo antes de que el primero terminara de actualizarlo.

El código de un sistema operativo que no contemple secciones críticas provocará fallos erráticos porque al ejecutarse en un orden arbitrario, difícilmente lograremos que se vuelva a repetir y, por lo tanto, será muy difícil de detectar para poder corregirlo.

El problema de la exclusión mutua, si bien es importante para todos los sistemas operativos, es específicamente importate e intrincado para un sistema operativo para multiprocesamiento simétrico (SMP), en el cual el mismo código ejecuta simultáneamente en más de un procesador, compartiendo ciertas estructuras de datos almacenadas en la misma memoria global.

Existen soluciones tanto en hardware como en software para forzar la exclusión mutua.

5.2.3 Soluciones hardware

5.2.3.1 Inhabilitación de las interrupciones

En el hardware, y tratándose de un sistema con un único procesador, la solución más simple para lograr la exclusión mutua es mediante la desactivación de las interrupciones

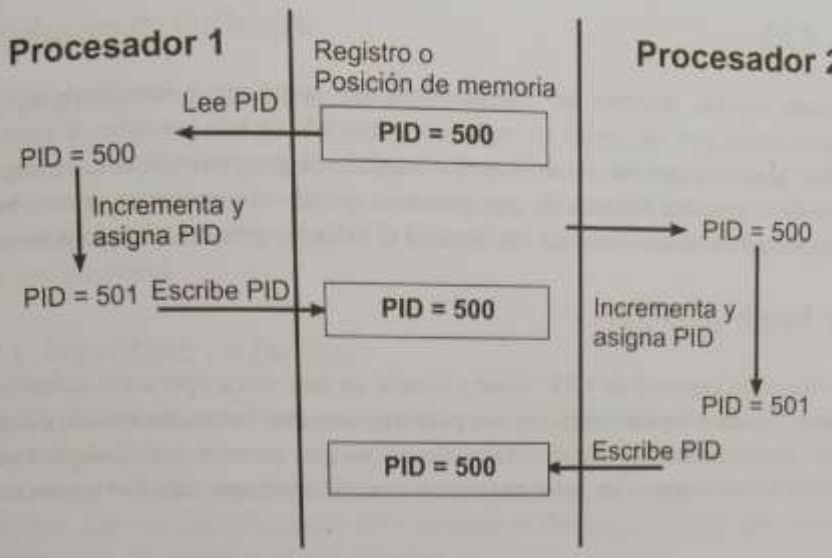


Fig. 5.1 – Generación de PID en un sistema multiprocesador.

durante la ejecución de la sección crítica. Esto evitará que ejecute cualquier rutina de servicio de interrupciones y que sea desalojado el proceso o hilo. Si bien esta solución es efectiva trae aparejados varios inconvenientes: si la sección crítica es larga entonces el reloj del sistema sufriría un retraso cada vez que se ejecute alguna sección crítica ya que su rutina no sería procesada. Si un proceso no sale de su sección crítica el control nunca pasará a otro proceso, deteniendo todo el sistema. En cambio, la **espera activa** (del inglés *busy waiting*) es efectiva tanto en sistemas con un único procesador como con varios. No obstante, la espera activa tiene el inconveniente de desperdiciar ciclos de CPU.

5.2.3.2 Instrucciones especiales de máquina

5.2.3.2.1 TSL

Muchos procesadores cuentan con instrucciones que comprueban y modifican (TSL, de *Test and Set Lock*, por sus siglas en inglés) el contenido de una palabra de la memoria en un registro, almacenando al mismo tiempo un valor distinto de cero en dicha palabra de memoria (generalmente un 1) en caso que la comprobación se haya realizado con éxito. Al tratarse de una instrucción máquina, el procesador nos garantiza que la instrucción TSL es realizada sin ningún tipo de interrupción por parte de otro hilo, es decir que las operaciones de lectura y escritura son atómicas. En un sistema multiprocesador, ninguno de los demás procesadores tendrá acceso a la palabra hasta que termine de ejecutarse la instrucción. El hilo invocador obtiene el cerrojo si el valor era cero y da vueltas (*spins*) escribiendo un 1 en la variable hasta que lo logre.

Existe también una solución más elaborada que TSL, que es la instrucción *test and test-and-set*, cuyo propósito principal es evitar ese «revoloteo» o *spin* sobre la memoria. Comprueba si está libre y si lo está entonces efectúa un *test-and-set*.

5.2.3.2.2 CAS

De manera similar, tenemos las instrucciones que comparan e intercambian (CAS, de *Compare-and-Swap*, por sus siglas en inglés) el contenido de una posición de memoria con un valor dado y, si son iguales, modifican el contenido de esa posición con un valor nuevo. También opera de manera atómica, lo que garantiza que el valor nuevo se calcula basado en información actualizada. La escritura fracasará si el valor ha sido actualizado por otro hilo.

5.2.3.2.3 Fetch-and-add

Esta instrucción especial de CPU «trae y suma» en una única operación atómica el contenido de una posición de memoria. Se usa para implementar exclusión mutua y algoritmos concurrentes en sistemas con varios procesadores, ya que permite que cualquiera de ellos incremente en forma atómica un valor en memoria previniendo que otro intente hacer lo mismo.

5.2.3.2.4 Read-modify-write

Esta instrucción lee, modifica y escribe. Es una clase de operación atómica como *test-and-set*, *fetch-and-add* y *compare-and-swap*. Leen una posición de memoria y escriben un valor nuevo en ella en forma simultánea, ya sea con un valor completamente nuevo o con alguna función del valor previo.

5.2.3.2.5 Load-link/store-conditional

O LL/SC son un par de instrucciones utilizadas en presencia de múltiples hilos para lograr sincronización. La instrucción `load-link` devuelve el valor actual de una posición de memoria mientras que una instrucción `store-conditional` posterior a la misma posición de memoria almacenará un valor nuevo sólo si no han ocurrido actualizaciones a esa posición desde la instrucción `load-link`. Juntas implementan una operación `read-modify-write` atómica libre de espera. Fueron propuestas por Jensen, Hagensen y Broughton (1987) para el multiprocesador S-1 AAP.

5.2.3.2.6 Soluciones libres de espera

Si bien los cerrojos son efectivos se puede provocar una sobrecarga producida por la espera en la contienda por el recurso. Una implementación libre de espera (del inglés *wait-free*), según Herlihy (1991), es una que garantice que cualquier proceso puede completar cualquier operación en un número finito de pasos, sin tener en cuenta las velocidades de ejecución de los otros procesos. Herlihy demostró que la instrucción `test-and-set` tiene un número finito de **consenso** (del inglés, *consensus number*), en contraste con la operación `compare-and-swap`; no obstante, esta última es una solución más general al problema.

5.2.4 Soluciones software

5.2.4.1 El problema de la sección crítica

Las primeras soluciones propuestas para lograr la exclusión mutua fueron, obviamente, en software. A partir del primer análisis en el artículo clásico de Dijkstra (1965a) –ilustrado mediante un ejemplo similar al visto[1] en la figura 5.1–, se presentaron varios algoritmos para resolver este problema.

5.2.4.1.1 Algoritmo de Dekker

El matemático holandés Dekker diseñó un algoritmo de exclusión mutua que fue publicado por Edsger Dijkstra (tanto en 1965b como en 1968a). Este método tiene la ventaja de ilustrar la mayoría de los errores habituales que se producen en la construcción de programas concurrentes. Existen cinco versiones del algoritmo de Dekker, los cuatro primeros presentan ciertos fallos pero el quinto es el más eficiente.

Su solución garantizaba la exclusión mutua pero mediante un programa complejo, difícil de seguir y cuya corrección es difícil de demostrar y permitía la posibilidad de que un proceso quedara atascado para siempre en la sección de entrada (inanición) mientras que otros procesos permanecerían en la sección crítica. Así, lo señaló D. E. Knuth (1966) quien además presentó una solución que no provocaba inanición, garantizando el acceso para N computadoras dentro de $2^{N-1}-1$ turnos. Este algoritmo fue modificado posteriormente por de Bruijn (1967) para proveer acceso dentro de los $\frac{1}{2}N(N-1)$ turnos. Eisenberg y McGuire (1972) presentaron otro en el que se garantizaba que una computadora no esperaría más de $N-1$ turnos. Todos estos algoritmos funcionan bien siempre y cuando no se produzca un fallo en alguna de las N computadoras, el fallo de una unidad detendría el sistema completo. Lamport (1974) presentó una solución simple que permitía que el sistema continuara operando a pesar del fallo de un componente individual.

5.2.4.1.2 Algoritmo de Peterson

Peterson (1981) desarrolló una solución simple y elegante en la que una variable global indica la posición de cada proceso con respecto a la exclusión mutua y otra, variable global resuelve los conflictos de simultaneidad.

5.2.4.1.3 Requisitos de una solución

En definitiva, y para sintetizar, para Silberschatz y Galvin (1997) una solución al problema de la sección crítica debe satisfacer los tres requisitos siguientes:

- **Exclusión mutua**: Si un hilo está ejecutando código de la sección crítica, ningún otro hilo lo podrá hacer.
- **Progreso**: Si ningún hilo está ejecutando dentro de la sección crítica, la decisión de qué hilo entra en la sección se hará sobre los hilos que desean entrar. Los hilos que no

1. Por supuesto, en ese trabajo no se habla de hilos sino de computadoras.

quieren entrar no pueden formar parte de esta decisión. Además, esta decisión no podrá posponerse indefinidamente, debe realizarse en tiempo finito.

- **Espera limitada**: Hay un límite para el número de veces que se permite a otros hilos ingresar en sus secciones críticas después de que un hilo haya solicitado ingresar en la suya y antes de que se le otorgue la autorización para hacerlo.

No obstante, ya Brinch Hansen (1973) había presentado criterios similares en su obra *Operating System Principles*.

5.2.4.2 Variables mutex

Dos técnicas usadas comúnmente en software para lograr exclusión mutua entre hilos son las variables mutex y las variables de condición.

> Una variable ***mutex*** es un objeto en un programa que sirve para trabar y se usa para lograr exclusión mutua (y de ahí su nombre: de la contracción en inglés de *mutual exclusion* o exclusión mutua). Está siempre en uno de dos estados: «trabado» o «destrabado»[2] (*locked/unlocked*).

Un hilo que quiere ejecutar en una sección crítica realiza una operación `lock()` sobre la variable mutex correspondiente. Si la variable mutex está en el estado destrabado, la operación `lock()` tiene éxito y el estado de la variable mutex cambia de destrabado a trabado en una única acción atómica. Después de esto, el hilo puede ejecutar en la sección crítica.

Sin embargo, si la variable mutex ya está trabada, dependiendo de la implementación, la operación `lock()` se maneja de una de las siguientes maneras:

1. El hilo se bloquea e ingresa en una cola de hilos que espera la variable mutex.
2. Se devuelve al hilo un código de estado indicando un fallo. En este caso, el hilo tiene la flexibilidad de continuar con algún otro trabajo. Sin embargo, para ingresar a la sección crítica, el hilo tiene que seguir reintentando trabar la variable mutex hasta que lo logre.

Una biblioteca que implemente hilos (por ejemplo, POSIX Threads) puede soportar ambas, proveyendo operaciones diferentes para trabar efectivamente y obtener el estado de una variable mutex.

En un sistema multiprocesador, en el cual diferentes hilos corren en paralelo en CPU diferentes, puede ocurrir que dos hilos realicen una operación de traba en la misma variable mutex, simultáneamente. En tal situación, uno de ellos gana y el perdedor se bloquea, o bien, retorna un código de estado indicando fallo.

Cuando un hilo finaliza de ejecutar en su sección crítica, realiza una operación `unlock()` sobre la variable mutex correspondiente. En este momento, si se usa el método bloqueante y si uno o más hilos están bloqueados esperando la variable mutex, a uno de ellos se les da la traba y su estado se cambia de «bloqueado» a «corriendo», mientras que los otros continúan esperando.

2. Valores binarios «uno» o «cero».

5.2.4.3 Variables de condición

Las variables mutex son simples de implementar porque tienen sólo dos estados. Sin embargo, su uso está limitado a vigilar ingresos a las secciones críticas. Para requerimientos de sincronización más generales, se usan las **variables de condición**.

> Una variable de condición está asociada con una variable mutex y refleja un estado lógico[3] (*boolean*, en inglés) de esa variable.

Las variables de condición indican eventos y no poseen ningún valor. Normalmente, las dos operaciones que se proveen para una variable de condición son `wait()` y `signal()`. Cuando un hilo realiza una operación `wait()` sobre una variable de condición, se destraba la variable mutex asociada y se bloque el hilo hasta que algún otro hilo realice una operación `signal()` sobre la variable, indicando que puede haber ocurrido el evento que se estaba esperando. Cuando un hilo realiza una operación `signal()` sobre la variable de condición, la variable mutex se traba y el hilo que estaba bloqueado esperando sobre la variable de condición comienza a ejecutar en la sección crítica. La figura 5.2 ilustra el uso de la variable mutex y de la variable de condición para la sincronización de hilos.

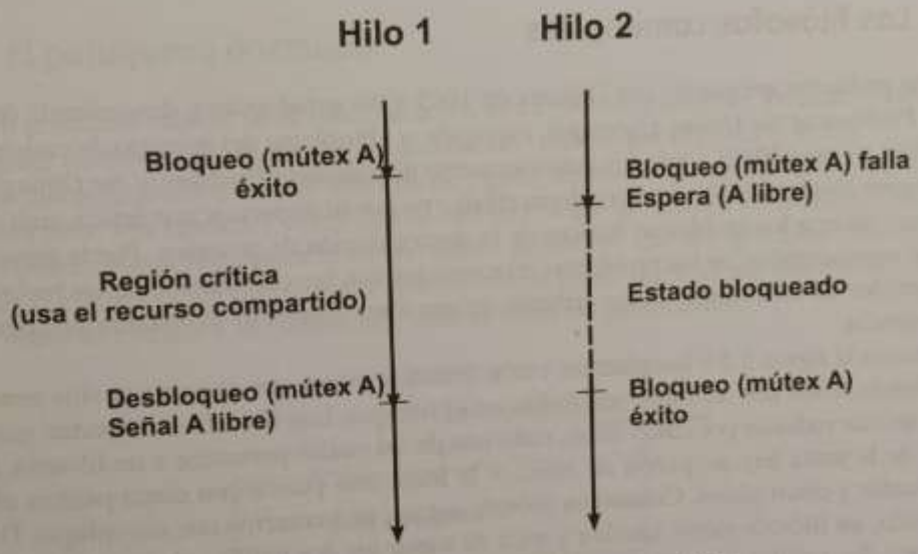


Fig. 5.2 – Variable mutex y variable de condición.

Las variables de condición se usan a menudo para la cooperación entre hilos. Por ejemplo, en una aplicación en la cual dos hilos de un proceso tienen una relación productor-consumidor, el hilo productor crea datos y los coloca en un *buffer* limitado, y el hilo consumidor toma el dato desde el *buffer* y lo usa para su procesamiento. En este caso, si el *buffer* está vacío cuando el hilo consumidor lo revisa, ese hilo puede estar hecho para esperar a una variable de condición **no-vacía**, y cuando el hilo productor coloque algunos datos en el *buffer*, puede activar la señal de la variable de condición **no-vacía**. De manera similar, si el

3. Es decir, «verdadero» o «falso».

buffer está lleno cuando el hilo productor lo revisa, ese hilo puede estar hecho para esperar una variable de condición **no-llena**, y cuando el hilo consumidor tome algunos datos desde el *buffer*, puede activar la señal de la variable de condición **no-llena**. De esta manera, los dos hilos pueden trabajar en cooperación uno con otro mediante el uso de variables de condición.

5.2.4.4 Spinlock

Como decíamos en el apartado 4.1.4, un *spinlock* es una traba (*lock*) que un hilo puede adquirir para tener acceso exclusivo a un recurso y si al tratar de adquirirla no está disponible, simplemente esperará en un lazo comprobando repetidamente (*spin*) hasta que lo esté. Dado que el hilo permanece activo pero no está realizando una tarea útil, es una forma de espera activa. Por esto, es un mecanismo que se usa cuidadosamente en el núcleo del sistema operativo, sobretodo para arquitecturas con varios procesadores simétricos (SMP). Es ventajoso utilizarlos si el tiempo de espera es menor que la sobrecarga que introduciría un cambio de contexto.

5.3 Problemas clásicos

5.3.1 Los filósofos comensales

Este problema, propuesto por Dijkstra en 1965 a sus estudiantes y denominado originalmente *Problem of the Dining Quintuple*, en inglés, o «Problema del quinteto de comensales» (ven Misa y Frana 2010), y que fue posteriormente denominado *Problem of the Dining Philosophers* por Hoare (1978), es un problema clásico no por su importancia práctica, sino porque sirve para ilustrar los problemas básicos de la sincronización de procesos. Puede verse como un caso representativo de los problemas relacionados con la coordinación de los recursos no compartibles de un sistema. Es un ejemplo de una clase amplia de problemas de control de concurrencia.

Veamos la figura 5.3 e imaginemos cinco filósofos chinos que se pasan la vida pensando y comiendo (estas dos acciones son finitas en el tiempo). Los filósofos comparten una sola mesa circular rodeada por cinco sillas, cada una de las cuales pertenece a un filósofo. En el centro de la mesa hay un platón de arroz, y la mesa está puesta con cinco palillos chinos individuales y cinco platos. Cuando un filósofo piensa, no interactúa con sus colegas. De vez en cuando, un filósofo siente hambre y trata de tomar los dos palillos chinos que le quedan más cerca (los palillos que están entre él y sus vecinos izquierdo y derecho). Un filósofo sólo puede tomar un palillo a la vez y, obviamente, no puede tomar un palillo que ya está en la mano de uno de sus vecinos. Cuando un filósofo hambriento tiene sus dos palillos al mismo tiempo, come sin soltarlos; cuando termina de comer, los deja sobre la mesa y comienza a pensar otra vez.

La solución al problema, por lo tanto, consiste en inventar un ritual (algoritmo) que permita comer a los filósofos.

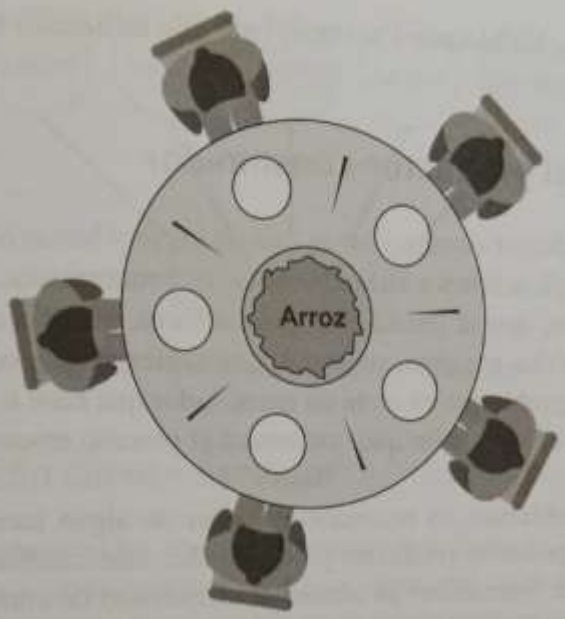


Fig. 5.3 – Filósofos comensales.

5.3.2 El peluquero dormilón

Otro problema clásico de la sincronización entre procesos es el del Peluquero Dormilón (o *Sleeping Barber Problem*, en inglés) presentado también por Dijkstra (1965a):

Hay una peluquería que tiene una sala de espera separada del salón de peinados. La sala de espera tiene una entrada y cerca de ella una puerta que conduce a la habitación donde se corta y peina; ambas comparten la misma puerta deslizante que siempre cierra una de ellas; además, la entrada es tan pequeña que sólo un cliente puede ingresar por ella por vez, fijando así su orden de entrada. Las exclusiones mutuas están así garantizadas.

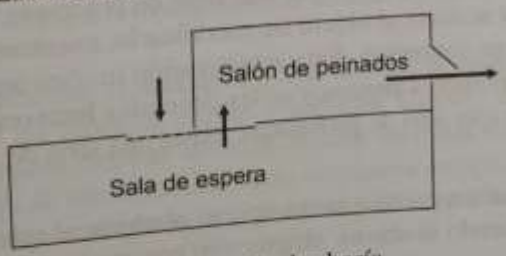


Fig. 5.4 – La barbería.

Cuando el peluquero ha finalizado su corte de cabello, abre la puerta a la sala de espera y la inspecciona. Si la sala de espera no está vacía, invita al próximo cliente, sino se va a dormir en una de las sillas en la sala de espera. El comportamiento complementario de los clientes es el siguiente: cuando encuentran cero o más clientes en la sala de espera, simplemente esperan su turno, pero si encuentran al Peluquero Dormilón lo despiertan.

Para estos problemas, se deben proponer alternativas de solución que coordinen la sincronización, el acceso a las secciones críticas, la exclusión mutua, y que eviten las condiciones

de carrera, la inanición y los bloqueos mutuos; todo ello utilizando las técnicas que veremos a continuación.

5.3.3 Problema del productor-consumidor

El problema del productor-consumidor es uno de los problemas más habituales que surge cuando se programan aplicaciones utilizando procesos concurrentes. En este tipo de problemas, uno o más procesos, que se denominan **productores**, generan cierto tipo de datos que son utilizados o consumidos por otros procesos, que se denominan **consumidores**. Un claro ejemplo de este tipo de problemas es el de un compilador que hace las funciones de productor al generar el código ensamblador que consumirá el proceso ensamblador para generar el código máquina.

En esta clase de problemas, es necesario disponer de algún mecanismo de comunicación que permita a los procesos productor y consumidor intercambiar información. Ambos procesos, además, deben sincronizar su acceso al mecanismo de comunicación para que la interacción entre ellos no sea problemática: cuando el mecanismo de comunicación se llene, el proceso productor se deberá quedar bloqueado hasta que haya hueco para seguir insertando elementos. A su vez, el proceso consumidor deberá quedarse bloqueado cuando el mecanismo de comunicación esté vacío, ya que en este caso no podrá continuar su ejecución al no disponer de información a consumir. Por tanto, este tipo de problema requiere servicios para que los procesos puedan comunicarse y servicios para que se sincronicen a la hora de acceder al mecanismo de comunicación.

5.3.4 Problema de los lectores-escritores

En este problema, –planteado por Courtois, Heymans y Parnas (1971)– existe un determinado objeto (véase figura 5.5), que puede ser un archivo, un registro dentro de un archivo, etc., que va a ser utilizado y compartido por una serie de procesos concurrentes. Algunos de estos procesos sólo van a acceder al objeto sin modificarlo, mientras que otros van a acceder al objeto para modificar su contenido. Esta actualización implica leerlo, modificar su contenido y escribirlo. A los primeros procesos se les denomina **lectores** y a los segundos se les denomina **escritores**. En este tipo de problemas, existe una serie de restricciones que han de seguirse:

- Sólo se permite que un escritor tenga acceso al objeto al mismo tiempo. Mientras el escritor esté accediendo al objeto, ningún otro proceso lector ni escritor podrá acceder a él.
- Se permite, sin embargo, que múltiples lectores tengan acceso al objeto, ya que ellos nunca van a modificar el contenido del mismo.

En este tipo de problemas, es necesario disponer de servicios de sincronización que permitan a los procesos lectores y escritores sincronizarse adecuadamente en el acceso al objeto.



Fig. 5.5 – Procesos lectores y escritores.

5.3.5 Comunicación cliente-servidor

En el modelo cliente-servidor, tenemos a un proceso llamado servidor, por una parte, que ofrece un servicio a otros procesos, por otra parte, que se denominan clientes (véase figura 5.6). El proceso servidor puede residir en la misma máquina que el cliente o en una distinta, en cuyo caso la comunicación deberá realizarse a través de una red de interconexión. Muchas aplicaciones y servicios de red, como el correo electrónico y la transferencia de archivos, se basan en este modelo[4].



Fig. 5.6 – Comunicación cliente-servidor.

En este tipo de aplicaciones, es necesario que el sistema operativo ofrezca servicios que permitan comunicarse a los procesos cliente y servidor. Cuando los procesos ejecutan en la misma máquina, se pueden emplear técnicas basadas en memoria compartida o archivos. Sin embargo, este modelo de comunicación suele emplearse en aplicaciones que ejecutan en computadores que no comparten memoria y, por tanto, se usan técnicas basadas en pase de mensajes.

5.4 Semáforos

El primer gran avance en la resolución de problemas de procesos concurrentes llegó con el artículo publicado por Dijkstra (1965b).

Dijkstra estaba interesado en el diseño de un sistema operativo como un conjunto de procesos secuenciales cooperantes y en el desarrollo de mecanismos eficientes y fiables para

4. Los conceptos de cliente y servidor son conceptos de software, no de hardware. Si bien, por extensión, se le suele llamar servidor al equipo sobre el cual ejecuta software servidor.

dar soporte a la cooperación. Los procesos de usuario podrían utilizar estos mecanismos si el procesador y el sistema operativo los ofrecían, afirma Stallings (2008).

El principio fundamental es el siguiente: dos o más procesos pueden cooperar por medio de simples señales, de forma que se pueda obligar a detener a un proceso en una posición determinada hasta que reciba una señal específica. Cualquier requisito complicado de coordinación puede satisfacerse por medio de la estructura de señales adecuada.

Para la señalización y sincronización, se usa una entidad física para representar al recurso, a estas variables especiales se las llama **semáforos**. Para transmitir una señal por el **semáforo s** los procesos ejecutan la operación `signal(s)` mientras que para recibir una señal del **semáforo s**, los procesos ejecutan la operación `wait(s)`; si la señal correspondiente aún no se ha transmitido, el proceso se suspende hasta que tenga lugar la transmisión[5]. Los semáforos fueron inventados por Dijkstra (1968b) para el sistema operativo THE.

Las soluciones a la exclusión mutua discutidas en la sección 5.2.4.1.3 tienen cierta desventajas ya que los procesos esperan de una forma activa. Un proceso que no está haciendo nada ocupa un procesador, lo cual limita la eficiencia del sistema. Analizaremos la primitiva de comunicación entre procesos que denominamos semáforo y que bloquea al proceso en lugar de desperdiciar tiempo de CPU cuando no se le permite entrar a su sección crítica o realizar una acción determinada. Como son pocos los procesadores que introducen el concepto de semáforo dentro de su conjunto de instrucciones, no lo trataremos como una solución *hardware*. Sin embargo, el semáforo es un mecanismo de más bajo nivel que otros por su potencia.

Podemos considerar que un semáforo es un tipo de datos abstracto que, por regla general, sólo admite como posibles valores a enteros no negativos. Si el semáforo sólo admite como valores el 1 y el 0, estaremos hablando de un **semáforo binario**. En cualquier otro caso estaremos hablando de un **semáforo general**.

Una vez analizados los valores que puede tomar un semáforo nos quedaría analizar las distintas operaciones que se pueden realizar sobre los semáforos. Estas son:

- wait(s). Decrementa el valor de *s* si éste es mayor que cero. Si *s* es igual a 0, el proceso se bloqueará en el semáforo.
- signal(s). Desbloquea algún proceso bloqueado en *s*, y en el caso de que no haya ningún proceso incrementa el valor de *s*.

Teniendo en cuenta las anteriores definiciones, las operaciones `wait()` y `signal()` se pueden expresar de la siguiente manera

wait(s):	signal(s):
if s>0 then s:=s-1 else bloquear proceso;	if (hay procesos bloqueados) then desbloquear un proceso else s:=s+1

Tabla 5.3 – wait() y signal().

5. En el artículo original de Dijkstra y en gran parte de la bibliografía, se emplea la letra **P** para wait y **V** para signal; éstas son las iniciales de las palabras holandesas que significan comprobar (**proberen**) e incrementar (**verhogen**).

Tanto la operación `wait()` como la operación `signal()` implican una comparación y una actualización de la variable entera asociada o, en su caso, el bloqueo y desbloqueo de los procesos. Una de las principales características de los semáforos es que todas las operaciones asociadas al `wait()` y al `signal()` deben ejecutarse de forma indivisible, es decir, la utilización de un semáforo implica la exclusión mutua en la ejecución de las operaciones sobre él definidas.

Los semáforos basan su funcionamiento en el uso de variables globales; además, el uso y función de las variables no es explícito, es decir, no podemos determinar qué semáforos están siendo utilizados para la sincronización entre procesos y cuáles están encargándose del control de la exclusión mutua en el acceso a las secciones críticas.

Si a esto agregamos que las operaciones `wait()` y `signal()` están dispersas por el código y que nada nos protege sobre el uso indebido de las variables, el mantenimiento de los programas puede ser una tarea bastante costosa.

Para resolver estos problemas, necesitamos introducir una nueva herramienta que nos permita un control estructurado de la exclusión mutua y nos proporcione un mecanismo de control de las condiciones de sincronización igual de versátil que el proporcionado por los semáforos. Pero además, necesitamos introducir un cierto nivel de encapsulación de datos, para protegernos del uso indebido de las variables y estructuras por parte de los usuarios. En definitiva, necesitaríamos un mecanismo que permitiera la implementación de tipos abstractos de datos, al mismo tiempo que proporcionara mecanismos para garantizar el acceso exclusivo a los datos y para implementar condiciones de sincronización. Los monitores cumplen con estos requerimientos.

5.5 Monitores

Básicamente, un monitor es un mecanismo de abstracción de datos, lo que nos permite representar de forma abstracta un recurso compartido por medio de un conjunto de variables que definen su estado.

Formalmente, un monitor es una construcción del lenguaje de programación, similar a un objeto en la programación orientada a objetos. Un monitor puede verse como un módulo que consiste de variables y procedimientos. Las variables pueden accederse sólo llamando uno de los procedimientos del monitor. En este sentido, un monitor es similar a un objeto: un objeto tiene sus propios datos privados, que sólo pueden accederse por medio de métodos implementados por ese objeto. La diferencia con los objetos, es que el monitor permitirá a sólo un proceso por vez ejecutar un procedimiento. En otras palabras, si un proceso A está ejecutando un procedimiento que se contiene en un monitor (decimos que A ha **ingresado** al monitor), y un proceso B también llama uno de los procedimientos del monitor, B se bloqueará hasta que A finalice (es decir, hasta que A **deje** el monitor).

Las primeras descripciones de **procedimientos monitor** aparecen en el trabajo de Brinch Hansen (1970), quien luego presentó una propuesta para una representación estructurada de multiprogramación en un lenguaje de alto nivel (cf. Brinch Hansen 1972) y de un monitor como un método para estructurar un sistema operativo (cf. Brinch Hansen 1973). Posteriormente, en el trabajo de Hoare (1974) se desarrolla el concepto de monitor de Per Brinch Hansen. Finalmente, Brinch Hansen (1975) implementó el concepto de monitor en el lenguaje Pascal Concurrente (*Concurrent Pascal*).

Los monitores son más fáciles de controlar que los semáforos y se implementan como bibliotecas de programas.

5.5.1 Sintaxis de un monitor

Por ejemplo, consideremos un monitor simple para proteger una variable entera como se muestra a continuación. El monitor contiene una variable única (privada) llamada `count` que puede accederse sólo por medio de tres procedimientos (públicos) para leer su valor actual, incrementarlo, o decrementarlo en uno, respectivamente. La construcción `monitor` garantiza que cualquier proceso que llame uno de estos procedimientos puede acceder en forma atómica a los datos privados que se contienen en el monitor:

```
monitor Counter {
private:
    int count=0;
public:
    int value()  {return count;}
    void incr()  {count = count + 1;}
    void decr()  {count = count - 1;}
}
```

5.6 Sincronización en pase de mensajes

Los mecanismos estudiados hasta ahora basan su actuación en la existencia de una memoria común compartida por todos los procesos implicados, como decíamos en el apartado 2.3.4.

No obstante, estos mecanismos no pueden utilizarse en sistemas que no comparten memoria, por ejemplo, en sistemas distribuidos.

Cuando se quieren comunicar y sincronizar procesos que ejecutan en máquinas distintas, es necesario recurrir al **pase de mensajes**. En este tipo de comunicación, los procesos intercambian mensajes entre ellos, como se ilustra en la figura 2.11.

Es obvio que este esquema también puede emplearse para comunicar y sincronizar procesos que ejecutan en la misma máquina, en este caso los mensajes son locales a la máquina donde ejecutan los procesos. La máquina puede tener un único procesador o varios (SMP).

Cuando dos procesos, locales o remotos, se comunican intercambiando mensajes, suele ocurrir que uno de ellos sea más lento que el otro, ya sea en la transmisión como en la recepción. Este es un problema clásico del modelo de productor y consumidor, que se nos presenta frecuentemente en Ciencias de la Computación y que los resolvemos mediante la utilización de una **memoria de almacenamiento intermedio** o *buffer*. Sinha (1996) hace una exposición muy completa de este tema desde la perspectiva de los sistemas operativos distribuidos en los que este problema se presenta acrecentado.

Los mensajes pueden transmitirse desde un proceso a otro copiando el cuerpo del mensaje desde el espacio de direcciones del proceso transmisor al espacio de direcciones del proceso receptor (posiblemente a través del espacio de direcciones del núcleo).

En la comunicación entre procesos, la estrategia de almacenar mensajes está fuertemente relacionada con la estrategia de sincronización.

5.6.1 Características de la sincronización

Un tema central en la estructura de comunicación es la sincronización impuesta en los procesos que se comunican mediante las primitivas de comunicación. La **semántica** o interpretación utilizada para la sincronización puede clasificarse como **bloqueante** o **no bloqueante**. Se dice que una primitiva tiene una semántica no bloqueante si su invocación no bloquea la ejecución en su invocante (el control regresa casi inmediatamente al invocante), de otra manera una primitiva se dice ser de tipo bloqueante.

La sincronización impuesta sobre los procesos que se comunican depende básicamente de uno de dos tipos de semánticas usadas para las primitivas `send()` y `receive()`, que ya presentamos en el apartado 2.3.4.2.

En el caso de una primitiva `send()` bloqueante, luego de ejecutar la instrucción `send()`, el proceso transmisor se bloquea hasta que reciba un acuse de recibo desde el receptor indicando que ha recibido el mensaje. En cambio, para una primitiva `send()` no bloqueante, luego de ejecutar la instrucción `send()`, al proceso transmisor se le permite seguir adelante con su ejecución tan pronto como el mensaje haya sido copiado al *buffer*.

En el caso de una primitiva `receive()` bloqueante, luego de ejecutar la instrucción `receive()`, el proceso receptor se bloquea hasta que reciba un mensaje. En cambio, para una primitiva `receive()` no bloqueante, el proceso receptor sigue adelante su ejecución después de ejecutar la instrucción `receive()`, que devuelve el control casi inmediatamente justo después de decirle al núcleo dónde está el *buffer* del mensaje.

Un punto importante en una primitiva `receive()` no bloqueante es cómo el proceso receptor sabe que el mensaje ha llegado al *buffer* del mensaje. Uno de los dos siguientes métodos se usa con frecuencia para este propósito:

1. Consulta (*Polling*). En este método, se provee una primitiva `test()` para permitirle al receptor revisar el estado del *buffer*. El receptor usa esta primitiva para consultar periódicamente al núcleo para revisar si el mensaje ya está disponible en el *buffer*.
2. Interrupción. En este método, cuando el mensaje ha sido llenado en el *buffer* y está listo para que el receptor lo use, se utiliza una interrupción software para notificar al proceso receptor. Este método permite a dicho proceso continuar con su ejecución sin necesidad de hacer pedidos infructuosos a `test()`. A pesar de que este método es altamente eficiente y permite paralelismo máximo, su principal debilidad, según Tanenbaum (1995), es que las interrupciones en espacio de usuario hacen difícil la programación.

Una variante de la primitiva `receive()` no bloqueante es la primitiva `receive()` condicional, que también devuelve el control al proceso invocante casi inmediatamente ya sea con un mensaje o con un indicador de que no está disponible ningún mensaje.

En una primitiva `send()` bloqueante, el proceso emisor puede bloquearse para siempre en situaciones donde el potencial proceso receptor haya caído o el mensaje enviado se haya perdido en la red debido a una falla en la comunicación. Para prevenir esta situación, las primitivas `send()` bloqueantes a menudo usan un valor de tiempo límite de espera (*timeout*) para especificar un intervalo de tiempo luego del cual la operación `send()` se finaliza con un estado de error. El valor de tiempo límite de espera puede ser un valor preestablecido o se

les puede proveer a los usuarios la flexibilidad de especificarlo como un parámetro más de la primitiva `send()`.

También puede estar asociado un valor de tiempo límite de espera con una primitiva `receive()` bloqueante para prevenir que el proceso receptor quede bloqueado indefinidamente en situaciones donde el potencial proceso emisor haya caído o el mensaje esperado se haya perdido en la red debido a fallas en la comunicación.

Cuando ambas primitivas `send()` y `receive()` de una comunicación entre dos procesos usan semánticas de bloqueo, se dice que la comunicación es **sincrónica**, sino es asincrónica. Para comunicaciones sincrónicas, el transmisor y el receptor deben estar sincronizados para intercambiar un mensaje. Esto se ilustra en la figura 5.7

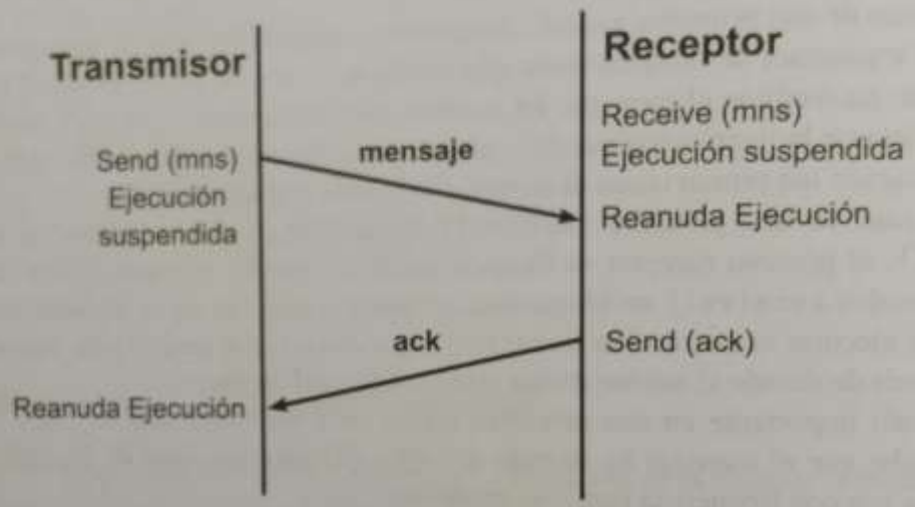


Fig. 5.7 – Modo sincrónico y bloqueante de comunicación.

Conceptualmente, el proceso transmisor envía un mensaje al proceso receptor, luego espera un acuse de recibo (*acknowledge*, en inglés, se abrevia ACK). Después de ejecutar la instrucción `receive()`, el receptor permanece bloqueado hasta que recibe el mensaje enviado por el transmisor. Al recibir el mensaje, el receptor envía un mensaje de acuse de recibo al transmisor. El transmisor retoma la ejecución sólo después de haber recibido este mensaje de acuse de recibo. Si tanto la transmisión como la recepción son bloqueantes, se logra la sincronización más fuerte.

Comparado con la comunicación asincrónica, la sincrónica es simple y fácil de implementar. También contribuye a la confiabilidad porque asegura al proceso transmisor que sus mensajes han sido aceptados antes de que el proceso transmisor retome la ejecución. Como resultado, si el mensaje se pierde o no se puede despachar, no es necesaria una recuperación tardía del error en el proceso transmisor para establecer un estado consistente y retomar la ejecución, observa Shatz (1984). Sin embargo, la principal desventaja de la comunicación sincrónica es que limita la concurrencia y está sujeto a interbloqueos en la comunicación. Es menos flexible que la comunicación asincrónica porque el proceso transmisor siempre tiene que esperar un acuse de recibo del proceso receptor aun cuando no es necesario. En un sistema que soporte varios hilos en un mismo proceso, las primitivas de bloqueo pueden usarse sin la desventaja de concurrencia limitada.

Un sistema de pasaje de mensajes flexible usualmente provee ambas primitivas (bloqueantes y no bloqueantes) para `send()` y `receive()` de tal manera que los usuarios pueden elegir la más adecuada para las necesidades específicas de sus aplicaciones.

5.6.2 Memoria intermedia o *buffer*

Los mensajes pueden transmitirse desde un proceso a otro copiando el cuerpo del mensaje desde el espacio de direcciones del proceso transmisor al espacio de direcciones del proceso receptor (posiblemente a través del espacio de direcciones del núcleo; la computadora transmisora y receptora puede ser la misma o no). En algunos casos, el proceso receptor puede no estar listo para recibir un mensaje que se le está transmitiendo, pero quiere que el sistema operativo guarde ese mensaje hasta que pueda leerlo. En estos casos, el sistema operativo confiará en que el receptor tenga un *buffer* en el cual los mensajes puedan almacenarse antes que el proceso receptor ejecute un código específico para recibir el mensaje.

En la comunicación interprocesos, la estrategia de almacenar mensajes está fuertemente relacionada con la estrategia de sincronización. Los nodos sincrónicos y asincrónicos de comunicación corresponden respectivamente a los dos extremos del *buffering*: un *buffer* nulo, o sin *buffer* y un *buffer* con capacidad ilimitada. Otras dos estrategias comúnmente usadas son las del *buffer* de mensaje único y la de límite finito o mensaje múltiple.

5.6.2.1 Buffer nulo

En este caso, no hay lugar para almacenar temporalmente el mensaje. Entonces, puede usarse una de las siguientes estrategias de implementación:

1. Cita o *rendez-vous*. El mensaje permanece en el espacio de direcciones del proceso transmisor y la ejecución del `send()` se demora hasta que el receptor ejecuta el correspondiente `receive()`. Para hacer esto, el proceso transmisor se respalda y suspende de manera tal que cuando se desbloquea, comienza por reejecutar la sentencia `send()`. Cuando el receptor ejecuta `receive()`, se envía un acuse de recibo al núcleo (local o remoto) del transmisor diciéndole que ahora puede enviar el mensaje. Al recibirlo el mensaje de acuse de recibo, el remitente se desbloquea, provocando que se ejecute el `send()` otra vez. Esta vez el mensaje se transfiere satisfactoriamente desde el espacio de direcciones del transmisor al espacio de direcciones del receptor porque el receptor está esperando recibir el mensaje.
2. Descarte. El mensaje simplemente se descarta y se usa el mecanismo de tiempo límite de espera (*timeout*) para reenviarlo el mensaje después de ese período. Esto es, después de ejecutar el `send()`, el proceso transmisor espera un acuse de recibo del proceso receptor. Si no se recibe ningún acuse de recibo dentro de un período de tiempo límite, asume que el mensaje ha sido descartado e intenta otra vez esperando que esta vez el receptor haya ejecutado el `receive()`. El transmisor puede tener que intentar varias veces antes de tener éxito. El transmisor se da por vencido después de reintentar un número predefinido de veces.

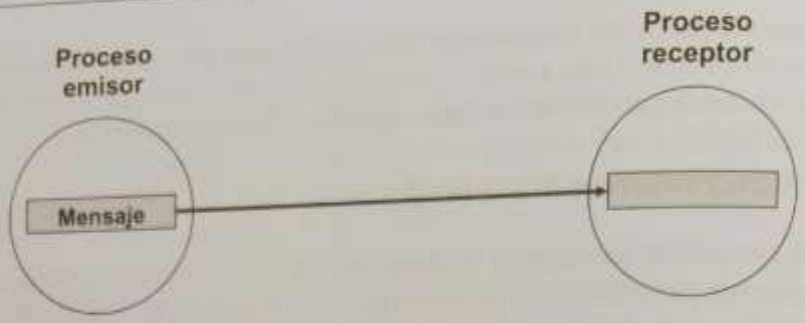


Fig. 5.8 – Buffer nulo.

Como se ilustra en la figura 5.8, en el caso del *buffer* nulo, la ruta lógica de transferencia del mensaje es directamente desde el espacio de direcciones del remitente al espacio de direcciones del receptor, involucrando a una única operación de copiado.

5.6.2.2 Buffer de mensaje único

La estrategia del *buffer* nulo generalmente no es apta para comunicaciones sincrónicas entre dos procesos en un sistema distribuido porque si el receptor no está listo, el mensaje debe transferirse dos o más veces, y el receptor del mensaje debe esperar todo el tiempo que toma transferir el mensaje a través de la red.

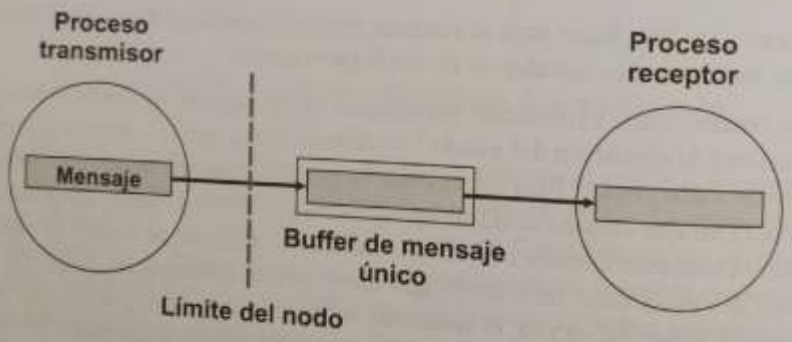


Fig. 5.9 – Buffer de mensaje único.

En un sistema distribuido, la transferencia de un mensaje a través de la red puede requerir un tiempo significativo en algunos casos. Entonces, en lugar de usar una estrategia de *buffer* nulo, los mecanismos de comunicación en sistemas de red/distribuidos usan una estrategia de *buffer* de mensaje único. En ésta, un *buffer* que tiene la capacidad de almacenar un único mensaje se usa en el nodo del receptor. Esto es porque en sistemas en comunicación sincrónica, un módulo de aplicación puede tener como máximo un mensaje pendiente por vez. La idea de la estrategia del *buffer* de mensaje único es mantener el mensaje listo para su uso en el lugar del receptor. Por lo tanto, en este método, el mensaje de pedido se almacena en el nodo del receptor si el receptor no está listo para recibir el mensaje. El *buffer* del mensaje puede estar en el espacio de direcciones del núcleo o en el espacio de direcciones del proceso

receptor. Como se muestra en la figura 5.9, en este caso, la ruta lógica de transferencia del mensaje involucra a dos operaciones de copia.

5.6.2.3 Buffer de capacidad ilimitada

En el modo asincrónico de comunicación, dado que el transmisor no necesita esperar que el receptor esté listo, puede haber varios mensajes pendientes que no hayan sido aceptados aún por el receptor. Por lo tanto, se necesita un *buffer* de mensaje de capacidad ilimitada que pueda almacenar todos los mensajes no recibidos para soportar comunicaciones asincrónicas con la garantía de que serán despachados todos los mensajes enviados al receptor.

5.6.2.4 Buffer de capacidad limitada

Es prácticamente imposible un *buffer* de capacidad ilimitada. Por lo tanto, en la práctica, los sistemas que usan modos asincrónicos de comunicación usan *buffers* de límite finito, también conocidos como de **múltiples mensajes**. Cuando el *buffer* tiene límites finitos, también es necesaria una estrategia para manejar el problema de un posible desborde del mismo, que puede tratarse,

1. **Comunicación no exitosa**. En este método, la transferencia de mensajes simplemente fracasa toda vez que no haya más espacio en el *buffer*. El `send()` normalmente regresa un mensaje de error al proceso transmisor, indicando que el mensaje no pudo ser despachado al receptor porque el *buffer* está lleno. Desafortunadamente, el uso de este método hace menos confiable al pase de mensajes.
2. **Comunicación de flujo controlado**. El segundo método consiste en usar control de flujo, lo que significa que el transmisor se bloquea hasta que el receptor acepta algunos mensajes, creando así espacio en el *buffer* para nuevos mensajes. Este método introduce una sincronización entre el transmisor y el receptor y puede dar como resultado estancamientos inesperados. Además, debido a la sincronización impuesta, el envío asincrónico no opera en el modo asincrónico verdadero para todos los comandos `send()`.

La cantidad de espacio de memoria a asignarse en la estrategia del *buffer* limitado depende de la implementación. En el enfoque más usado, se les provee a los usuarios una llamada a sistema del tipo `create_buffer()`, que al ser ejecutada por el proceso receptor crea un *buffer* (puede ser un *mailbox* o *port*) de un tamaño especificado por el receptor. El buzón (*mailbox*) del receptor puede ubicarse en el espacio de direcciones del núcleo o en el espacio de direcciones del proceso receptor. Si se ubica en el espacio de direcciones del núcleo, los buzones son considerados recursos del sistema que deben asignarse a procesos a medida que son requeridos. Esto tiende a limitar el número de mensajes que un proceso individual puede mantener en sus buzones. Por otra parte, si el buzón está ubicado en el espacio de direcciones del proceso receptor, el sistema operativo tendrá que confiar que el proceso asigne una cantidad apropiada de memoria, protegiendo al buzón de contratiempos y demás.

Como se muestra en la figura 5.10, en el caso de un `send()` asíncrono con una estrategia de *buffer* de límite finito, el mensaje primero es copiado desde la memoria del proceso

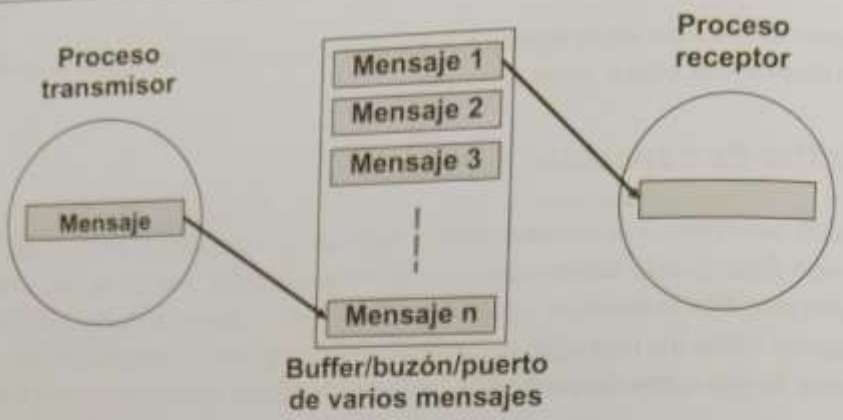


Fig. 5.10 – Buffer de límite finito.

transmisor al buzón del proceso receptor y luego copiado desde el buzón a la memoria del receptor, cuando éste lo solicita. Por lo tanto, involucra dos operaciones de copiado.

5.6.3 Sincronización en tuberías

Una tubería es un mecanismo de comunicación y sincronización. Desde el punto de vista de su utilización, es como un pseudoarchivo mantenido por el sistema operativo, como ya se dijo en el apartado 2.3.4.7. Conceptualmente, cada proceso ve la tubería como un conducto con dos extremos, uno de los cuales se utiliza para escribir o insertar datos y el otro para extraer o leerlos, es decir que la comunicación es unidireccional. La escritura se realiza mediante el servicio que se utiliza para escribir datos en un archivo. De igual forma, la lectura se lleva a cabo mediante el servicio que se emplea para leer de un archivo, como se muestra en la figura 5.11.

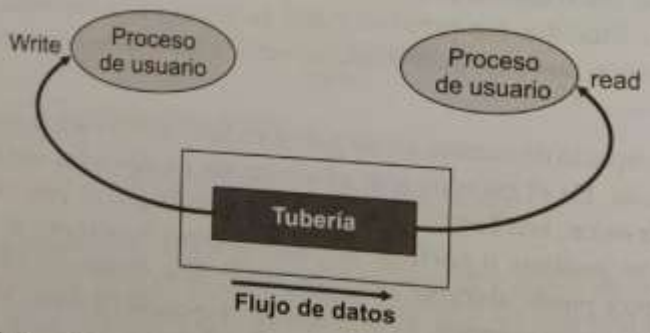


Fig. 5.11 – Comunicación unidireccional empleando una tubería.

Las tuberías son dispositivos seriales; los datos se leen siempre de la tubería en el mismo orden en el que fueron escritos.

Pueden tener o no un nombre. Las tuberías sin nombre no tienen forma de identificarse, por eso, sólo las puede acceder el proceso que las creó o sus descendientes, que lo heredan a través de la llamada a `fork()`.

En cambio, FIFO es una tubería que tiene un nombre en el sistema de archivos. Cualquier proceso puede abrir o cerrar un FIFO, y no necesitan estar relacionados entre ellos.

Las tuberías tienen un *buffer* de capacidad limitada con flujo controlado. Si el proceso escritor envía datos más rápidamente que los que pueda consumir el proceso lector –y si la tubería no puede almacenar más datos– el proceso escritor se bloquea hasta que haya más capacidad disponible. Si el proceso receptor intenta leer datos pero no hay disponibles, se bloquea hasta que haya. Así, la tubería automáticamente sincroniza dos procesos.

En Windows, las tuberías con nombre funcionan más como sockets, entonces pueden conectar procesos en computadoras separadas conectadas mediante una red.

5.6.4 Sincronización mediante señales

Las señales descritas en el apartado 2.3.4.4 pueden utilizarse para sincronizar procesos. Utilizando la norma POSIX, por ejemplo, un proceso (o hilo) puede bloquearse en el servicio `pause()` esperando la recepción de una señal que, o bien haga finalizar el proceso, o provoque la invocación de otra función. Esta señal esperada puede ser enviada por otro proceso (o hilo) mediante `kill()` o `sigusr1()`. Con este mecanismo, se consigue que unos procesos esperen a que se cumpla una determinada condición y que otros los despierten cuando se cumple la que les permite continuar su ejecución.

No obstante esto, advierte Carretero Pérez (2001), hay que tomar en consideración ciertas limitaciones si se pretende utilizar señales para sincronizar procesos. Por una parte, las señales tienen un comportamiento asíncrono. Un proceso puede recibir una señal en cualquier punto de su ejecución, aunque no esté esperando su recepción. Por otra parte, las señales no se encolan. Si hay una señal pendiente de entrega a un proceso y se recibe una señal del mismo tipo, la primera se perderá. Sólo se entregará la última. Esto hace que se puedan perder eventos de sincronización importantes.

5.7 Bloqueo mutuo

En un entorno de multiprogramación, varios procesos podrían competir por un número finito de recursos. Un proceso solicita recursos, si éstos no están disponibles, el proceso pasa a un estado de espera. Puede suceder que los procesos en espera nunca vuelvan a cambiar de estado, porque los recursos que solicitaron están en manos de otros procesos que también están esperando. Esta situación se llama **bloqueo mutuo, interbloqueo o *deadlock***.

Tal vez la mejor ilustración de un bloqueo mutuo sea la que ofrece una ley aprobada por la legislatura de Kansas a principios de siglo. Esta ley, en parte, decía:

> «Si dos trenes se aproximan uno al otro en un cruce, ambos harán alto total y ninguno arrancará de nuevo hasta que el otro se haya ido».

Un sistema consiste en un número finito de recursos que deben distribuirse entre varios procesos que compiten y una función del sistema operativo es la de distribución de recursos. Los recursos se dividen en varios tipos, cada uno de los cuales consiste en cierta cantidad de ejemplares idénticos. Cuando hablamos de recursos, pueden ser tanto físicos como lógicos –por ejemplo, la CPU, la memoria y los dispositivos son ejemplos de recursos físicos y los

archivos, semáforos y monitores son ejemplos de recursos lógicos. Un proceso debe solicitar un recurso antes de utilizarlo y debe liberarlo después de usarlo. En el modo de funcionamiento normal, un proceso sólo puede utilizar un recurso siguiendo la secuencia **solicitud, uso, liberación** (Silberschatz y Galvin 1997).

5.7.1 Condiciones para que se cumpla el interbloqueo

Para que ocurra un interbloqueo, se deben cumplir cuatro condiciones:

- **Exclusión mutua**: al menos un recurso debe adquirirse de modo que no pueda compartirse; es decir, sólo un proceso a la vez podrá usar ese recurso. Si otro proceso lo solicita, el proceso solicitante deberá esperar hasta que se haya liberado. Cada recurso puede ser asignado a sólo un proceso en un instante dado.
- **Retención y espera**: debe existir un proceso que haya adquirido al menos un recurso y esté esperando para adquirir recursos adicionales que ya han sido asignados a otros procesos. Los procesos no liberan los recursos asignados previamente mientras esperan que se les sean asignados solicitudes pendientes.
- **No expropiación**: los recursos no se pueden arrebatar; es decir, la liberación de un recurso siempre es voluntaria por parte del proceso que lo adquirió, una vez que ha terminado su tarea. Los recursos asignados previamente no pueden arrebatárseles a los procesos que los tienen.
- **Espera circular**: debe existir un conjunto $\{P_0, P_1, ..., P_n\}$ de procesos en espera tal que P_0 está esperando un recurso que fue adquirido por P_1, P_1 está esperando un recurso que fue adquirido por P_2, ..., P_{n-1} está esperando un recurso que fue adquirido por P_n, y P_n está esperando un recurso que fue adquirido por P_0. Es decir, existe una cadena de dos o más procesos de manera tal que cada proceso en la cadena tiene un recurso solicitado por el próximo proceso en la misma.

Hacemos hincapié en que se deben cumplir las cuatro condiciones para que ocurran bloqueos mutuos. La condición de espera circular implica la condición de retener y esperar, así que las cuatro condiciones no son del todo independientes. No obstante, es útil considerar cada condición por separado, como veremos luego.

5.7.2 Grafos de asignación de recursos

Los interbloqueos se pueden describir con mayor precisión en términos de un grafo dirigido, llamado **grafo de asignación de recursos del sistema**. Este grafo está compuesto por un conjunto de vértices V y un conjunto de aristas E. El conjunto V se divide en dos tipos de nodos distintos: $P = \{P_1, P_2, ..., P_n\}$, el conjunto de todos los procesos activos del sistema, representados gráficamente por un círculo, y $R = \{R_1, R_2, ..., R_m\}$, el conjunto de todos los tipos de recursos del sistema, representados gráficamente por un rectángulo; los puntos dentro de los rectángulos representan instancias de recursos. Un arco entre el proceso P_i y el recurso R_j indica que P_i solicitó una instancia del recurso R_j (arco de requerimiento) y se lo denota $P_i \rightarrow R_j$. Un arco entre el recurso R_j y el proceso P_i indica que el recurso R_j fue asignado al proceso P_i (arco de asignación) y se lo denota $R_j \rightarrow P_i$. El arco de requerimiento

sólo apunta al cuadrado R_j, pero un arco de asignación parte de uno de los puntos que están dentro del rectángulo.

Cuando el proceso P_i solicita un ejemplar del tipo de recursos R_j, se inserta un arco de solicitud en el grafo de asignación de recursos. En el momento en que dicha solicitud puede satisfacerse, el arco de solicitud se transforma instantáneamente en un arco de asignación. Cuando el proceso ya no necesita acceder al recurso, lo libera y la arista de asignación se borra.

Si el grafo no contiene ciclos, ningún proceso del sistema estará en bloqueo mutuo. Pero si el grafo contiene un ciclo, **podría** existir un bloqueo mutuo. Si cada tipo de recursos tiene exactamente un ejemplar, un ciclo implica que ha ocurrido un interbloqueo. Si en el ciclo interviene sólo un conjunto de tipos de recursos, en donde cada uno de ellos sólo tiene un ejemplar, entonces hay interbloqueo. Cada uno de los procesos que intervienen en el ciclo está en interbloqueo. En este caso, un ciclo en el grafo es condición necesaria y suficiente para la existencia de un interbloqueo. Si cada tipo de recursos tiene varios ejemplares, un ciclo no necesariamente implica que ha ocurrido un interbloqueo. En este caso, un ciclo en el grafo es una condición necesaria, pero no suficiente para la existencia de un interbloqueo.

5.7.3 Métodos para el tratamiento del bloqueo mutuo

Hay tres métodos principales para enfrentar el problema del bloqueo mutuo:

1. Podemos usar un protocolo que asegure que el sistema **nunca** llegará a un estado de bloqueo mutuo, ya sea **previniendo** o **evitando**.
2. Podemos permitir que el sistema entre en bloqueo mutuo, **detectarlo** y luego **recuperarlo**.
3. Podemos desentendernos del problema y hacer como si nunca ocurrieran bloqueos mutuos en el sistema. Esta solución es la que adopta la mayor parte de los sistemas operativos.

5.7.4 Prevención

El interbloqueo se puede prevenir si no permitimos que ocurra alguna de las cuatro condiciones mencionadas en 5.7.1. Mediante la implementación de una política que haga imposible una de las condiciones no aseguraremos que el sistema esté libre de interbloqueos. Sin embargo, el implementar tal política puede introducir más problemas de los que soluciona.

No es una solución práctica eliminar los accesos mutuamente exclusivos a cualquier recurso. Algunos recursos no pueden compartirse. No obstante, en algunos casos, recursos que aparentemente no se pueden compartir pueden hacerlo si no se requiere el acceso directo. Por ejemplo, una impresora podría parecer un recurso no compartible, ya que dos usuarios no pueden imprimir simultáneamente en ella. Sin embargo, no es importante el momento exacto en el que toma lugar la impresión. Por ende, un sistema de colas (*spooling*) permite a los procesos que accedan a una impresora virtual simultáneamente. Los trabajos de impresión se envían a la impresora real de una manera secuencial no compartible. A menos que se requiera un acceso directo a la impresora verdadera, el *spooling* elimina la condición de acceso mutuamente exclusivo al recurso impresora.

La condición de retención y espera nunca se cumplirá si los pedidos de recursos pueden ser hechos solamente por procesos que no retienen recursos. Un protocolo que puede usarse obliga a cada proceso a solicitar y recibir todos sus recursos antes de iniciar su ejecución. Hay dos grandes desventajas en esta política. Primero, un proceso puede no conocer qué recursos requerirá, los recursos necesitados pueden depender de sus cómputos. Un proceso que solicita todos los recursos que pudiera necesitar puede estar solicitando recursos que nunca usará en realidad. Aun si los recursos se usaran, el obligar a que las solicitudes se hagan antes de que comience el proceso puede dar como resultado que el proceso mantenga retenidos los recursos mucho tiempo antes de que sean realmente necesitados. Esto puede dar como resultado un desempeño pobre del sistema, debido a la baja utilización de los recursos. Esta estrategia también es muy susceptible a la inanición (*starvation*, en inglés). Un proceso que necesite varios recursos muy solicitados podría tener que esperar indefinidamente, porque al menos uno de los recursos necesitados siempre estará asignado a algún otro proceso.

Otra estrategia de prevención de la retención y espera es requerir que todos los procesos liberen todos los recursos mantenidos cuando se hace una solicitud. Desafortunadamente, en la mayoría de los casos, esta estrategia no es práctica. Luego de que un proceso haya escrito datos al disco, liberar el disco para solicitar la impresora pone en peligro los datos en el mismo.

Eliminar la condición de no expropiación es aún menos práctica que eliminar la de exclusión mutua. Expropiar un recurso esencialmente es compartir forzado. La expropiación puede aplicarse a dispositivos cuyo estado puede ser resguardado y luego restaurado. Por ejemplo, un proceso puede ser desalojado de la CPU, resguardando el estado de sus registros. Sin embargo, las características del dispositivo necesarias para hacer esto son esencialmente las mismas que las que se necesitan para crear un dispositivo virtual compartible para el cual no se requiere acceso mutuamente exclusivo.

La eliminación de la condición de espera circular es tal vez la más promisoria de las técnicas de prevención. Un método para romper el ciclo consiste en asociar cada recurso con un número único de prioridad. Los procesos sólo pueden solicitar un recurso si su prioridad es mayor que la de todos los recursos asignados al momento. Usando esta estrategia puede solicitarse cualquier recurso. Si el recurso no tiene una prioridad más alta que todos los recursos asignados al momento, el recurso asignado en este momento con una prioridad más alta debe ser liberado primero. En esas situaciones, este enfoque es similar al usado para eliminar la condición de retención y espera, pero esta estrategia es más flexible, permitiendo que sean asignados recursos bajo ciertas circunstancias cuando se hace un pedido adicional.

5.7.5 Cómo evitar el bloqueo mutuo

En lugar de prevenir alguna de las condiciones que debe estar presente para que exista el bloqueo mutuo, éste puede evitarse si nunca se asigna un recurso si puede provocar un bloqueo mutuo. Una forma simple de hacer esto es asignando recursos solamente a un recurso a la vez. A pesar de no ser eficiente, claramente no puede ocurrir un bloqueo mutuo. Se puede implementar una política de asignación más generosa, con información adicional acerca de las necesidades potenciales de recursos para todos los procesos.

Se dice que un sistema está en un **estado seguro** si hay una **secuencia de ejecución segura**.

Una secuencia de ejecución es un ordenamiento para la ejecución de procesos, de manera tal que cada proceso al ser ejecutado corre hasta que termina o queda bloqueado y todos los pedidos de recursos se entregan inmediatamente si el recurso está disponible. Una secuencia de ejecución segura es una secuencia de ejecución en la cual todos los procesos ejecutan hasta finalizar.

Un **estado inseguro** es un estado que no es seguro y, no necesariamente, uno de bloqueo mutuo. Un estado inseguro puede no estar interbloqueado, pero hay al menos una secuencia de solicitudes de procesos que podría hacer que el sistema quede interbloqueado. Si esa secuencia de pedidos ocurre, ningún conjunto de respuestas a esos pedidos le permitiría al sistema operativo evitar el estado de interbloqueo.

Para determinar si un estado es seguro o no, se debe suministrar un algoritmo con la siguiente información: la asignación de recursos actual a cada proceso, el número máximo de cada recurso que un proceso puede solicitar y el número total de cada recurso disponible actualmente en el sistema.

5.7.5.1 El algoritmo del banquero

El algoritmo del banquero, propuesto por Edsger W. Dijkstra, toma decisiones acerca de si otorgar recursos basándose en si esto pondrá o no en estado inseguro al sistema. Cuando se recibe una solicitud, el algoritmo primero verifica que el proceso no esté solicitando más recursos de los máximos especificados para solicitar, provocando un error por excepción por tratarse de una solicitud inválida. En el caso de tratarse de una solicitud válida, se compara el número de recursos solicitados con el número de recursos disponible. Si no hay disponibles suficientes, se deniega la solicitud. Si hay suficientes, el algoritmo crea una versión modificada del estado actual que refleje el efecto de otorgar el recurso. El algoritmo luego verifica si la versión modificada del estado del sistema es segura. El recurso es otorgado si y sólo si la versión modificada del estado del sistema es segura.

Si bien es muy elegante en su diseño, el algoritmo del banquero tiene una gran desventaja: requiere conocimiento del número máximo de recursos que cada proceso podría requerir. Esa información no está disponible en la mayoría de los sistemas. Por ende, pocos sistemas usan este tipo de algoritmo.

5.7.6 Detección del interbloqueo

En lugar de intentar prevenir o evitar el interbloqueo, los sistemas podrían permitir que éstos ocurran (esperando que lo hagan con poca frecuencia) y recuperarse luego de que hayan ocurrido. Tales estrategias requieren de mecanismos para detectar y recuperarse del interbloqueo.

Se puede utilizar un grafo de asignación de recursos para modelar el estado de la asignación de recursos y solicitudes. Tal como vimos en el apartado 5.7.2, los procesos se dibujan como círculos y los recursos como rectángulos. Dentro del rectángulo del recurso, se dibuja un punto por cada instancia de ese recurso. Por cada solicitud del recurso que está pendiente,

se dibuja un arco desde el proceso al rectángulo del recurso. Por cada solicitud acordada, se dibuja un arco desde un punto del recurso al proceso.

Se puede usar una versión reducida del grafo de asignación de recursos para determinar si existe un interbloqueo o no. Para reducir un grafo de asignación de recursos, se revisan los arcos asociados con cada proceso y cada recurso:

- Si un recurso sólo tiene arcos saliendo de él, lo que significa que no tiene solicitudes pendientes, se borran todos sus arcos.
- Si un proceso sólo tiene arcos hacia él, lo que significa que todos sus recursos solicitados han sido otorgados, se borran todos sus arcos.
- Si un proceso tiene arcos saliendo de él, pero por cada arco de requerimiento existe un punto de recurso disponible (es decir, un punto sin un arco saliendo de él, en el recurso al cual apunta el arco), se borran todos los arcos del proceso.

Si al revisar todos los procesos se encontró al menos uno por el que pudieron borrarse sus arcos, hay que recomenzar y reiterar el procedimiento. Se debe continuar hasta que no haya arcos restantes o bien hasta que a ningún proceso se le puedan borrar los arcos. El sistema está interbloqueado sí y sólo sí no quedan arcos.

Si bien un grafo reducido de asignación de recursos provee una descripción útil del estado del sistema para seres humanos, para una computadora es más adecuado el enfoque de un algoritmo. El algoritmo para determinar un interbloqueo es similar al algoritmo para detectar si el sistema está en un estado seguro. Podemos ver un ejemplo en Harris 2002.

5.7.7 Recuperación del interbloqueo

Tenemos tres enfoques para la recuperación a partir de un interbloqueo: desalojo automático, finalización automática o intervención manual.

Con el desalojo automático, el sistema operativo expropia un subconjunto de los recursos asignados. Se deben resolver tres problemas importantes para implementar una estrategia como ésta:

Selección: ¿Qué recursos y de qué procesos se deben expropiar? Si se toma la decisión de expropiar un recurso de un proceso, ¿deberían expropiarse todos sus otros recursos? Otros factores para tener en cuenta serían:

- La prioridad de un proceso.
- Cuánto tiempo ha estado ejecutando un proceso.
- Cuánto tiempo un proceso necesita seguir ejecutando.
- Cuántos recursos el proceso tiene actualmente.
- El tamaño de los diferentes conjuntos de recursos que, en caso de ser liberados, podrían romper el interbloqueo.
- Cuántos procesos se verían afectados.

Consecuencia: ¿Qué les ocurre a los procesos que tienen recursos expropiados? ¿El sistema puede mantener información de estado acerca de procesos, de tal manera que si son víctimas de desalojo puedan ser devueltos a un estado anterior? Mantener esa información puede ser costoso tanto en tiempo como en espacio de almacenamiento. Puede

no ser práctico el devolver un proceso a su estado anterior, puesto que algunos datos de entrada podrían ser irrecuperables, a menos que esos datos se hayan capturado y resguardado cuando el proceso ejecutó por primera vez, puede que se haya perdido la capacidad para procesar esa información.

Inanición: Si el mismo proceso es víctima repetidamente de desalojo, las permanentes vueltas atrás podrían incapacitarlo de finalizar.

En lugar de expropiar recursos, la **terminación automática** elimina el interbloqueo mediante la finalización de los mismos. Bien pueden finalizarse todos los procesos interbloqueados o también el sistema puede seleccionar un subconjunto de procesos interbloqueados para finalizarlos. Si bien es más drástica, la finalización de todos los procesos se puede hacer rápidamente en sistemas en los que es importante una resolución rápida. Si sólo se finaliza un subconjunto de los procesos, los mismos factores que encontramos en el desalojo automático los tendremos en la selección de los procesos por finalizar. También deberíamos hacer notar que la finalización de un proceso puede desencadenar efectos más allá del mismo, puesto que podrían verse perjudicados los procesos dependientes de los terminados y los archivos parcialmente actualizados pueden quedar en un estado inconsistente.

El enfoque de la **intervención manual** le delega toda la responsabilidad de resolver el problema al operador del sistema. Dadas las limitaciones de los enfoques automáticos, en muchos casos, esta es una opción atractiva. Algunos sistemas, tales como los *mainframes*, corren desatendidos, sin un operador de tiempo completo, y otros deben reaccionar a situaciones de interbloqueo al poco tiempo de su detección. En tales casos, la intervención manual no es una solución práctica.

5.7.8 Algoritmo del avestruz

Todos los algoritmos presentados hasta ahora para lidiar con los interbloqueos tienen una o más desventajas serias. El hecho es que no hay una buena forma de lidiar con los interbloqueos. Para muchos sistemas operativos, los interbloqueos aparecen infrecuentemente. De manera que, en muchos casos, el problema del interbloqueo se ignora, como un avestruz que oculta su cabeza en la tierra hasta que el problema desaparezca.

Los sistemas fallan por una gran variedad de razones. En cada caso, los diseñadores deben considerar si el costo de la solución se justifica por la magnitud del problema que se ha de resolver. ¿La frecuencia y los daños causados por los cortes de energía eléctrica justifican el costo de una unidad de suministro ininterrumpible (UPS, de *Uninterruptable Power Supply*)? ¿La frecuencia y los daños causados en los discos justifican el costo de un arreglo de discos RAID? De la misma manera, ¿la frecuencia y los daños provocados por un interbloqueo justifican el costo de implementar una estrategia para lidiar con los interbloqueos? En muchos casos, la respuesta es no.

5.8 Sincronización en POSIX

5.8.1 Señales

Habiendo visto los mecanismos de sincronización con los que cuenta Linux en el núcleo, pasemos ahora a considerar los mecanismos de sincronización que están disponibles para las aplicaciones.

Las señales son mecanismos para comunicarse con y manipular procesos en Linux. El tópico de señales es largo, aquí discutimos algunas de las señales más importantes y técnicas que se usan para controlar procesos.

Una señal es un mensaje especial enviado a un proceso. Las señales son asíncronas; cuando un proceso recibe una señal, la procesa inmediatamente, sin finalizar la función actual o aun la línea actual de código. Hay varias docenas de diferentes señales, cada una con un significado diferente. Cada tipo de señal está especificada por su número de señal, pero en los programas usualmente nos referimos a una señal por su nombre. En Linux, las señales están definidas en el archivo de cabecera `signum.h`, pero si queremos usarlas en un programa escrito en lenguaje C deberíamos incluir al archivo de cabecera `signal.h`.

Cuando un proceso recibe una señal, puede hacer una de varias cosas, dependiendo de la **disposición** de la señal. Para cada señal, hay una **disposición predeterminada**, la cual determina qué le pasa al proceso si el programa no especifica algún otro comportamiento. Para la mayoría de los tipos de señales el programa puede especificar algún otro comportamiento, ya sea ignorar la señal o llamar a una función especial **manejadora de señal** para responder a la misma. Si se usa un manejador de señales, se pausa el programa que se está ejecutando en ese momento, se ejecuta el manejador de señales y cuando éste finaliza, se retoma la ejecución del programa.

El sistema Linux envía señales a los procesos en respuesta a condiciones específicas. Por ejemplo, a un proceso que intenta realizar una operación ilegal pueden enviarse las señales:

- `SIGBUS` (error de colector o *bus error*, en inglés),
- `SIGSEV` (violación de segmentación o *segmentation violation*, en inglés) o
- `SIGFPE` (excepción de punto flotante o *floating point exception*, en inglés).

La disposición predeterminada para estas señales es la de terminación del proceso y producir un archivo de volcado de memoria (o *core dump file*, en inglés).

Un proceso puede enviar también una señal a otro proceso. Un uso común de este mecanismo es el poner fin a otro proceso enviándole la señal `SIGTERM` o `SIGKILL`.[6] Otra alternativa común es enviarle un comando a un programa en ejecución. Hay dos señales reservadas para este propósito: `SIGUSR1` y `SIGUSR2`. También se usa la señal `SIGHUP` a veces para este propósito, generalmente para despertar a un programa ocioso o forzarlo a releer sus archivos de configuración.

Se puede usar la función `sigaction()` para establecer una disposición. El primer parámetro es el número de señal. Los dos parámetros siguientes son punteros a estructuras de `sigaction`: la primera de éstas contiene la disposición deseada para ese número de señal.

6. ¿Cuál es la diferencia? La señal `SIGTERM` le pide a un proceso que finalice, el proceso puede ignorar el pedido enmascarando o ignorando la señal. La señal `SIGKILL` siempre finaliza el proceso inmediatamente porque el proceso no puede enmascarar o ignorar a `SIGKILL`.

mientras que la segunda recibe la disposición previa. El campo más importante en la primera o segunda estructura de `sigaction()` es `sa_handler`. Puede tomar uno de tres valores:

- `SIG_DFL`, el cual especifica la disposición predeterminada para la señal.
- `SIG_IGN`, especifica que la señal debería ser ignorada.
- Un puntero a la función del manejador de señales. La función debería tomar un parámetro, el número de señal y devolver `void`.

Dado que las señales son asíncronas, el programa principal puede estar en un estado muy frágil cuando se procesa una señal y, por consiguiente, cuando ejecuta una función de manejador de función. Por lo tanto, se debería evitar realizar operaciones de entrada o salida o invocar a la mayoría de las funciones de biblioteca y de sistema desde los manejadores de señales.

Un manejador de señales debería realizar el mínimo de trabajo necesario para responder a una señal y luego, devolver el control al programa principal (o finalizar el programa). En la mayoría de los casos, esto simplemente consiste en grabar el hecho de que ocurrió una señal. Después, el programa principal verifica periódicamente si ha ocurrido una señal y reacciona en consecuencia.

Es posible que un manejador de señales sea interrumpido por el procesamiento de otra señal. Si bien esto puede sonar como de difícil ocurrencia, si llega a ocurrir, será muy difícil de diagnosticar y depurar el problema. (Esto es un ejemplo de condición de carrera). Por lo tanto, se debería ser muy cuidadoso con lo que un programa hace dentro de un manejador de señales.

Aún el asignar un valor a una variable global puede ser peligroso porque puede llevarse a cabo en dos o más instrucciones de máquina y la segunda señal puede ocurrir entre ellas, dejando corrupta a la variable. Si se usa una variable global para «señalizar una señal» desde una función manejadora de señales, debería ser del tipo especial `sig_atomic_t`. Linux garantiza que las asignaciones a variables de este tipo se hacen en una única instrucción, y por lo tanto, no se pueden interrumpir por la mitad. En Linux, el tipo `sig_atomic_t` es, en realidad, un entero tipo `int`. De hecho, si se quiere escribir un programa que sea portable a cualquier sistema Unix estándar, se debería usar el tipo `sig_atomic_t` para estas variables globales.

5.8.2 Mutex

Piense en un mutex como en la llave de un sanitario. Cualquiera que entre ahí primero entra al sanitario y tranca la puerta. Si alguien más intenta entrar al sanitario mientras que está ocupado, esa persona encontrará la puerta con llave y se verá forzado a esperar afuera hasta que el ocupante salga.

Para crear un mutex, cree una variable de tipo `pthread_mutex_t` y pásele un puntero a `pthread_mutex_init`. El segundo argumento a `pthread_mutex_init` es un puntero a un objeto de atributo del mutex, el cual especifica atributos del mutex. Como con `pthread_create`, si el puntero de atributos es nulo, se asumen los atributos predeterminados. La variable mutex debería ser inicializada sólo una vez. Este fragmento de código demuestra la declaración e inicialización de una variable mutex:

```
pthread_mutex_t mi_mutex;
```

```
pthread_mutex_init (&mi_mutex, NULL);
```

Otra forma simple de crear un mutex con los atributos predeterminados es inicializándolo con el valor especial `PTHREAD_MUTEX_INITIALIZER`. No es necesaria una llamada adicional a `pthread_mutex_init`. Esto es particularmente conveniente para variables globales (y en C++, miembros de datos estáticos). El fragmento previo de código podría haber sido escrito de forma equivalente así:

```
pthread_mutex_t mi_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Un hilo puede intentar trabar a un mutex llamándolo a través de `pthread_mutex_lock`. Si no está ocupado, lo traba y la función devuelve inmediatamente. Si el mutex ya estaba trabado por otro hilo, `pthread_mutex_lock` bloquea la ejecución y retorna sólo eventualmente cuando el mutex haya sido liberado por el otro hilo. Puede haber más de un hilo bloqueado esperando a un mutex trabado. Cuando el mutex es liberado, sólo uno de los hilos bloqueados será desbloqueado (elegido impredeciblemente) y se le permitirá adquirir el mutex, los otros hilos permanecerán bloqueados.

Una llamada a `pthread_mutex_unlock` destraba al mutex. Esta función debería ser llamada siempre desde el mismo hilo que trabó al mutex.

5.8.2.1 Mutex no bloqueante

Ocasionalmente, es útil comprobar si un mutex está ocupado sin quedarse bloqueado. Por ejemplo, un hilo podría necesitar obtener un mutex, pero puede tener otro trabajo que realizar en lugar de quedarse bloqueado si el mutex ya está ocupado por otro hilo. Dado que `pthread_mutex_lock` no retornará hasta que el mutex esté destrabado, es necesaria alguna otra función.

En los sistemas Linux, contamos con `pthread_mutex_trylock` para este propósito. Si se invoca a `pthread_mutex_trylock` sobre un mutex que está desocupado se lo obtendrá y trabará tal como si se hubiera invocado a `pthread_mutex_lock`, y la llamada a `pthread_mutex_trylock` devolverá cero. Sin embargo, si el mutex ya estuviera trabado por otro hilo, la llamada a `pthread_mutex_trylock` no provocará un bloqueo, sino que retornará inmediatamente devolviendo el código de error `EBUSY`. La traba sobre el mutex que mantiene el otro hilo no se ve afectada. Se podrá intentar trabar al mutex más tarde.

5.8.3 Semáforos para hilos

Como el lenguaje C no tiene un tipo «semáforo» predefinido, si queremos usar semáforos tenemos que crearlos mediante una llamada al sistema. En los sistemas GNU/Linux, tenemos dos implementaciones ligeramente diferentes de semáforos. Por una parte, tenemos a la implementación de semáforos del estándar POSIX que se usa entre hilos. Por otra, la otra implementación que se usa para la comunicación entre procesos y que proviene del Unix tradicional.

```
#include <semaphore.h>
sem_t misemaforo;
...
sem_init(&misemaforo, 0, 0);
```

Para utilizar la implementación de semáforos para hilos, hay que incluir al archivo de cabecera `semaphore.h` y representarlos mediante la variable `sem_t`. Antes de usarlos, se los deben inicializar utilizando la función `sem_init` pasando un puntero a la variable `sem_t`. El segundo parámetro de la función debería ser cero y el tercer parámetro es el valor inicial del semáforo. Si el semáforo, ya no se usa hay que liberarlo mediante la llamada a `sem_destroy`.

Para esperar al semáforo se utiliza la función `sem_wait` y para incrementarlo la función `sem_post`. También existe una función de espera no bloqueante `sem_trywait` similar a la `pthread_mutex_trylock` en la que, si la espera hubiera provocado un bloqueo porque el valor del semáforo era cero, la función retorna inmediatamente con el valor de error `EAGAIN` en lugar de provocar un bloqueo.

También en los sistemas GNU/Linux una función para obtener el valor actual de un semáforo: `sem_getvalue`, la cual almacena el valor en la variable tipo `int` apuntada por su segundo argumento. No se debería utilizar el valor del semáforo así obtenido de esta función para tomar la decisión de incrementar o esperar al semáforo, dado que esto podría provocar una condición de carrera, porque otro hilo podría haber cambiado el valor del semáforo entre la llamada a `sem_getvalue` y la llamada a otra función de semáforo. Se deberían utilizar, en su lugar, las funciones atómicas `post` y `wait`.

5.8.4 Semáforos para procesos

Las llamadas `semget()` y `semctl()` reservan y liberan semáforos, y son análogas a `shmget()` y `shmctl()` para la memoria compartida. Para utilizarlos, hay que incluir al archivo de cabecera `<sys/sem.h>`, el cual provee la interfaz para los tradicionales semáforos XSI (originalmente de Unix System V). No son parte del estándar POSIX y ya no se los debería usar, en un futuro quedarán obsoletos y, por este motivo, no se recomienda su uso. Se caracterizan porque son objetos que residen en espacio de memoria del núcleo. Entonces, Se caracterizan porque son objetos que residen en espacio de memoria del núcleo. Entonces, como cada operación con ellos involucra una permutación a modo núcleo, esto los vuelve lentos.

Estos semáforos continúan existiendo aún después de que hayan finalizado todos los procesos que los usan. El último proceso en utilizarlos debería liberarlo explícitamente para que se borre y asegurarse de no sobrecargar al sistema operativo.

5.8.5 Variables de condición

Una variable de condición permite implementar una condición bajo la cual ejecuta un hilo e, inversamente, la condición bajo la cual se bloquea el hilo. Siempre que todo hilo que potencialmente cambie el sentido de la condición use la variable de condición apropiadamente, Linux garantiza que los hilos bloqueados en la condición serán desbloqueados cuando cambie la condición.

Como con un semáforo, un hilo puede *esperar* sobre una variable de condición. Si un hilo A espera sobre una variable de condición, está bloqueado hasta que algún otro hilo, el hilo B, señalice la misma variable de condición. A diferencia de un semáforo, una variable de condición no tiene contador o memoria; el hilo A debe esperar en la variable de condición antes de que el hilo B la señalice. Si el hilo B señaliza la variable de condición antes de que A la espere, la señal se pierde, y el hilo A se bloquea hasta que algún otro la señalice otra vez.

Una variable de condición está representada mediante una instancia de `pthread_cond_t`. Cada variable de condición debe usarse en conjunto con un mutex para prevenir ciertas condiciones de carrera. Estas son las funciones que manipulan las variables de condición:

- `pthread_cond_init` inicializa la variable de condición. El primer argumento es un puntero a una instancia de `pthread_cond_t`. El segundo argumento, un puntero a un objeto de atributo de la variable de condición, es ignorado en los sistemas GNU/Linux. El mutex debe ser inicializado de forma separada.
- `pthread_cond_signal` señaliza una variable de condición. Un único hilo que esté bloqueado en la variable de condición será desbloqueado. Si ningún otro hilo está bloqueado en la variable de condición, la señal es ignorada. El argumento es un puntero a la instancia `pthread_cond_t`. Una llamada similar, `pthread_cond_broadcast`, desbloquea a todos los hilos que están bloqueados sobre la variable de condición, en lugar de sólo uno.
- `pthread_cond_wait` bloquea al hilo llamador de la misma hasta que la variable de condición sea señalizada. El argumento es un puntero a la instancia `pthread_cond_t`. El segundo argumento es un puntero a la intancia del mutex declarado en `pthread_mutex_t`. Cuando se llama a `pthread_cond_wait`, el mutex debe ya estar trabado por el hilo llamador. Esa función atómicamente destraba el mutex y se bloquea sobre la variable de condición. Cuando la variable de condición es señalizada y el hilo llamador se desbloquea, `pthread_cond_wait` automáticamente readquiere la traba sobre el mutex.

La condición protegida por una variable de condición puede ser arbitrariamente compleja.

5.8.6 Memoria compartida

La memoria compartida es una técnica mediante la cual más de un proceso accede a un mismo segmento de memoria. Mediante llamadas a sistema específicas, se solicita al núcleo que reserve, libere, establezca permisos, lea y escriba datos en esa región de la memoria.

El sistema operativo no obtiene la memoria necesaria para esta operación del espacio de memoria que ya tiene asignado el proceso, porque ésta es privada, sino que la obtiene del espacio de memoria libre que tenga el sistema y se la anexa al proceso que la solicita, asignándole también un nombre a esta región (elegido también por el proceso) para poder referirse a ella posteriormente. A esta operación de anexado se la denomina **mapear** (*mapping*, en inglés). El segmento compartido de memoria se asigna a direcciones locales, en espacios de direcciones propios de cada proceso, de manera que quedan correlacionadas byte a byte la región de memoria anexada al proceso y la región de memoria que reservó el núcleo del sistema operativo, como se ilustra en la figura 5.12.

La norma POSIX establece las siguientes llamadas a sistema para crear, sincronizar y deshacer segmentos de memoria compartidos:

- `shm_open()`: Crea una región nueva de memoria compartida o se vincula con una región existente mediante su nombre. Esta llamada devuelve un descriptor de archivo.

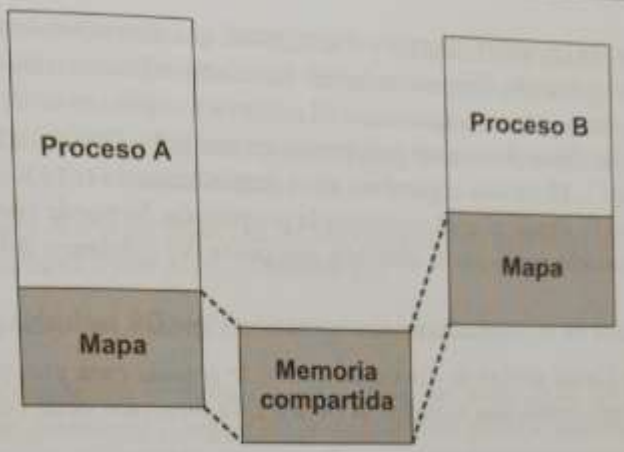


Fig. 5.12 – Memoria compartida entre procesos.

- `shm_unlink()`: Elimina una región de memoria compartida apuntada mediante un descriptor de archivo (el que devolvió `shm_open()`). La región no es eliminada por completo hasta que no hayan finalizado todos los procesos que la acceden.

Esta técnica también permite leer un archivo cargándolo en el área de memoria compartida, permitiendo así no sólo que el proceso tenga un acceso más rápido al contenido del mismo sino también que diferentes procesos se comuniquen entre sí a través de este archivo compartido.

El sistema operativo divide el archivo en trozos del tamaño de una página y luego, los copia en páginas de la memoria virtual de manera que puedan estar disponibles en el espacio de direcciones de un proceso. Así, el proceso puede leer el contenido del archivo con accesos comunes a memoria. También puede modificar el contenido del archivo escribiendo en la memoria. Esto permite el acceso rápido a archivos.

Se puede pensar en la memoria compartida como el asignar un *buffer* para contener el contenido completo del archivo y luego leer el archivo desde el *buffer* y, si el *buffer* fue modificado, escribirlo de vuelta al archivo. El sistema operativo maneja por sí mismo estas operaciones de lectura y escritura del archivo.

5.8.6.1 Proyectar un archivo compartido

Para proyectar a la memoria de un proceso un archivo para compartir se utiliza la llamada `mmap()` (de *Memory MAPped*, en inglés):

```
#include <sys/mman.h
void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
```

El primer argumento de la llamada es la dirección (`addr`) a la cual le gustaría que Linux proyectara el archivo dentro del espacio de direcciones de su proceso; el valor `NULL` le permite a Linux elegir una dirección disponible de comienzo. El segundo argumento es la longitud (`len`) del mapa en bytes. El tercer argumento especifica la protección (`prot`) sobre el rango de direcciones mapeadas. La protección consiste en una operación lógica OR inclusiva entre

los permisos PROT_READ, PROT_WRITE y PROT_EXEC que corresponden a los permisos de lectura, escritura y ejecución, respectivamente. El cuarto argumento es un valor indicador (flags) que especifica opciones adicionales. El quinto argumento es un descriptor de archivo abierto (fildes) –el del archivo a ser proyectado en memoria. Puede ser también el que nos devolvió shm_open(). El último argumento es el desplazamiento (off) –contando desde el comienzo del archivo– desde el cual comenzar la proyección. Se puede proyectar todo o parte del archivo a la memoria mediante la elección apropiada del comienzo del desplazamiento y la longitud.

El valor indicador es el resultado de una operación lógia OR inclusiva entre:

- MAP_FIXED: Linux utiliza la dirección que se le solicita para proyectar el archivo en lugar de tratarla como una sugerencia (*hint*). Esta dirección debe estar alineada a una página.
- MAP_PRIVATE: Las escrituras al rango de memoria no se deberían escribir inmediatamente al archivo adjuntado sino a una copia privada del mismo. Ningún otro proceso ve estas escrituras. Este modo no debería usarse con MAP_SHARED.
- MAP_SHARED: Las escrituras se reflejan inmediatamente en el archivo proyectado en lugar de ser almacenadas a un *buffer*. Este modo es el que se debería utilizar para IPC con memoria mapeada. Este modo no se debería usar con MAP_PRIVATE.

Si la llamada es exitosa, devuelve un puntero al comienzo de la memoria. Si no, devuelve el código de error MAP_FAILED.

Cuando se ha finalizado de utilizar la proyección en memoria, hay que liberarla mediante munmap(), pasándole la dirección de comienzo y la longitud de la región de memoria mapeada:

```
#include <sys/mman.h>
int munmap(void *addr, size_t len);
```

Linux automáticamente libera las regiones de memoria mapeadas cuando finaliza el proceso.

5.8.6.2 Acceso compartido a un archivo

Diferentes procesos se pueden comunicar utilizando regiones de memoria a las que se les ha proyectado el mismo archivo. Mediante el bit MAP_SHARED, se especifica que cualquier escritura a estas regiones sea transferida inmediatamente al archivo de origen y hecha visible a otros procesos. Si no se especifica este modo, Linux puede almacenar en *buffer* las escrituras antes de transferirlas al archivo.

Alternativamente, se puede forzar a Linux para que incorpore escrituras almacenadas en *buffer* al archivo en disco mediante el llamado a msync(). Sus primeros dos parámetros indican una región de memoria mapeada, como con munmap(). El tercer parámetro puede tomar alguno de estos valores:

- MS_ASYNC: Se planifica la actualización pero no necesariamente ejecuta antes de que retorne la llamada.

- `MS_SYNC`: La actualización es inmediata; la llamada a `msync()` bloquea al hilo hasta que la llamada haya finalizado. No se deberían usar MS_SYNC y MS_ASYNC juntas a la vez.
- `MS_INVALIDATE`: Todas las otras proyecciones de archivo se invalidan de manera que puedan ver los valores actualizados.

Por ejemplo, para descargar (*flush*) un archivo compartido mapeado en la dirección `mem_addr` de longitud `mem_length` bytes haga la llamada así:

```
msync (mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);
```

Como con los segmentos de memoria compartida, los usuarios de las regiones de memoria mapeada deben establecer y seguir un protocolo para evitar condiciones de carrera. Por ejemplo, se puede usar un semáforo para prevenir que más de un proceso acceda a la memoria mapeada a la vez.

5.9 Sincronización en Linux

Los sistemas Linux contemplan los mecanismos de sincronización, concurrencia y exclusión mutua estudiados, no sólo en el núcleo mismo sino también en bibliotecas para que puedan usarse desde los lenguajes de programación; tanto si ejecuta sobre un único procesador o sobre varios.

El núcleo es muy dinámico en su ejecución, por ello, está expuesto a todos los problemas de sincronización vistos. En los primeros días de Linux, se habilitaban las interrupciones como método para lograr la exclusión mutua. Esto se lleva a cabo aún actualmente mediante la llamada en a `local_irq_save()` y con `local_irq_restore()` para habilitarlas nuevamente; aunque no es recomendable, como ya sabemos, pero se hace de forma muy cuidadosa.

5.9.1 Operaciones atómicas

La forma más simple de sincronización en el núcleo de Linux la otorgan las operaciones atómicas. Los operadores atómicos son ideales para situaciones en las cuales los datos a proteger son simples, tales como una variable contadora. Como el lenguaje C no puede garantizar operaciones atómicas, se debe confiar la tarea al hardware y, dado que las arquitecturas difieren mucho, nos encontraremos con distintas implementaciones, las cuales están desarrolladas en los archivos de cabecera, encapsulándolas en una única llamada para ocultar la diversidad (API). En el archivo de cabecera `atomic.h` contamos con apoyo para realizar operaciones atómicas; para declarar a una variable como atómica, hay que hacerlo especificándola del tipo `atomic_t`. Por ejemplo, la siguiente sentencia en C:

```
atomic_t contador = ATOMIC_INIT(0);
```

declara a la variable `contador` como entera, atómica y con valor inicial cero.

Además, hay muchas operaciones que se pueden realizar de manera atómica sobre la variable declarada de esta manera: `atomic_read()`, `atomic_set()`, `atomic_add()`, `atomic_inc()`, `atomic_sub()`, `atomic_dec()`. También podemos efectuar más de una

operación de forma atómica. Por ejemplo, la función `atomic_add_negative()` nos permite sumar a la variable (declarada `atomic_t`) y comprobar si es negativa, en caso de serlo, devolverá el valor preestablecido para «verdadero»; todo esto en una única operación indivisible. De manera similar contamos en la cabecera `atomic.h` con operaciones como: `atomic_sub_and_test()`, `atomic_dec_and_test()`, `atomic_inc_and_test()`, `atomic_add_return()` y `atomic_sub_return()`. Y para realizar operaciones atómicas con máscaras de bits: `atomic_clear_mask()` y `atomic_set_mask()`.

5.9.2 Spinlock

Para algo un poco más complicado que las operaciones atómicas está el *spinlock*, que es la forma más común en el núcleo Linux. En Linux, tenemos dos tipos de *spinlock*: los propiamente dichos (o *full spinlock*, en inglés) y los de lectores/escritores.

5.9.2.1 Full spinlock

Para ellos, existe la declaración de tipo `spinlock_t` disponible en el archivo de cabecera `spinlock_types.h`

```
spinlock_t traba=SPIN_LOCK_UNLOCKED;
```

que también se puede haber hecho a través de la llamada a función

```
spin_lock_init(&traba);
```

Las operaciones están definidas en el archivo de cabecera `spinlock.h`, por ejemplo: `spin_lock()`, `spin_unlock()`, `spin_lock_irqsave()`, `spin_unlock_irqrestore()`. Para las mitades inferiores (o *bottom halves*, en inglés), están las funciones `spin_lock_bh()`, `spin_unlock_bh()`.

Los *spinlocks* en Linux no son recursivos, como suelen ser en otros sistemas operativos. Muchos consideran ésta una sana decisión de diseño ya que si lo fueran favorecerían un código defectuoso pero esto también implica que se debe tener mucho cuidado de no intentar adquirir un *spinlock* que ya tenemos porque esto provocaría un interbloqueo.

Se pueden encontrar más detalles acerca de cómo se implementan los *spinlocks* en el núcleo Linux en el archivo `spinlocks.txt` que está en el subdirectorio `Documentation` perteneciente al paquete de distribución del código fuente.

5.9.2.2 Lectores y escritores

Además de los clásicos *spinlocks*, el núcleo Linux dispone de una variante para los casos que siguen el modelo de lectores y escritores, y hay definiciones y funciones específicas. A partir de la declaración,

```
rwlock_t lectorescritor;
```

pueden utilizarse sobre esta variable las funciones:

`rwlock_init()`, `write_lock()`, `write_unlock()`, `read_lock()` y `read_unlock()`.

Esta es una apreciable optimización, según Love (2002), no obstante, una vez que se obtuvo la traba para leer no se puede cambiarla fácilmente para escribir, porque se produciría un interbloqueo. Por lo tanto, si existiera la posibilidad de necesitar la traba para escribir, es preferible solicitarla inicialmente de este modo. Sino, habría que liberar la traba de lectura y volver a solicitarla para escribir.

5.9.3 Mutex

El *mutex* existe en el núcleo Linux como una forma de llevar a cabo el comportamiento de un semáforo, opina M. T. Jones (2007a). El *mutex* del núcleo está implementado sobre la API de las operaciones atómicas, de esta manera:

```
struct mutex {

     atomic_t count;
     spinlock_t wait_lock;
     struct list_head wait_list;

};
```

En el archivo de cabecera `mutex.h`, está el código para crear e inicializar un mutex

```
DEFINE_MUTEX(mimutex);
```

Sólo una tarea por vez puede tener el *mutex* y sólo esta tarea puede liberarlo. Tampoco se puede trabar ni destrabar recursivamente el *mutex*.

En el archivo de cabecera, también están las operaciones que se pueden realizar con el *mutex*: `mutex_trylock()`, `mutex_lock()`, `mutex_lock_interruptible()`, `mutex_unlock()` y `mutex_is_locked()`.

Como con los *spinlocks*, se pueden encontrar más detalles acerca de cómo se implementan los *mutex* en el núcleo Linux en el archivo `mutex-design.txt` que está en el subdirectorio `Documentation` perteneciente al paquete de distribución del código fuente.

5.10 Sincronización en Windows

Windows, como buen ejemplo de sistema operativo moderno, cuida mucho los aspectos de la sincronización. Según M. E. Russinovich y D. A. Solomon (2004), el núcleo y el ejecutivo (originalmente, *executive*) de Windows no se apoyan en mecanismos de sincronización que puedan provocar un fallo de página o una operación de planificación para sincronizar el acceso a estructuras de datos cuando el IRQL es de nivel DPC/Dispatch o superior. Para lograr exclusión mutua y proteger sus estructuras de datos globales, se usan distintos mecanismos de sincronización: para niveles altos de IRQL, por una parte, y para niveles bajos de IRQL, por otra.

5.10.1 Sincronización con IRQL alto

En el caso en el que el IRQL es alto (nivel del IRQL mayor o igual al IRQL de DPC/Dispatch), el núcleo tiene que proteger sus estructuras críticas de datos globales, tales como la base

de datos del activador (*dispatcher*, en inglés) del núcleo o su cola DPC. La mayor área de preocupación son las interrupciones. Windows no se apoya en la deshabilitación de las interrupciones como método para lograr exclusión mutua. Antes de utilizar un recurso global, el núcleo temporariamente enmascara esas interrupciones cuyos manejadores de interrupciones también usan ese recurso. Lo hace elevando el IRQL del procesador al nivel más alto usado por cualquier fuente de interrupción potencial que acceda al dato global. Esta estrategia es buena en un sistema con un único procesador pero no con varios, como ya sabemos. El elevar el IRQL de un procesador no previene que ocurra una interrupción en otro. El núcleo necesita garantizar acceso exclusivo entre varios procesadores.

La forma más simple de mecanismos de sincronización se apoya en el soporte que brinda el hardware de sistemas multiprocesadores para la manipulación de valores enteros y realización de comparaciones. Incluyen funciones tales como `InterlockedIncrement`, `InterlockedDecrement`, `InterlockedExchange` e `InterlockedCompareExchange`. La función `InterlockedDecrement`, por ejemplo, utiliza el prefijo de instrucción `lock` de la arquitectura x86 (por ejemplo, `lock add`) para trabar el colector (o bus) del multiprocesador durante la operación de sustracción, de manera que otro procesador que también está modificando la posición de memoria que está siendo decrementada, no será capaz de modificarla entre la lectura de decremento del valor original y la escritura del valor decrementado. Esta forma de sincronización básica es usada por el núcleo y los manejadores.

5.10.1.1 Spinlocks

El *spinlock* es el mecanismo que utiliza el núcleo para lograr exclusión mutua con varios procesadores. Es una primitiva trabante asociada con una estructura de datos global, como se muestra en la figura 5.13.

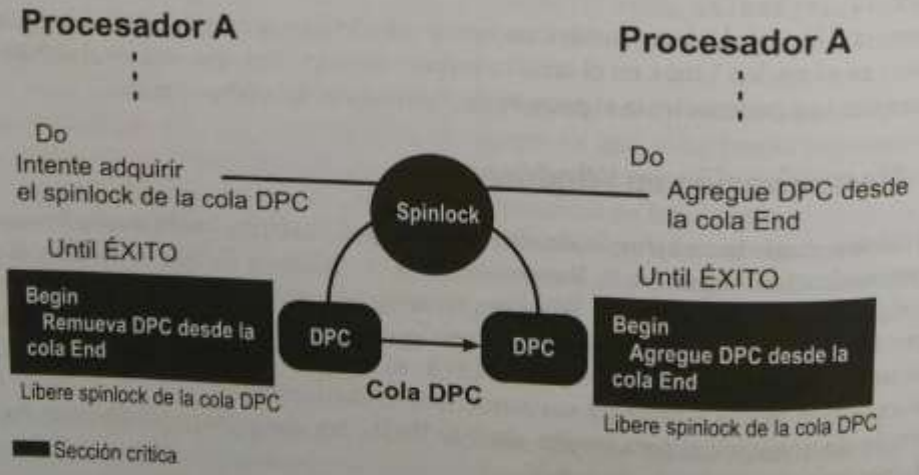


Fig. 5.13 – Spinlock.

Antes de entrar en cualquiera de las secciones críticas mostradas en la figura, el núcleo debe adquirir el *spinlock* asociado con la cola DPC protegida. Si el *spinlock* no está libre, el núcleo sigue intentando adquirir la traba hasta que lo logre.

Con cola: en ciertas circunstancias, se usa un tipo especial de *spinlock* que escala mejor con multiprocesadores y utiliza una cola (originalmente, *queued spinlock*, en inglés). Cuando un procesador quiere adquirirlo y se encuentra en uso, coloca su identificador en una cola asociada con el *spinlock*. Cuando el procesador que estaba utilizando el *spinlock* lo libera, se lo cede al primer procesador de la cola.

5.10.2 Sincronización con IRQL bajo

El software ejecutivo que está fuera del núcleo también necesita sincronizar el acceso a estructuras de datos globales presentes en un ambiente de varios procesadores, afirma M. E. Russinovich y D. A. Solomon (2004). Por ejemplo, el administrador de la memoria tiene sólo una base de datos de marcos de páginas a la cual accede como una estructura de datos global, y los manejadores de dispositivos necesitan asegurarse de que pueden tener acceso exclusivo a sus dispositivos. Mediante llamadas a funciones del núcleo, el ejecutivo puede crear un *spinlock*, adquirirlo y liberarlo.

Sin embargo, los *spinlocks* sólo alcanzan a cubrir parcialmente las necesidades de mecanismos de sincronización que tiene el ejecutivo. Debido a que el esperar a un spinlock literalmente atasca un procesador, los spinlocks se pueden usar sólo bajo las estrictamente limitadas circunstancias:

- El recurso protegido debe accederse de forma rápida y sin interacciones complicadas con otro código.
- El código de la sección crítica no puede ser desalojado de memoria, no puede hacer referencias a datos paginables, no puede llamar a procedimientos externos (incluyendo a servicios de sistema) y no puede generar interrupciones o excepciones.

Estas restricciones son limitantes y no se pueden lograr bajo todas las circunstancias. Más aún, el ejecutivo necesita realizar otros tipos de sincronizaciones además de la exclusión mutua, y debe además proveer mecanismos de sincronización al modo usuario.

Hay varios mecanismos de sincronización adicionales para usar cuando no son viables los spinlocks:

- Objetos Activadores del Núcleo (*Kernel Dispatcher Objects*, en inglés): Estos objetos pueden ser: objetos temporizadores, de eventos, de semáforos, de mutex y de hilos. Los manejadores pueden usarlos como mecanismos de sincronización dentro de un contexto de hilos no arbitrario mientras se ejecuta a un nivel IRQL bajo.
- Mutex rápidos y custodiados (*Fast Mutexes and Guarded Mutexes*, en inglés): Los manejadores pueden usar mutex rápidos si requieren una forma de exclusión mutua de baja sobrecarga para código que ejecuta a nivel IRQL menor o igual a `APC_LEVEL`. El mutex custodiado (*guarded mutex*) es un reemplazo de los mutex rápidos pero proveen mejor rendimiento, porque ejecutan más rápidamente. También ejecuta con IRQL menor o igual a `APC_LEVEL`, pero con todos los APC deshabilitados.

- Pushlocks: Son mecanismos de sincronización para la protección de datos paginables en el modo núcleo, según M. Russinovich y D. Solomon (2001). Requieren sólo el tamaño de un puntero (4 bytes en 32 bits y 8 bytes en 64 bits) en almacenamiento. La adquisición y liberación se llevan a cabo sin el uso de *spinlocks* en modo núcleo si no hay competencia en el *pushlock*.
- Recursos del ejecutivo (*Executive Resources*, en inglés)

5.10.3 Llamada a Procedimiento Local

Las aplicaciones de usuario y los subsistemas de entorno de ejecución no se comunican directamente en Windows, sino que pasan a través del ejecutivo del sistema operativo por criterios de protección y de compartición de recursos. El ejecutivo se encarga de optimizar estas comunicaciones mediante una utilidad de Llamada a Procedimiento Local (LPC, *Local Procedure Call*, por sus siglas en inglés), algo muy similar al concepto de llamada a procedimiento remoto, pero optimizada para procesos que comparten memoria. De esta forma, cuando una aplicación se quiere comunicar con otra, sus mensajes pasan a través del subsistema de su entorno hasta el gestor de LPC, como se aprecia en la figura 5.14.

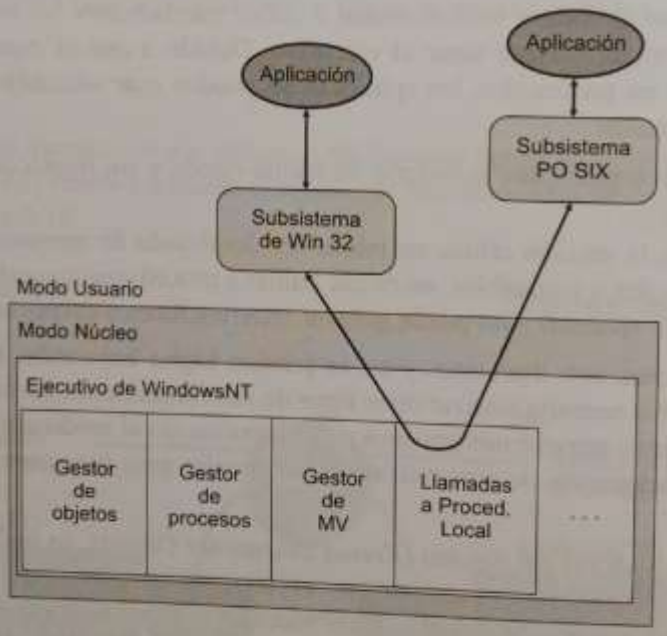


Fig. 5.14 – Comunicación LPC.

Las aplicaciones no son conscientes de esta forma de comunicación porque el uso de las LPC está oculto dentro de las DLL (de *Dynamic Link Libraries*, o Bibliotecas de Enlace Dinámico) que proporciona el sistema operativo. Cuando se genera una aplicación que usa un determinado subsistema de ejecución, se enlaza con la DLL de este subsistema, que incluye los mecanismos de comunicación a través de LPC. Estos mecanismos incluyen el

uso de suplentes que se encargan de empaquetar, o desempaquetar, los parámetros y de la comunicación. Esta última se optimiza mediante el uso de memoria compartida.

El sistema operativo Windows NT es un ejemplo de diseño moderno que utiliza modularidad para incrementar la funcionalidad y reducir el tiempo necesario para implementar nuevas características. NT apoya múltiples entornos operativos o subsistemas con los que los programas de aplicación se comunican empleando un sistema de transferencia de mensajes. Los programas de aplicación se pueden considerar como clientes del servidor de subsistemas NT.

El recurso de transferencia de mensajes en Windows NT se llama Recurso de Llamada a Procedimiento Local (LPC, de *Local Procedure Call*, por sus siglas en inglés), y sirve para que se comuniquen entre sí dos procesos que están en la misma máquina. El mecanismo es similar al de llamada a procedimientos remotos estándar de uso general, pero optimizado y específico para NT. Windows NT, al igual que Mach, utiliza un objeto puerto para establecer y mantener una conexión entre dos procesos. Cada cliente que invoca un subsistema necesita un canal de comunicación, mismo que el objeto puerto proporciona y que nunca se hereda. NT maneja dos tipos de puertos, puertos de conexión y puertos de comunicación, que en realidad son iguales pero reciben diferentes nombres según la forma en que se usen. Los puertos de conexión son objetos con nombre, visibles para todos los procesos, y ofrecen a las aplicaciones una forma de establecer un canal de comunicación. La comunicación funciona así:

- El cliente abre un mando (*handle*, en inglés) para el objeto puerto de conexión del subsistema.
- El cliente envía una solicitud de conexión.
- El servidor crea dos puertos de comunicación privados y devuelve el mando de uno de ellos al cliente.
- El cliente y el servidor usan el mando de puerto correspondiente para enviar mensajes o devoluciones de llamada y esperar respuestas.

NT emplea tres tipos de técnicas de transferencia de mensajes por un puerto que el cliente especifica cuando establece el canal. La más sencilla, que se usa si el mensaje es pequeño, consiste en utilizar la cola de mensajes del puerto como almacenamiento intermedio y copiar el mensaje de un proceso al otro. Es posible enviar mensajes de hasta 256 bytes con este método.

Si un cliente necesita enviar un mensaje más grande, lo transfiere a través de un objeto sección (memoria compartida). El cliente tiene que decidir, cuando establece el canal, si necesitará o no enviar un mensaje grande. Si el cliente determina que desea enviar mensajes grandes, pide que se cree un objeto sección. Asimismo, si el servidor decide que las respuestas serán grandes, también creará un objeto sección.

Para usar este objeto, se envía un mensaje pequeño que contiene un puntero e información de tamaño referente al objeto sección. Este método es un poco más complicado que el primero, pero evita el copiado de datos. En ambos casos, se puede utilizar un mecanismo de devolución de llamada si el cliente o el servidor no puede responder de inmediato a una solicitud. El mecanismo de devolución de llamada hace posible el manejo asíncrono de mensajes.

Una de las desventajas del uso de transferencia de mensajes para desempeñar funciones rudimentarias, como la de gráficos, es que el desempeño podría no ser tan bueno como el de un sistema de memoria compartida. Con objeto de mejorar el desempeño, el entorno NT nativo (Win32) utiliza un tercer método de transferencia de mensajes llamado LPC rápida. Un cliente envía una solicitud de conexión al puerto de conexión del servidor e indica que usará LPC rápida. El servidor crea un hilo dedicado para atender las solicitudes, un objeto sección de 64 KB, y un objeto par de sucesos. A partir de ese momento, los mensajes se pasan a través del objeto sección y el objeto par de sucesos se encarga de la sincronización. Esto elimina el copiado de mensajes, el gasto extra de usar el objeto puerto, y el gasto extra que implica determinar cuál hilo cliente llamó, ya que sólo hay un hilo servidor para cada hilo clientes. Además, el núcleo concede al hilo dedicado preferencia de planificación. Desde luego, la desventaja de este método es que consume más recursos que los otros dos. Puesto que la transferencia de mensajes implica cierto gasto extra, NT podría juntar varios mensajes en uno solo para reducir ese gasto extra.

5.10.4 Mailslots

Un *mailslot* es un pseudoarchivo que reside en memoria y ofrece un mecanismo de comunicación interprocesos (IPC) unidireccional. Los procesos pueden ser tanto locales como remotos (red mediante). Conceptualmente, son parecidos a las tuberías con nombre de Windows y a las colas de mensajes de POSIX, pero su uso es más simple que éstas y que los *sockets*, sobretodo tratándose de un número relativamente pequeño de mensajes cortos, por ejemplo: mensajes de cambio de estado o como parte de un protocolo de descubrimiento de pares (*peer-discovery protocol*, en inglés). Los datos en un mensaje a un *mailslot* pueden ser de cualquier forma pero el tamaño máximo de los mensajes que se pueden enviar es de 424 bytes.

Funcionan en el modelo cliente-servidor: un servidor *mailslot* es un proceso que puede crear un *mailslot* y al hacerlo recibe un mando que se usa para leer mensajes del *mailslot*. Sólo el servidor que creó el *mailslot* o los procesos que lo hereden pueden leer los mensajes. El servidor *mailslot* sólo puede crearlos localmente, no puede crear un *mailslot* remoto.

Un cliente *mailslot* es un proceso que puede escribir mensajes en *mailslot* mediante su nombre. El mecanismo permite el envío de mensajes cortos de multidifusión (*broadcast*) para que sea escuchado por todas las computadoras que comparten el mismo dominio[7] de red.

Por esto es que representan a un mecanismo de comunicación unidireccional y se logra tener a múltiples lectores y múltiples escritores.

El *mailslot* sigue un protocolo de Llamada Procedimiento Remoto (o RPC, en inglés) entre todas las computadoras que comparten un dominio en la red. No ofrecen confirmación de que el mensaje haya sido recibido porque se utilizan datagramas. Son una buena opción cuando un proceso cliente tiene que difundir un mensaje a varios procesos servidores.

7. Un dominio –en este contexto– es un grupo de computadoras que comparten un nombre de grupo

5.11 Trabajos prácticos

5.11.1 Tuberías

A continuación, veremos un ejemplo de Carretero Pérez (Carretero Pérez 2001) de ejecución de comandos con tuberías. Podemos colocar el comando:

```
$ ls | wc
```

que genera un listado de archivos y lo comunica a través de un *pipe* con `wc` que es el comando `word count`, para contar palabras, para mayor información coloque el comando:

```
$ man wc
```

El carácter pleca «|» le indica al intérprete `bash` que debe crear una tubería entre los dos procesos. A continuación, se presenta el programa 5.9 de Carretero Pérez que permite la ejecución de este comando, mediante llamadas a sistema.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
void main(void)
{
int fd[2];
pid_t pid;
if (pipe(fd) < 0) {
perror("Error al crear la tuberia");
exit(-1);
}
pid = fork();
switch(pid) {
case -1: /* error */
perror("Error en el fork");
exit(-1);
case 0: /* proceso hijo ejecuta "ls" */
close(fd[0]); /* cierra el pipe de lectura */
close(STDOUT_FILENO); /* cierra la salida estandar */
dup(fd[1]);
close(fd[1]);
execlp("ls","ls",NULL);
perror("execlp");
exit(-1);
default: /* proceso padre ejecuta "wc" */
close(fd[1]); /* cierra el pipe de escritura */
close(STDIN_FILENO); /* cierre la entrada estandar */
dup(fd[0]);
close(fd[0]);
execlp("wc","wc",NULL);
perror("execlp");
}
}
```

Se compila con:

```
$ gcc -o tuberia tuberia.c
```

Y se ejecuta

```
$./tuberia
```

El programa `wc` por defecto devuelve tres valores: el número de `newlines`, el de palabras y el de bytes.

5.11.2 Otro ejemplo:

Se trata simplemente de procesos en concurrencia sobre los 100 primeros caracteres de un archivo (se ejecutarán varios ejemplares del mismo programa). Para resolver el problema de la exclusión mutua sobre el acceso, se coloca un candado (*lock*) bloqueante, utilizamos la función `lockf()`, cuyo primer parámetro es un descriptor de archivo abierto, ya sea para lectura o para lectura/escritura. El siguiente parámetro de operación es alguno de los siguientes:

- F_ULOCK: eliminación de candado
- F_LOCK: candado exclusivo en modo bloqueante
- F_TLOCK: candado exclusivo en modo no bloqueante
- F_TEST: comprobación de existencia de un candado

El tercer parámetro es el tamaño, que permite especificar la extensión del bloqueo. La cual se expresa con respecto a la posición actual (el valor puede ser negativo para bloquear una zona anterior a la posición actual). Un tamaño nulo permite bloquear hasta el final del archivo (cualquiera que pueda ser en el futuro).

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/file.h>
main()
{
int d;
char buffer[100];
d=open("archivo", O_RDWR | O_CREAT,S_IRWXU, 0);
write(d,buffer,100);
if(lockf(d, F_LOCK, 100) == -1)
perror("lockf");
else
printf("Proceso%d: candado colocado\n", getpid());
sleep(5);
printf("Proceso%d: candado eliminado\n", getpid());
lockf(d, F_ULOCK, 100);
}
```

La llamada a `open()` abre el archivo en modo lectura/escritura, si «archivo» no existe lo crea. Luego la llamada a `write()` le escribe 100 caracteres, pero como *buffer* no ha sido inicializado le escribe basura, lo cual no nos importa en este caso. Se compila con:

```
$ gcc -o candado candado.c
```

Y se ejecutan varias copias:

```
$ ./candado & ./candado & ./candado &
```

enviándolas al *background*, veremos que el primer proceso coloca el candado sobre el archivo, espera 5 segundos y luego lo saca, sólo entonces el segundo proceso puede colocar el candado, esperar 5 segundos y luego sacarlo, para que el tercer proceso pueda hacer lo mismo.

5.12 Autoevaluación

1. ¿Cuál no es un mecanismo de sincronización válido entre procesos UNIX?
   a) Cerrojos o candados implementados sobre archivos.
   b) Un *PIPE*.
   c) Instrucciones test-and-set.
   d) Las señales.
2. ¿Qué NO es cierto respecto a las propiedades que se deben cumplir en el problema de la sección crítica?
   a) No deben importar las velocidades relativas de los procesos.
   b) Exclusión mutua.
   c) Inanición
   d) No debe haber espera activa.
3. Se desea usar un semáforo S para asegurar que el inicio de una determinada actividad del proceso P_2 comienza después de que finalice una actividad del proceso P_1. ¿Qué primitiva de semáforo debe usar cada proceso y cuál debe ser el valor inicial del semáforo?
   a) P_1 wait(S) y P_2 signal(S); S=0.
   b) P_1 signal(S) y P_2 wait(S); S=0.
   c) P_1 wait(S) y P_2 signal(S); S=1.
   d) P_1 signal(S) y P_2 wait(S); S=1.
4. Los planificadores *round robin* normalmente mantienen una lista de todos los procesos ejecutables, y cada proceso aparece una y sólo una vez en la lista. ¿Qué sucedería si un proceso ocurriera dos veces en la lista? ¿Puede usted pensar en alguna razón para permitir esto?
5. Describa las diferencias entre la planificación a corto plazo, mediano plazo y largo plazo.
6. ¿Qué es falso acerca de los *PIPES* de UNIX?
   a) Permiten comunicar a un proceso con el Sistema Operativo.
   b) Se puede utilizar para sincronizar procesos.
   c) Se puede utilizar para comunicar procesos.
   d) Tiene un *buffer* para amortiguar las velocidades en la comunicación entre procesos.
7. ¿Qué es cierto acerca del algoritmo de la panadería de Lamport?

a) Resuelve el problema de los lectores-escritores.
b) Resuelve el problema del productor-consumidor.
c) Resuelve el problema de la sección crítica para 2 procesos.
d) Resuelve el problema de la sección crítica para n procesos.

8. Sea un programa concurrente con tres procesos iguales cuyo código consiste tan solo en incrementar en uno una variable V compartida entre ellos. ¿Cuál de las siguientes opciones es correcta respecto al posible valor resultante de V después de la ejecución concurrente de los tres procesos si V vale 0 inicialmente?

   a) V tiene valor 1.5.
   b) V tiene valor 4.
   c) V tiene valor 0.
   d) V tiene valor 2.

9. Sugiera tres aplicaciones que podrían utilizar el modelo productor/consumidor.
10. Sugiera tres aplicaciones que podrían utilizar el modelo escritor/lectores.
11. Supongamos que estamos en una discoteca y resulta que no se puede utilizar el baño de chicas y todos tienen que compartir el de chicos. Se pretende establecer un protocolo de entrada al baño usando semáforos en el que se cumplan las siguientes restricciones:

    a) chicas sólo puede haber una a la vez en el baño.
    b) chicos puede haber más de uno, pero con un máximo de cinco.
    c) (versión machista) los chicos tienen preferencia sobre las chicas. Esto quiere decir que si una chica está esperando y llega un chico, éste puede pasar. Incluso si no puede entrar en el baño porque hay ya cinco chicos, pasará antes que la chica cuando salga algún chico.
    d) (versión feminista) las chicas tienen preferencia sobre los chicos. Esto quiere decir que si un chico está esperando y llega una chica, ésta debe pasar antes. Incluso si no puede entrar en el baño porque hay una chica, pasará antes que el chico cuando salga la chica.

Almacenamiento

6

Tradicionalmente, los sistemas operativos han manejado un único tipo de sistema de archivos, el propio. Pero con el evolucionar del *hardware*, por una parte, el de los sistemas operativos por otra y la necesidad de interoperar entre unos y otros, se fue haciendo cada vez más necesario que los sistemas operativos dieran soporte a más de un tipo de sistema de archivos, además del propio. Esto provocó que el *software* de gestión del almacenamiento (*storage management*), por usar una expresión más abarcadora, se volviera más complejo y el código del sistema de archivos, que alguna vez estuvo íntimamente ligado en el código del núcleo, ha debido aislarse en módulos y la interacción con él es ahora a través de interfaces estándares, con el objeto de facilitar el intercambio de datos entre arquitecturas y diversidad de *hardware*. A estas nuevas capas que operan como interfaces se les suele llamar **sistema de archivos virtual**.

La gestión de almacenamiento define la forma en la que un sistema operativo interactúa con los dispositivos y medios de almacenamiento no volátiles. El término «almacenamiento» involucra a muchos dispositivos diferentes, que incluyen a los dispositivos de cinta, medios ópticos, discos compactos o CD, diskettes, discos rígidos locales, unidades de memoria *flash* y redes de áreas de almacenamientos (*Storage Area Networks* o SANs). De manera que la gestión del almacenamiento de los sistemas operativos modernos involucra más aspectos que la mera gestión de archivos de los sistemas de archivos propiamente dichos.

Para la mayoría de los usuarios, el sistema de archivos es el aspecto más visible de un sistema operativo, pues propociona el mecanismo para el almacenamiento y acceso tanto de datos como de programas del sistema operativo. Algunos sistemas operativos han puesto un mayor énfasis en esta característica, por ejemplo en UNIX y sus semejantes, los dispositivos de entrada y salida pueden modelarse como archivos, como también se puede hacer con los procesos. El sistema operativo distribuido Plan 9 de Bell Labs logra representar todas las interfaces del sistema, incluyendo aquellas requeridas para redes y la interfaz con el usuario, a través del sistema de archivos.

6.1 Conceptos

Sabemos que un archivo es un conjunto de documentos, pero en Ciencias de la Computación

> se le llama archivo a un conjunto de información relacionada, definida por su creador y almacenada en la computadora.

Estos archivos pueden o no tener una estructura o formato desde el punto de vista del usuario o la aplicación, pero para los sistemas operativos modernos simplemente son tratados como una tira de *bytes*, de hecho, el sistema operativo se abstrae de la estructura física del medio de almacenamiento para manejarse con la estructura lógica. Pero los archivos en un medio de almacenamiento no se nos presentan de manera caótica,

> la forma que tiene una computadora de organizar, nombrar, almacenar y manipular a los archivos es lo que denominamos sistema de archivos (*filesystem*).

Y por tratarse de una parte del sistema operativo le podemos decir subsistema de archivos. El subsistema de archivos o sistema de gestión de archivos desde el punto de vista del sistema operativo es el sistema de *software* que proporciona a los usuarios y aplicaciones los servicios relativos al uso de archivos. Según Grosshans (1986) los objetivos del sistema de archivos son, por una parte, cumplir con la gestión de datos y solicitudes del usuario, por otra, garantizar la integridad del contenido del contenido de los archivos, también dar soporte de entrada y salida para los distintos dispositivos, además brindar un conjunto de rutinas estándar de interfaces de entrada y salida y finalmente optimizar el rendimiento tanto a nivel de productividad como de tiempo de respuesta.

Desde el punto de vista del usuario, éste debe poder crear, borrar y modificar sus archivos; controlar su acceso y el de los otros usuarios; definir que tipo de acceso utilizará; mover datos entre sus archivos; hacer copias de respaldo o *backups* y poder recuperar la información desde ellos.

Dentro de la visión que tiene un usuario (o las aplicaciones) al hablar de archivos, se utiliza frecuentemente los términos **campo** (*field*) para referirse al elemento de datos básico, tales como un nombre o una fecha, un **registro** (*record*) es un conjunto de campos relacionados que pueden tratarse como una unidad en algunos programas de aplicación, entonces un **archivo** será un conjunto de registros similares. El sistema operativo considera a un archivo como a una colección de **bloque lógicos** de tamaño fijo. Un bloque es la unidad básica de una operación de entrada y salida entre el disco y los *buffers* de memoria del sistema de archivos, particularmente en POSIX. El disco en sí mismo es una colección de **bloques físicos**. Cada bloque físico almacena un bloque lógico y posiblemente algunos datos administrativos. El tamaño del bloque es un múltiplo de la unidad básica de entrada y salida provista por el manejador de dispositivos del disco.

6.1.1 Archivos

6.1.1.1 Tipos

Los archivos representan datos y programas. Los archivos de datos pueden ser numéricos, alfabéticos, binarios. Un archivo es una secuencia de bits, bytes, líneas, registros, de acuerdo a lo que necesita tanto la aplicación como el usuario creador de ese archivo. Hay, entonces, diferentes tipos de archivo, tales como de texto, archivos fuente, archivos objeto, ejecutables. Pero una buena parte de los distintos tipos de archivos no son utilizados de manera directa por el usuario sino por aplicaciones.

El sistema operativo para trabajar de una manera razonable con un archivo debe saber qué tipo de archivo es. Imprimir un archivo binario o mostrarlo por pantalla, nos desplegará un serie de signos no legibles.

Por eso es importante que se almacene junto con la información relacionada con el archivo qué tipo de archivo es. Por ejemplo, es común que en los sistemas operativos tipo UNIX se utilice el llamado «número mágico» que es un número que está dentro del archivo, al comienzo de éste.

En UNIX existe el comando `file` que permite saber con qué tipo de archivo estoy trabajando. Antes de imprimir, por ejemplo, el archivo «`miarchivo`», me aseguro que sea un tipo legible colocando el comando:

```
$ file miarchivo
miarchivo: ASCII text
```

por tratarse de una archivo de texto ASCII puedo verlo o imprimirlo. Esto es así porque en general, en UNIX no se utiliza ninguna forma de declarar el tipo de archivo como parte del nombre de éste. Otros sistemas utilizan «extensiones» en los nombres de archivo para ayudar a identificar su contenido. Por ejemplo, en Windows para ayudar a declarar el tipo de archivo se le agrega un punto al final del nombre de archivo seguido por unas letras que ayudan a identificar el tipo. Entonces, siguiendo nuestro ejemplo, éste debería ser nuestro «`miarchivo.txt`». Las extensiones en los nombres de archivos entonces, no sólo ayudan a identificar el contenido sino también pueden servir para indicar la **estructura** interna del archivo, tal como vimos en la descripción del ejecutable PE en 3.2.2.3.

Es común organizar a un grupo de archivos relacionados entre sí en un **directorio**[1] o **carpeta** (*folder*), como contenedor de archivos y otros directorios, a los que les llamamos **subdirectorios** o **subcarpetas** por depender de éstos, formando una **estructura jerárquica** en forma de **árbol**.

Los sistemas UNIX tradicionales soportan dos tipos generales de dispositivos *hardware*, los de caracteres y los de bloque, y ambos son modelados como si se trataran de archivos de un tipo especial. Los dispositivos de caracteres tales como los puertos seriales y los teclados manipulan los datos como si fueran un flujo de caracteres o bytes, uno a la vez. A la inversa, los dispositivos de bloques manipulan los datos en grupos de un tamaño fijo llamados **bloques**, como vimos recientemente. Los dispositivos de bloques no se limitan a enviar o recibir

1. Este es el término difundido y aceptado entre los hispanohablantes como traducción de *directory*, que se refiere más bien a «guía telefónica»

un flujo de datos sino que cualquiera de sus bloques son accesibles. El moverse de un bloque a otro se le denomina **búsqueda** o *seeking*. Ejemplos de dispositivos de bloques son los discos rígidos y los dispositivos de CD-ROM. Nos extenderemos más en este tema cuando tratemos a los dispositivos de entrada y salida en el Capítulo 7.

6.1.1.2 Atributos

Los atributos de un archivo K. Smith y Seltzer (1994) son los metadatos[2] que describen o están asociados a un archivo, tales como el nombre, el tipo, el tamaño o medida, la localización (dónde se ubica), derechos o permisos de acceso, las fechas de creación, acceso o modificación, número identificador (UID) del creador, la extensión y la aplicación que lo utiliza. Podemos también citar a la volatilidad, que es la frecuencia con la que se agregan y borran ítems en un archivo; la actividad, que es el porcentaje de ítems accedidos durante un determinado período de tiempo y los comentarios que pueda hacer acerca del archivo. Los sistemas operativos dan soporte a estos atributos de distintas maneras, todas las formas sirven como mapas para acceder a la información del archivo en los dispositivos de almacenamiento.

El **nodo-i** (*index node* o *i-node*) de UNIX contiene información acerca del tipo de archivo (normal o regular, directorio, enlace simbólico o *link*, tubería o *pipe*, archivo dispositivo, *socket*[3], ver 6.1.4.2.2), permisos o modos de acceso, el propietario, el grupo propietario, tamaño, fechas de creación, de último acceso y de última modificación, y punteros a bloques de datos (ver 6.2.2).

El *Master File Table* o **MFT** de Windows describe el archivo mediante los siguientes atributos: cabecera, información estándar (por ejemplo fechas y conteo de enlaces), descriptor de seguridad, nombre de archivo y datos, el identificador de objeto (*Object ID*), entre otros M. E. Russinovich y D. A. Solomon (2004). También se almacenan atributos en los flujos o *streams* del sistema de archivos NTFS de Microsoft, como veremos en detalle más adelante en 6.2.1.2.

En MS-DOS la **tabla de directorio** (*directory table*) es un tipo especial de archivo que representa a un directorio (carpeta). Cada archivo o directorio almacenado dentro de ella está representado por una entrada de 32 bytes en la tabla, cada entrada registra el nombre, extensión, los atributos (bits de archivado, directorio, oculto, sólo-lectura, sistema y volumen), la fecha y hora de creación, la dirección del primer *cluster* de los datos del archivo o directorio y el tamaño del archivo o directorio. La tabla de directorio está almacenada en la Región de Datos (*Data Region*) de la Tabla de Asignación de Archivos (*File Allocation Table* o *FAT*).

> Una agrupación o *cluster* es un conjunto contiguo de sectores que componen la unidad más pequeña de almacenamiento de un disco. Son los bloques direccionables que usan muchos sistemas de archivos. El tamaño es siempre un múltiplo del tamaño del sector. Se utiliza el concepto de agrupación para administrar más eficientemente el espacio del disco (M. E. Russinovich y D. A. Solomon 2004).

2. datos que describen otros datos
3. un punto final de un flujo de comunicación bidireccional entre procesos

6.1.1.3 Operaciones

Existen muchas operaciones que podemos efectuar con archivos, las cuales varían según los programas que las efectúan sobre los distintos sistemas operativos, lo que nos lleva a relacionarlas con las distintas llamadas a sistema que ofrecen, pero las más comunes pueden ser «crear» el archivo con un nombre dado, «abrir» un archivo existente para su lectura y/o escritura, «leer» el contenido del mismo, «escribir» algo en él ya sea «actualizándolo», «reescribiendo», «añadiendo» al final o «truncarlo» a partir de un cierto punto, no sin antes «trabarlo» para tener acceso exclusivo, para luego «destrabarlo» y finalmente «cerrarlo». Pero si en lugar de manejarnos con su contenido necesitamos operar sobre él como un todo, podemos «colocarle atributos» o permisos, «copiarlo», «borrarlo» o «renombrarlo». Asociado a todas estas operaciones está el concepto de interpretación o **semántica**.

Estas operaciones se llevan a cabo a través de distintas llamadas a sistema, a veces como composición de unas pocas.

6.1.2 Particiones

Normalmente los discos rígidos no son usables por los sistemas operativos de propósito general hasta que no se les defina al menos una **partición** o áreas separadas de datos de un disco rígido físico, es decir que un disco rígido puede tener una única partición que abarca toda la capacidad del mismo o varias. Estas particiones son las que el sistema operativo toma como base para definir a los discos lógicos al crearles **sistemas de archivos**[4]. Entonces un mismo disco rígido puede contener varias particiones y cada una de ellas con un sistema de archivos distinto. El tener varias particiones acarrea varias ventajas, por una parte podemos separar los archivos propios del sistema operativo de los archivos que crean y usan los usuarios, de manera que ni el borrado ni la actualización del sistema operativo afectará a los archivos de los usuarios. Por otra parte, podemos destinar un área específica para el intercambio de páginas de la memoria virtual. También podemos colocar sistemas operativos distintos en esas particiones y tener la opción de elegir alguno de ellos al momento del arranque. Otra ventaja es que podemos destinar ciertas particiones para tener copias de seguridad, ya que si se daña un cierto sector perteneciente a una partición, esto no deja al disco completo inutilizable sino que puedo a partir de ese momento continuar utilizando la copia que había reservado como respaldo en la otra partición. También podemos decir que una partición es una colección de sectores contiguos en un disco. Una **tabla de particiones** almacena el sector de comienzo de una partición, su tamaño y otras características y está ubicada en el mismo disco que la partición.

Frecuentemente se le llama **volúmen** a objetos que representan sectores de una partición que los manejadores del sistema de archivos gestionan como una unidad. Los **volumenes** **multipartición** son objetos que representan sectores tomados a partir de varias particiones y que los manejadores del sistema de archivos gestiona como una unidad.

A menudo el tamaño y cantidad de los archivos puede llegar a ocupar toda la partición, y alterar el tamaño de una partición en uso no es tarea sencilla ni deseable ya que se corre el riesgo de perder los datos, en este caso se puede crear una estructura lógica que permite que

4. Operación que en la jerga se le llama «formatear», porque se efectuaba con el programa FORMAT del DOS.

el usuario trabaje una abstracción de la estructura de disco real, el sistema operativo ofrece al usuario un espacio que es la suma de la capacidad de almacenamiento de los discos reales, los sistemas operativos modernos nos permiten utilizar un **gestor de volumenes lógicos**, tales como *Logical Disk Manager* (LDM) de VERITAS Software y licenciado a Microsoft para el sistema operativo Windows y *Logical Volume Manager* (LVM) utilizado por Linux.

6.1.3 Nombres

Antes de continuar con nuestro estudio, nos permitiremos una pequeña digresión del tema que estamos tratando para comentar brevemente acerca del nombrado de objetos, que revisten suma importancia en los sistemas distribuidos pero su importancia en ciertos aspectos de los sistemas operativos no es menor, porque permite la identificación de objetos y servicios para que puedan ser accedidos de forma apropiada. A pesar de su importancia, no existe en la bibliografía un tratamiento unificado y general, por este motivo Sinha (1996) dedica una sección para definir y explicar los conceptos y terminología fundamentales asociados con el nombrado de objetos, fundamentalmente en los sistemas distribuidos, pero serán aplicables también a nuestro estudio, como él mismo reconoce.

Un **nombre** es una cadena compuesta por un conjunto de símbolos elegidos a partir de un alfabeto finito, por ejemplo abc es un nombre válido compuesto por símbolos del conjunto de caracteres ASCII, un nombre es un **identificador** porque se usa para denotar o identificar a un objeto. Un **sistema de nombres** emplea una o mas convenciones de nombrado para asignarle nombres a objetos. Al conjunto de nombres que cumplen con una dada convención de nombrado se dice que forma un **espacio de nombres** Terry (1984). El espacio de nombres más simple es el espacio de nombres **plano** porque no posee estructura alguna, pero cuando existe la necesidad de asignar nombres plenos de significado y que no resulten ambiguos a un gran conjunto de objetos se usa una convención de nombrado que particiona el espacio de nombres en clases disjuntas, entonces un **nombre compuesto** se compone de uno o más nombres simples que están separados por un carácter **delimitador** especial tal como «/», «@», «\», «.», etc. Por ende el nombre `/a/b/c/` es un nombre compuesto que consiste de tres nombres simples `a`, `b` y `c`. Una forma típica de espacio de nombres particionado es el espacio de nombres jerárquico en el cual el espacio de nombres está particionado en varios niveles y está estructurado como un árbol. A los nombres definidos en un espacio de nombres jerárquico se les llama **nombres jerárquicos**.

Los nombres jerárquicos han sido utilizados desde hace tiempo en los sistemas de archivos y se han adoptado recientemente para nombrar otros objetos y también en los sistemas distribuidos. Por otra parte, a veces estos nombres pueden volverse particularmente largos, en ese caso es posible definir unos nombres sustitutos llamados **abreviaturas** o **alias**.

Una de las características principales de un sistema operativo, de cara a los usuarios, es la forma en que se nombran los archivos. Todo objeto archivo debe tener un nombre a través del que se pueda acceder a él de forma inequívoca Abernathy y col. (1973).

Luego de esta discusión formal, podremos comprender ciertos temas que trataremos a continuación desde un panorama más amplio.

6.1.4 Sistema de archivos

El sistema de archivos (*file system* o *filesystem*) es un método de almacenamiento y organización de archivos de computadora y de los datos que contiene un medio de almacenamiento, para facilitar la búsqueda y el acceso.

Pueden usar un dispositivo de almacenamiento de datos tales como un disco rígido o un *Compact Disk* lo que conlleva mantener la ubicación física de los archivos; pueden proveer acceso a datos en un servidor de archivos actuando como clientes utilizando un protocolo de red; o pueden ser virtuales y existir sólo como un método de acceso para datos virtuales.

A continuación nos extenderemos en cada uno de estos conceptos para que se entienda cabalmente la definición.

6.1.4.1 Directorio

Previamente decíamos que es común organizar a un grupo de archivos relacionados entre sí en un directorio o carpeta, como contenedor de archivos y otros directorios. Un directorio, según Chapin (1969), es un objeto que relaciona de forma unívoca el nombre dado por el usuario a un archivo y el descriptor interno del mismo usado por el sistema operativo. Para evitar ambigüedades, un mismo nombre no puede identificar nunca a dos archivos distintos, aunque varios nombres se pueden referir al mismo archivo. Habitualmente, un directorio contiene tantas entradas como archivos son accesibles a través de él, siendo función principal de los directorios presentar una visión lógica simple al usuario, escondiendo los detalles de implementación del sistema de directorios Carretero Pérez (2001). Podemos pensar en el directorio como una tabla que traslada los nombres de archivo en la referencia necesaria para ubicar el archivo dentro de la estructura. Normalmente el directorio tiene información sobre los atributos de los archivos que pertenecen a él. Los servicios del sistema son los que acceden a esta estructura y proporcionan la información del directorio a usuarios y aplicaciones. Cuando se abre un archivo, el sistema operativo busca en el sistema de directorios hasta que encuentra la entrada correspondiente al nombre del archivo. A partir de dicha entrada, el sistema operativo obtiene el identificador interno del archivo y algunos de sus atributos. Todas las referencias posteriores al archivo se resuelven a través de su descriptor (nodo-i, registro MFT, etc.). Cada sistema organiza esta información a su manera y básicamente hay dos alternativas principales: o bien almacenar los atributos del archivo en su entrada del directorio, o almacenar únicamente el identificador del descriptor de archivo (en UNIX, el nodo-i) y colocar los atributos del objeto archivo dentro de la estructura de datos asociada a su descriptor. Los sistemas operativos modernos usan esta última solución, porque tiene varias ventajas respecto a la primera. Pero existen sistemas operativos tales como CP/M o MS-DOS, que almacenan atributos del archivo en su entrada del directorio.

La estructura creada por la colocación de nombres en directorios puede tomar un número de formas: directorios de un nivel, de dos niveles, árboles de múltiples niveles, grafos acíclicos o grafos generales.

6.1.4.1.1 Directorio de un nivel

En este sencillo enfoque todos los archivos están contenidos en un único directorio, es decir que no existe el concepto de subdirectorio. Es fácil de programar y entender pero se complica cuando hay muchos archivos y más de un usuario porque los archivos deben tener nombres únicos.

CP/M tenía un único directorio, en el que cada entrada incluía toda la información suficiente para acceder al archivo asociado.

Código Usuario	Nombre Archivo	Tipo	Grado	Conteo	Nros de bloques de disco
1 byte	8 bytes	3 bytes	1 byte	1 byte	16 bytes

Tabla 6.1

«Código de Usuario» fue agregado con el objeto de hacerlo compatible con MP/M pero como CP/M era un sistema operativo monousuario, este campo era meramente indicativo ya que no existe en la estructura información acerca de la protección del archivo. «Grado» indica cuántas entradas toma el archivo, si tiene más de 16 bloques. «Conteo» indica cuántas de las 16 entradas del bloque están en uso.

6.1.4.1.2 Directorio de dos niveles

En esta solución hay un directorio para cada usuario. Cada usuario tiene su UFD (*user file directory*) y cada UFD tiene una estructura similar.

Existe un MFD (*master file directory*) que esta indexado por nombre de usuario o cuenta y que apunta al UFD del usuario, como se aprecia en la figura 6.1.

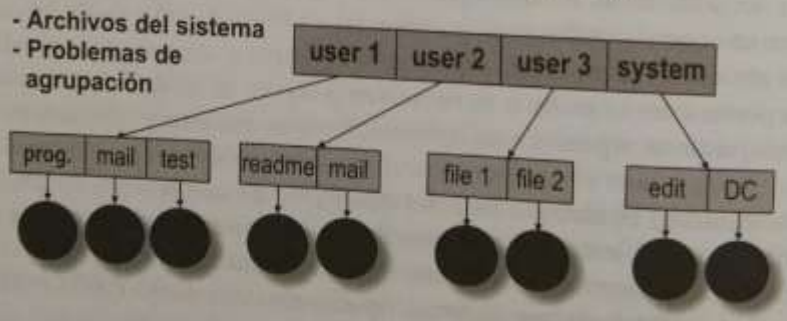


Fig. 6.1 – Directorio de dos niveles.

Al crear un archivo se comprueba el UFD del usuario para ver si ese nombre ya existe. Cuando un usuario quiere borrar un archivo que tiene determinado nombre, pero que también pertenece al UFD de otro usuario, no podrá hacerlo hasta que se lo borre desde ambos UFD. sus actividades se limitan a su UFD Needham y Birrell (1977).

El problema de esta estructura es que aisla un usuario de otro, en el caso de que estos necesiten cooperar, y tener archivos comunes Lampson y Sturgis (1976).

El MFD era una estructura de datos esencial en el sistema de archivos del ITS[5] (Zuzga 1995). Similar al «directorio raíz» en un sistema de archivos jerárquicos excepto que ITS tenía un sistema de archivos plano con sólo un nivel de directorios, el MFD contenía una lista de todos los directorios en el disco. Cada usuario tenía su propio directorio y cada directorio tenía un número índice único asociado con él. El autor de un archivo se registraba como un número índice a su directorio en el MFD.

6.1.4.1.3 Estructura de árbol

Es la evolución lógica de los niveles vistos previamente, un árbol multinivel permite el crecimiento del árbol más allá del segundo nivel. Cualquier directorio puede contener nombres de más directorios como así también de objetos que no sean directorios. Puede utilizarse tanto en sistemas monousuarios como multiusuarios. El usuario puede crear sus propios subdirectorios y así, organizar sus archivos como le plazca. En un sistema multiusuario, cada usuario tiene un directorio inicial de *login*, que es el directorio donde queda el usuario cuando inicia una sesión. Tiene comandos para ir moviéndose por la estructura (cd, change directory).

En MS/DOS, la entrada del directorio tiene 32 bytes de largo.

Nombre	Extensión	Atributos	Reservado	Tiempo	Fecha	1er. nro. bloque	Tamaño
8 bytes	3 bytes	1 byte	10 bytes	2 bytes	2 bytes	2 bytes	4 bytes

Tabla 6.2

La primera implementación de la estructura de árbol multinivel (o sistema de archivos jerárquico) data del sistema operativo Multics Daley y Neumann (1965). Al primer nombre en la estructura, a partir del cual dependen todos los demás, se le denomina **raíz** (*root*) del árbol. Pero en realidad utilizamos una estructura levemente modificada del árbol, que es el **grafo**.

6.1.4.1.4 Estructura de grafo

En esta estructura podemos tener un objeto con varios nombres. Y como dos enlaces (nombres) llevan a uno de los objetos, la estructura ya deja de ser un árbol para pasar a ser un grafo. Esta estructura mejora las capacidades para compartir archivos al permitir que el mismo objeto sea nombrado en directorios distintos. Y más flexibilidad se le puede agregar al sistema de archivos al permitirle a los objetos directorio el tener varios nombres. Esto puede ser especialmente útil en ambientes en los que varios usuarios necesiten compartir un cierto número de archivos asociados con algún proyecto en el cual estén trabajando. Cada usuario puede agregar un nombre al directorio del proyecto desde su propio directorio personal. Y a medida que los archivos en el directorio del proyecto se agreguen, borren o modifiquen, el

5. *Incompatible Timesharing System.*

directorio del proyecto con nombre en cada uno de los directorios personales de los usuarios reflejará estos cambios.

Si bien son más flexibles, las estructuras de nombres de grafos agregan complejidad. En los sistemas en los que cada objeto tiene sólo un nombre, la distintición entre nombre y objeto es rara vez importante. Claramente este no es el caso cuando un objeto tiene varios nombres. No se puede determinar si dos nombres se refieren al mismo archivo con sólo comparar los nombres para ver si son idénticos. La supresión de un objeto plantea más preguntas. ¿Existen comandos separados para la supresión de nombres y objetos? ¿Se puede eliminar un objeto antes de que se hayan eliminado todos los nombres para él? Si no ¿cómo determina el sistema si todos los nombres se han eliminado? En UNIX la primitiva de borrado elimina sólo el nombre. Si el nombre es el único nombre asociado con el archivo entonces también se elimina el archivo.

La solución de estos problemas es mucho más difícil cuando la estructura de nombres es un grafo que contiene ciclos. Los objetos pueden tener ahora un número infinito de nombres (dependiendo de cuántas veces se atraviesa un ciclo). Los objetos pueden ser limitados a un número finito de nombres restringiendo la estructura a un grafo acíclico. Desafortunadamente, los algoritmos generales para determinar si un enlace dará como resultado un ciclo consumen mucho tiempo. Como resultado, las estructuras de grafos, ya sean generales o acíclicos, rara vez se implementan.

Por ejemplo, UNIX utiliza una forma limitada de organización acíclica para proporcionar la facultad de contar con varios nombres pero a la vez limitando la complejidad. Los objetos que no son directorios pueden tener varios nombres sin ninguna restricción. Sin embargo, sólo el sistema puede crear nombres adicionales para los objetos directorio. Cada vez que se crea un directorio se agregan dos nombres dentro del directorio nuevo, el punto «.» y los dos puntos seguidos «..». El nombre «.» es un alias para el directorio creado recientemente, mientras que los dos puntos «..» es un alias para el directorio padre del directorio creado recientemente. Los nombres «.» y «..» crean ciclos. Los nombres

```
directorio1/archivo
directorio1/../directorio1/archivo
directorio1/./archivo
./directorio1/./archivo
```

se refieren todos al mismo `archivo`. Sin embargo, puesto que «.» y «..» son los únicos nombres que pueden crear ciclos y esos ciclos son detectables fácilmente, la complejidad creada por su inclusión es limitada. Los directorios «.» y «..» también están implementados en la FAT de DOS y en NTFS de Windows.

6.1.4.1.5 Entradas de directorio

La información acerca de los archivos está almacenada en una entrada de directorio. La información almacenada depende del sistema operativo. Típicamente una entrada de directorio contiene información acerca de la propiedad del archivo, ubicación, tamaño, derechos de acceso, fechas de diferente clases de eventos (última modificación, creación). La mayoría de los sistemas operativos requiere que un archivo sea **abierto** antes de que pueda ser accedido. La operación de apertura obtiene un puntero a la ubicación de directorio del archivo, enton-

ces, todas las referencias posteriores al archivo pueden ser a través del puntero en lugar de a través del nombre del archivo.

En algunos sistemas, la entrada de directorio apunta a una estructura de archivo distinta en la que está almacenada la información acerca del archivo. Por ejemplo, en los sistemas Linux (y en UNIX a partir de System V) la información del archivo está almacenada en una estructura llamada **nodo-i** o **nodo índice** (*index node* o *inode*). La entrada de directorio (*directory entry* o **dentry**) contiene sólo el nombre del archivo y su número de nodo-i, es decir que el directorio es un archivo que asocia nodos índice a nombres de archivo. El núcleo de Linux no tiene una estructura de datos especial para representar un directorio, el cual se trata como a un archivo normal en la mayoría de las situaciones. La estructura del archivo directorio es una tabla donde cada entrada tiene una estructura, simplificada y genérica, como la siguiente:

Numero de nodo-i	Nombre del archivo
2 bytes	X bytes

Tabla 6.3

Porque, en realidad, la estructura de una entrada de directorio en el sistema de archivos `ext3` de Linux, está definida en un archivo fuente del núcleo Linux, que es `/usr/src/linux/include/linux/ext3_fs.h` y como sigue:

```
#define EXT3_NAME_LEN 255
struct ext3_dir_entry {
  __le32 inode; /* Inode number */
  __le16 rec_len; /* Directory entry length */
  __le16 name_len; /* Name length */
  char name[EXT3_NAME_LEN]; /* File name */
};
```

Es decir que un nodo-i está asociado con cada archivo y mantiene toda la información referida a él excepto su nombre y sus datos. El propietario, el grupo propietario, permisos y fechas de creación, acceso y modificación, son datos almacenados en el nodo-i, así como el tamaño de los datos que contiene y el número de enlaces.

Otra estructura de datos importante en el sistema de archivos de Linux es el **súper bloque** (*super block*), que debe su nombre a su herencia UNIX, desde cuando se utilizó el primer bloque de datos de un disco o partición, para mantener la metainformación de la partición misma. El super bloque ahora está separado del concepto de bloque de datos, pero todavía contiene información acerca de cada sistema de archivos montado. La estructura de datos real en Linux se llama `struct super_block` y contiene diversa información de mantenimiento (*housekeeping*), como opciones de montaje, fechas del momento de montaje y tamaño de bloque del dispositivo. El núcleo en la versión 2.0 mantiene un arreglo estático de estas estructuras para manejar hasta 64 sistemas de archivos montados.

6.1.4.2 Designación de un archivo

El usuario se refiere al archivo por su nombre simbólico, que debe ser único para permitir su localización sin ambigüedad. En un sistema jerárquico, cualquier archivo puede ser localizado siguiendo un camino desde el directorio **raíz** o maestro, descendiendo por las ramas hasta el directorio donde se ubica el archivo. Este es el nombre **completo** del camino (*full pathname*) del archivo. Los caracteres separadores son por ejemplo, la barra de división «/» en todos los sistemas tipo UNIX, tales como Linux, y la barra invertida «\» en Windows.

En un sistema operativo multiusuario, el directorio asociado con cada usuario es su directorio personal (*home directory*) o de *login* o sesión, que es en el que tiene permisos para crear y borrar archivos y directorios así como también crear y ejecutar programas propios. Como el usuario (o la aplicación) trabaja normalmente sobre ese directorio, se refiere a sus archivos directamente, sin dar el nombre completo y el sistema asume que se encuentran en su directorio personal. La mayoría de los sistemas utilizan el concepto de directorio actual (*current directory*) o de trabajo. Se dice entonces que una **ruta** o **camino** es **relativa** (*relative pathname*) con respecto al directorio actual. Más específicamente, al ejecutar un programa (o comando) el sistema lo busca por su nombre en las posibles rutas a directorios que contienen ejecutables y, si lo encuentra, lo ejecuta, pero si no, emite un mensaje de error. Tanto Windows como DOS y Linux utilizan estas rutas, pero difieren en que, generalmente, en Linux no se incluye el directorio actual en la ruta de búsqueda.

En algunos sistemas, tales como Windows (y desde DOS en adelante), el disco lógico establecido dentro de una partición es parte del camino o ruta para designar un archivo, por ejemplo `C:\WINDOWS\WINDOWS.INI` nos indica que el archivo `WINDOWS.INI` está dentro de la carpeta `\WINDOWS` en el disco lógico `C`, los dos punto «:» se utilizan como carácter separador entre el nombre del disco lógico del nombre y los nombres de archivos. Mientras que en los sistemas tipo UNIX, con el objeto de ocultar las particularidades de los dispositivos de almacenamiento (recordemos que ocultar los detalles del *hardware* para facilitar la comprensión a los usuarios, es un objetivo de los sistemas operativos), los sistemas de archivos definidos en las particiones de los discos rígidos (y además en todos los medios de almacenamiento) deben ser **montados** para ofrecerle al usuario la visión de un único espacio de nombres unificado. Para llevar a cabo esta operación hay que designar por una parte, la partición (o medio de almacenamiento) a ser montada y el directorio, que debe existir previamente, y se le denomina **punto de anclaje** o **punto de montaje**. Primero debe montarse la partición (o el medio) que contenga los archivos que deban anclarse a la raíz (/) del sistema (*root*) y posteriormente se podrán ir montando otras particiones o medios en distintos directorios que dependan de esta raíz, por ejemplo, podemos tener otra partición separada que contiene los archivos de los usuarios y montarla en el directorio `/home`. De manera que en ningún momento vemos al nombre de los discos lógicos equivalentes a `C:` o `D:`

6.1.4.2.1 Directorio raíz por proceso

En Linux es posible designar a un directorio que servirá como directorio raíz de un proceso. Una vez que se efectúa esa designación sólo el subárbol encabezado por ese directorio es visible a ese proceso. En un sistema en el que no se desea compartir archivos, el directorio personal de cada usuario podría ser el directorio raíz para todos los procesos que el usuario

ejecute. Para esos procesos, la raíz del sistema de archivos sería el directorio personal del usuario. Esta capacidad también puede ser utilizada por las aplicaciones para proporcionar seguridad adicional. Es decir, se puede configurar un servidor Web o FTP de esta manera limitando su acceso a un subárbol reservado para los datos Web o FTP.

6.1.4.2.2 Otros tipos de objetos

En muchos sistemas operativos, los nombres en el sistema de archivos se pueden referir a objetos que no son ni archivos ni directorios. Los objetos que se pueden encontrar en un sistema de archivos pueden ser los **accesos directos** (*shortcut*) o **enlaces blandos** (*soft link*), que son un puntero a otro nombre en el sistema de archivos; en la mayoría de los casos referirse al acceso directo es equivalente a referirse al nombre apuntado por el acceso directo y al borrar el nombre apuntado por el acceso directo genera un puntero colgante (*dangling pointer*). También podemos encontrarnos con **tuberías** (*pipes*) que es un canal o cauce de comunicación entre dos procesos, como vimos en 5.6.3. O bien un **archivo dispositivo**, o un **semáforo**, como vimos en 5.4, o **memoria compartida**.

6.1.4.3 Journaling

Algunos sistemas de archivos implementan un sistema de **transacciones** que se le suele llamar **registro por diario**, **bitácora** o ***journaling*** en inglés, de manera que todas las interacciones con las estructuras de datos involucradas que deberían aplicarse una luego de la otra para mantener la integridad, se realicen de una sola vez y que la estructura a medio manipular no sea jamás alcanzable por el resto del sistema hasta que haya finalizado todos sus procesos, es decir debe ser atómica. En el *journal* se registra la información necesaria para restablecer los datos afectados por la transacción en caso de que ésta falle. Es decir que en caso de que hubiere un fallo eléctrico en el medio de la transacción, es posible revertir los cambios efectuados a las estructuras de datos hasta ese momento cuando se inicie nuevamente el sistema, entonces el sistema de archivos no queda inconsistente. Es decir, que no es necesario revisar exhaustivamente el sistema de archivos, sino que se revisa el *journal* y como tiene un registro cronológico de todos los cambios recientes, simplemente inspecciona aquellas porciones de los metadatos que hayan sido modificadas recientemente, restableciendo el sistema a un estado consistente en cuestión de segundos aún en sistemas de varios gigabytes.

6.1.5 Arquitectura del software

Decíamos que un sistema de archivos presenta dos problemas de diseño muy distintos. Por una parte hay que definir qué aspecto debe presentar el sistema de archivos a los usuarios. Por otra, crear algoritmos y estructuras de datos que establezcan una correspondencia entre el sistema de archivos lógico (como los ve el usuario) y los dispositivos de almacenamiento secundario físicos, como los ve el sistema operativo.

El *software* de gestión de archivos en sí generalmente se compone de varios niveles y componentes diferentes, por supuesto que los diferentes sistemas se organizan de forma diferente, pero esta organización es razonablemente representativa. La figura XXX es un ejemplo

hipotético de diseño por capas. Cada nivel del diseño aprovecha las funciones de los niveles inferiores para crear nuevas funciones que se usarán en los niveles superiores.

En el nivel más cercano al *hardware*, el **control de entrada y salida**, consta de manejadores de dispositivo (*device drivers*) y manejadores de interrupciones para transferir información entre la memoria y el sistema de disco, se comunican directamente con los dispositivos periféricos o sus controladores o canales. El sistema operativo Windows soporta aún una arquitectura de capas entre los manejadores de dispositivo, en la que cada dispositivo es atendido por una cadena de manejadores llamados **pilas de manejadores** (*driver stack*)[6]. Podemos considerar un *driver* de dispositivo como un traductor Silberschatz y Galvin (1997). Su entrada consiste en instrucciones de alto nivel, como «recuperar el bloque 123». Su salida consiste en instrucciones de bajo nivel específicas para el *hardware*, que son utilizadas por el controlador en *hardware* (*hardware controller*) que comunica el dispositivo de entrada y salida con el resto del sistema, como veremos con mayor grado de detalle en el Capítulo 7.

El **sistema de archivos básico** o nivel de **entrada y salida física**, sólo necesita emitir órdenes genéricas al *driver* de dispositivo apropiado para leer y escribir bloques físicos en el disco. Se encarga de ubicar dichos bloques de datos en el dispositivo de almacenamiento secundario y del almacenamiento intermedio de los mismos en la memoria principal. Este nivel no comprenderá el contenido de los datos o la estructura de los archivos implicados (Stallings 2001).

Silberschatz y Galvin 1997 agregan las capas del **módulo de organización de archivos**, que conoce los archivos y sus bloques lógicos, además de los bloques físicos y puede traducir las direcciones de bloques lógicos en direcciones de bloques físicos para que el sistema de archivos básico realice la transferencia. Y el **sistema de archivos lógico** que emplea la estructura de directorios para proporcionar al módulo de organización de archivos la información que éste necesita, a partir de un nombre de archivo simbólico.

Por otra parte, Stallings (2001) agrega al **supervisor básico de entrada y salida** que es el responsable de la iniciación y terminación de toda la entrada y salida con archivos. Se ocupa de la selección del dispositivo donde va a realizarse la entrada y salida con los archivos, que depende del archivo seleccionado. La siguiente capa corresponde a la **entrada y salida lógica** que es la parte del sistema de archivos que permite a usuarios y aplicaciones acceder a los registros. Así, mientras el sistema de archivos básico trabaja con bloques de datos, el módulo de entradas/salida lógica lo hace con registros. Y finalmente, el nivel del *software* de gestión de archivos más cercano al usuario es el **método de acceso**.

De cualquier manera, cabe destacar que los actuales sistemas operativos modernos utilizan distintas arquitecturas de *software* para la gestión de archivos, algunos de estas capas o módulos operan en modo núcleo y otras lo hacen en modo protegido. Notoriamente los sistemas operativos de arquitectura de micronúcleo como Minix[7], que está estructurado en cuatro capas (Tanenbaum y Woodhull 2005), la más cercana al *hardware* atrapa todas las interrupciones y trampas, realiza la planificación y ofrece a las capas superiores un modelo de procesos secuenciales independientes que se comunican empleando mensajes. La siguiente o capa 2, contiene los procesos, más bien dicho las **tareas**, de entrada y salida, una por cada tipo de dispositivo, que se denominan **manejadores de dispositivos** indistintamente. Todas las tareas de la capa 2 y todo el código de la capa 1 se combinan para formar un solo progra-

6. `http://msdn.microsoft.com/en-us/library/ms791644.aspx`
7. sin olvidar que el propósito de Minix es didáctico, no es un sistema para uso en producción.

ma binario, el núcleo o *kernel*. La capa 3 contiene procesos servidores, entre ellos el sistema de archivos, que ejecuta todas las llamadas al sistema relacionadas con archivos.

6.1.6 Estructuras asociadas

Cuando un proceso abre un archivo efectúa una llamada a sistema y es el sistema operativo el encargado de abrirlo efectivamente y de gestionar el *hardware* para que realice todas las operaciones de entrada y salida sobre él. El sistema operativo le devuelve un **descriptor de archivo** (*file descriptor*) al proceso, que es una clave abstracta para acceder al archivo. Este término es usado por los sistemas operativos que siguen la norma POSIX (por ejemplo, Linux) y el equivalente en la terminología del sistema operativo Windows es **mando** o **identificador de archivo** (*file handle*).

En los sistemas operativos que siguen la norma POSIX, un descriptor de archivos es un número entero tipo `int` en lenguaje C y cada proceso tiene al iniciarse tres descriptores, la **entrada estándar** (*stdin*) cuyo número descriptor es cero, la **salida estándar** o *stdout* cuyo número es uno y el **error estándar** (*stderr*) con valor dos.

Los flujos o *stream* son otra forma de representar la conexión entre el proceso y el archivo y están representados como objetos tipo `FILE *` en lenguaje C. Los *streams* proveen una interfaz de mayor nivel en una capa adicional de software sobre las primitivas de descriptores de archivos, por esto se puede extraer el descriptor de archivo de un *stream* para realizar operaciones de entrada y salida de más bajo nivel.

Los descriptores de archivo forman una tabla, la tabla de los descriptores de archivos abiertos, y son un índice para una entrada en una estructura de datos del núcleo que contiene los detalles de todos los archivos abiertos. Como podemos ver en la figura 6.2 cada entrada de proceso en la tabla de procesos contiene una tabla de descriptores de archivos abiertos. Los descriptores de archivos sirven de índice para las entradas en esta tabla y cada entrada contiene un puntero a una entrada en una tabla que el núcleo mantiene para todos los archivos abiertos (*open files table*), la **tabla de los archivos abiertos**. La tabla de descriptores de archivo se conserva cuando se cambia el programa de un proceso con el servicio POSIX `exec`.

Cada entrada en esta tabla de los archivos abiertos contiene las banderas de estado del archivo (leer, escribir, añadir, etc), el desplazamiento en el archivo y un puntero a la entrada para este archivo en la así llamada tabla de **nodos-v** (*v-node table*) (Goodheart y J. Cox 1994).

Kleiman 1986 desarrolló el concepto de nodo virtual y a partir de ahí se ha seguido utilizando hasta la actualidad. Las estructuras de nodos-v (*virtual node*) son una parte esencial del soporte para el Sistema de Archivos Virtual (*Virtual File System - VFS*) de Linux. En el sistema de archivos original de UNIX el nodo-i representaba al archivo (o directorio o archivo especial), como hemos visto, y estaba referenciado por un número. Las entradas de los directorios simplemente contenían una lista de nombres de archivo con su correspondiente número de nodo-i, y este esquema aún persiste, pero el sistema VFS basado en nodos virtuales permite sistemas de archivos de cualquier tipo, tanto sistemas de archivos para ser usado en discos como sistemas de archivos de red o sistemas de archivos especiales, y retiene el concepto original de nodo-i ahora llamado nodo-v (*vnode*), se preserva una correspondencia entre la «vieja forma» y la «nueva forma», la estructura lógica de datos tal como la ven las

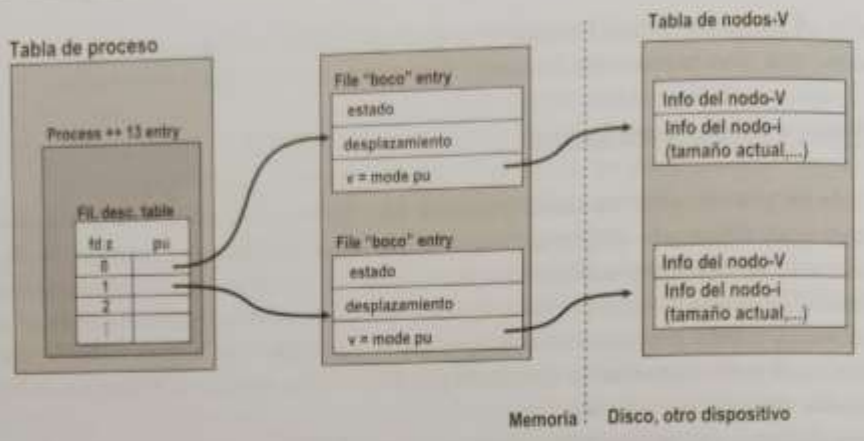


Fig. 6.2 – Estructura de datos del núcleo para los archivos abiertos.

aplicaciones existe tal como siempre y aisla a los programas de las diferencias físicas del medio de almacenamiento utilizado.

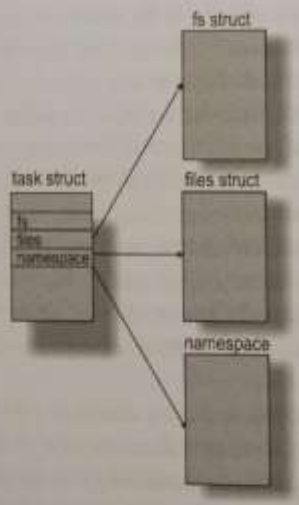


Fig. 6.3 – Estructuras relacionadas en el VFS.

Con el esquema que vemos en la Figura 6.3 dos procesos pueden abrir el mismo archivo, manteniendo cada uno su modo de acceso y desplazamiento. Por lo tanto, el VFS de Linux debe almacenar información acerca de cómo interactúan los procesos y los archivos. La estructura `fs_struct` mantiene toda la información que asocia a un proceso en particular con un archivo. Esta estructura puede ser referenciada por varios descriptores de procesos, es decir, es común que una estructura `fs_struct` que representa a un archivo esté referenciada por varios descriptores `task_struct`. El código donde está definido la estructura `fs_struct` es en el archivo de cabecera ubicado en el subdirectorio de los archivos fuente del núcleo, en `include/linux/fs_struct.h`

La estructura `files_struct` contiene información relacionada con los archivos abiertos y sus descriptores. Su definición la encontramos dentro de `include/linux/fdtable.h`. La solución utilizada en Linux es más compleja y eficiente que la del UNIX tradicional y está muy bien desarrollada en Bovet y Cesati (2005).

La tabla de nodos-v (o parte de ella) está almacenada en el dispositivo físico, no nos preocuparemos por ella por ahora pero no podemos pensar en sus entradas como el contenido «real» en el disco, con información asociada como la ubicación del archivo en el disco, tamaño, nombre, propietario, etc.

Ahora, qué pasa si dos procesos quieren abrir independientemente el mismo archivo físico, es decir, con cada proceso manteniendo su propio modo de acceso (leer, escribir, añadir) y desplazamiento del archivo también? Fácil: mueva la flecha derecha inferior para que apunte a la caja de la derecha superior y nos encontramos con la situación de la figura 7: aquí la tabla de archivos abiertos tiene dos entradas independientes para el mismo archivo, cada una asociada con uno de los procesos. Este es exactamente el arreglo a través del cual un hijo recién bifurcado comparte los archivos de su padre (pero a diferencia de la figura el hijo se refiere a ellos a través de descriptores de archivo iguales a los del padre).

Ahora, qué ocurre con esta disposición cuando se realizan operaciones de entrada y salida?

- Cada vez que se completa un `write()`, el desplazamiento en la entrada de la tabla de archivos se incrementa en el número de bytes escritos. Si esto provoca que se incremente el tamaño del archivo, esta información se actualiza en la entrada de la tabla de nodos-v.
- Si un archivo se abre `O_APPEND`, se coloca una bandera en la entrada de la tabla del descriptor de archivo. Cada vez que se efectúa un `write()` el desplazamiento en la entrada de la tabla del archivo para este archivo se pone primero en el tamaño actual del archivo. Esto fuerza a que los datos sean añadidos al final del archivo.
- La operación `lseek()` sólo lee/modifica el desplazamiento en la entrada de la tabla de archivos, no se realiza ninguna entrada y salida.
- Cada `read()` es realizado en el desplazamiento actual.

Ahora es fácil ver que este escenario, si bien es útil, es la fuente de varios tipos de problemas llamados colectivamente «condiciones de carrera» que se deben precisamente a la posibilidad de acceso concurrente al mismo archivo por procesos diferentes.

6.1.7 Organización de los archivos

Hay dos métodos fundamentales para acceder información dentro de un archivo, la secuencial y la directa. En el **acceso secuencial** la información en un archivo debe accederse en el orden en el cual está almacenada en el archivo. El acceso es secuencial desde el comienzo hasta el fin. Las operaciones de lectura o escritura no necesitan especificar la ubicación lógica dentro del archivo porque el sistema operativo mantiene un puntero al archivo que determina la ubicación del próximo acceso. El ejemplo típico de acceso secuencial a un archivo era el del acceso a un archivo almacenado en cinta.

Con el **acceso directo** cualquier ubicación lógica dentro del archivo puede accederse en cualquier momento. Típicamente el acceso directo se logra en una de dos maneras, o bien

mediante la especificación de la ubicación lógica a acceder como parámetro a la operación de lectura o escritura, o bien mediante la especificación de la ubicación en una operación de búsqueda o *seek* a ejecutar antes que la operación de lectura o escritura.

Los sistemas de bases de datos usan las dos operaciones de acceso básicas del sistema operativo para implementar una amplia variedad de métodos de acceso de alto nivel. Algunos sistemas operativos implementan ellos mismos métodos de acceso de alto nivel. De los métodos de acceso de alto nivel, el acceso **indexado** es tal vez el más significativo. Con el acceso indexado cada registro de archivo tiene uno o más campos y un campo sirve como el campo índice. Las operaciones de lectura y escritura incluyen un parámetro índice y el registro cuyo valor índice coincida, es el registro sobre el cual se opera.

Para cualquier método de acceso que tratemos, las operaciones de lectura o escritura pueden ser **síncronas** o **asíncronas**. La entrada o salida síncrona bloquea al proceso hasta que la operación de entrada o salida se haya completado, en cambio la entrada o salida asíncrona devuelve el control al proceso de forma inmediata, dejando libre al proceso para que pueda continuar su ejecución hasta que se haya realizado la operación de entrada o salida. Si las operaciones de ingreso de datos son asíncronas, se deberá emplear algún mecanismo para notificar al proceso que ha finalizado la operación. Esto se puede llevar a cabo enviándole al proceso una señal, colocándole un valor en particular a una de las variables del proceso o proveyendo una llamada a sistema especial para comprobar el estado de la operación de entrada o salida. La operaciones de salida asíncronas pueden emplear las mismas técnicas de notificación o no ofrecer ningún tipo de notificación.

La norma es la entrada o salida síncrona y hace que la programación de aplicaciones sea mucho más simple. La alta velocidad de las operaciones de entrada y salida en archivos provee de pocos incentivos para el uso de la entrada o salida asíncrona. Sin embargo, la entrada o salida asíncrona se puede usar ocasionalmente con la entrada y salida a dispositivos.

En una época, al definir en una aplicación durante la apertura del archivo (es decir, en el `OPEN()`) cuál sería su modo de acceso, esto era obedecido en gran manera por el sistema de archivos, porque la relación entre el archivo tal como lo veía la aplicación o el usuario y el archivo mismo almacenado en el dispositivo era más directa, pero con el evolucionar del *hardware* y del *software* la sentencia ya no es obedecida, por los motivos vistos en 6.1.5, ahora el manejador del dispositivo y la controladora en *hardware* juegan un rol mucho más protagónico, en lugar de asignarse registros, se utilizan bloques y agrupaciones o *clusters* y éstos sí intentan asignarse en forma secuencial, como veremos luego.

6.1.7.1 Gestión del espacio libre en disco

Además de seguir dónde está ubicado cada archivo en almacenamiento secundario, un sistema operativo debe estar capacitado para identificar a los bloques físicos que no están asignados actualmente a ningún archivo. Se han usado varios mecanismos para mantener una **lista de bloques libres**.

Por ejemplo, cada bloque puede estar o bien en uso o no, por lo tanto su estado puede estar almacenado como un único bit. El usar una serie de bits, un bit por bloque físico, nos da como resultado un **mapa de bits** de la lista de bloques libres (*free space bitmap*). Los bits están empacados o comprimidos en tan pocas palabras como sea posible. A pesar de que este mecanismo hace un uso eficiente del espacio, el acceso a la información de un bloque en

particular se hace más lenta por el procesamiento extra necesario para extraer información de bits individuales a partir de una palabra.

Una **matriz o arreglo** de lista de bloques libres (*free list array*) almacena las direcciones de todos los bloques libres. Entonces se considera que un bloque estará en uso si su dirección no figura en la matriz o arreglo. Un arreglo permite la implementación eficiente de operaciones para agregar o borrar bloques de la lista.

La lista también puede ser mantenida mediante la creación de una lista encadenada de todos los bloques libres. Una ventaja de este esquema es el uso eficiente del espacio, los bloques libres mismos almacenan la información de la lista de bloques libres.

Ambos métodos precedentes se pueden modificar para hacer un seguimiento de grupos de bloques libres en lugar de bloques libres individuales. En lugar de un mapa de bits, cada elemento en una matriz o **arreglo de huecos libres** podría almacenar el número de bloque de comienzo y número de bloques en un grupo de bloques libres. Si bien cada elemento en el arreglo debe estar representado por mucho más que un sólo bit, son necesarios menos elementos en el arreglo. Estos esquemas se pueden usar para encontrar de modo eficiente un gran número de bloques libres, particularmente si deben estar contiguos.

Un esquema de **lista de huecos encadenados** (*linked hole list*) involucra una lista encadenada modificada de manera similar. El primer bloque en cada hueco contiene el número de bloques en el hueco, así como un puntero al primer bloque en el próximo hueco en la lista.

6.1.7.2 Técnicas de asignación

Existen tres esquemas básicos para asignar (*allocate*) espacio a los archivos en almacenamiento secundario. El esquema de asignación es el responsable de la correspondencia (asignación o trazado) entre los bloques lógicos de un archivo y los bloques físicos en un dispositivo de almacenamiento secundario. En la mayoría de los sistemas operativos el tamaño del bloque físico es una potencia de 2 entre 512 y 4096.

El esquema más simple de **asignación** es:

Contigua. Los bloques lógicos de un archivo se almacenan en una partición de bloques físicos contiguos. La entrada del directorio sólo necesita almacenar la dirección de comienzo del archivo y su tamaño, en el almacenamiento secundario. La ubicación física de cualquier byte en el archivo se puede calcular agregando el desplazamiento apropiado a la dirección de comienzo del archivo en el almacenamiento secundario.

Cuando se crea un archivo, la asignación contigua requiere que haya preasignación de espacio para el archivo. El diseño del sistema operativo puede prever futuras expansiones del archivo en caso de que estas fueran necesarias. Si está permitida la expansión y el espacio de almacenamiento más allá del final del archivo está siendo usado, entonces habrá que mover uno o más archivos para hacer lugar a la expansión del primero. A los archivos a mover se les debe asignar nuevas ubicaciones en el disco y luego copiarlos ahí.

La asignación contigua de archivos en almacenamiento secundario comparte muchas de las características de la administración de memoria con particiones variables. El efecto de tablero de ajedrez deja al almacenamiento con un número de «huecos» de almacenamiento sin usar y se debe usar algún algoritmo de «encaje» para seleccionar el hueco al cual el archivo será asignado.

Hay una tendencia con todos los algoritmos de «encaje» a desarrollar un gran número de «huecos», haciendo que la fragmentación externa sea un problema. Se puede utilizar la **recolección de basura** o *garbage collection* como técnica de compactación, para recolectar todo el espacio libre desperdigado, colocando todo el espacio utilizado al principio y consolidando un único y gran «hueco» al final del espacio de almacenamiento. No obstante, la recolección de basura es particularmente cara en un sistema de archivos, dadas la relativamente baja velocidad de los dispositivos por un lado y los grandes tamaños involucrados por el otro. Por esta razón, el uso de esta técnica está bastante limitido.

Como consecuencia de estas limitaciones, la asignación contigua ya no se usa en sistemas de archivos modificables pero es ideal en sistemas cuyos archivos no cambian de tamaño, como ocurre en los CD-ROM (Norma ISO 9660), DVD o en el Sistema de Archivos de Arranque de UnixWare (*UnixWare Boot FileSystem*)

Encadenada. (*linked allocation*) En esta asignación los bloques físicos en los cuales está almacenado un archivo, pueden estar diseminados por todo el dispositivo de almacenamiento secundario, según se aprecia en la figura 6.4. Los bloques físicos son mayores que los bloques lógicos y almacenan tanto al bloque lógico como a un puntero al bloque físico donde está almacenado el próximo bloque lógico del archivo. La entrada del directorio almacena la ubicación del primer bloque físico. El bloque físico asociado con el *N-ésimo* bloque lógico puede ser determinado sólo mediante la lectura del bloque *N- 1* previo y siguiendo los enlaces (*links*) o eslabones que contienen. El rendimiento de las operaciones para añadir, es decir escribir al final del archivo, se puede mejorar significativamente incluyendo también en la entrada del directorio un puntero al último bloque del archivo.

Este método de almacenamiento no contiguo tiene diferentes formas de implementación. Supongamos que contemos con bloques de 1024 bytes (1K). Una forma posible de asignación sería que de ese bloque de 1024 bytes, 1022 sean para datos y los 2 restantes indiquen cuál es el próximo bloque. Las desventajas que podemos observar es que de esta forma el número de bytes disponibles para datos no es potencia de dos, lo que dificulta la aritmética binaria, y por otra parte, el acceso al azar es costoso de implementar ya que, por ejemplo, para acceder al tercer bloque, debo acceder primero a los dos anteriores. Es decir, supongamos que quiero leer el byte 32768, entonces divido 32768 por 1022 (bytes disponibles en el bloque), el cociente es 32 y el resto 64, por lo tanto, el byte 32768 está dentro del bloque 33, desplazamiento 64. Para poder accederlo deberé leer los 32 bloques anteriores para poder llegar al 32, de donde leeré la dirección del bloque 33 que es el que busco.

Indexada. (*indexed allocation*) Esta asignación es una variante de la asignación encadenada, en la que el bloque físico almacena sólo el bloque lógico y por lo tanto es del mismo tamaño que éste. Los enlaces o eslabones a los bloques físicos del archivo están almacendos de forma contigua en una tabla de índices. La entrada del directorio contiene o bien una tabla de índices o un puntero a ella, como vemos en la figura 6.5.

En la asignación contigua y la encadenada el sistema de archivos sólo necesita almacenar la ubicación física del comienzo del archivo. Todas las otras direcciones se pueden determinar a partir de la ubicación de comienzo del archivo. En la asignación indexada, el sistema de archivos necesita tener una entrada índice para cada bloque en el archivo. Para minimizar la cantidad de espacio requerido en las estructuras de directorios se puede usar la indexación multinivel, como se ilustra en la figura 6.6. En un esquema así, las entradas índices indirectos

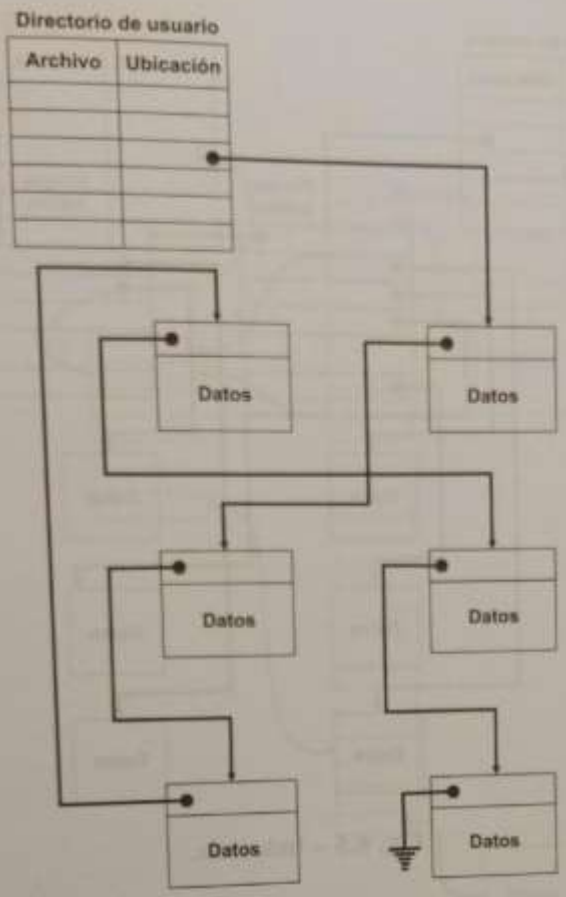


Fig. 6.4 – Encadenada.

apuntan a bloques que almacenan índices adicionales. Pero el crear capas de bloques índices indirectos incrementa el número de entradas de forma exponencial.

Asignador de bloques Orlov. Es un algoritmo para definir dónde residirá un archivo en particular en un sistema de archivos dado, toma su nombre de su creador, Orlov (2000) que publicó una breve descripción e implementación para OpenBSD, que luego fuera utilizada en el sistema de archivos BSD Fast Filesystem.

El rendimiento de un sistema de archivos depende de varios factores y uno de ellos precisamente es cómo ese sistema distribuye los archivos en el disco, si bien en general es mejor mantener cerca la información relacionada y, no obstante lo expresado en 6.1.7.2, podemos observar en un ejemplo un problema muy común, supongamos en un sistema Linux en el que todos los archivos de usuarios están en el directorio `/home`, si todos los archivos de usuario están ubicados unos cerca de otros (es decir, `/home/usuario1` `/home/usuario2` `.../home/usuarioN`) no quedará espacio para el contenido de estos directorios, por lo tanto, los archivos de los usuarios terminan siendo ubicados lejos de los directorios que los contienen, y esto afecta al buen rendimiento. Entonces, el dispersar los directorios en el disco le permite a los archivos de un mismo directorio permanecer más cerca y más o menos contiguos a medida que su número y/o sus tamaños aumentan. El algoritmo de Orlov intenta

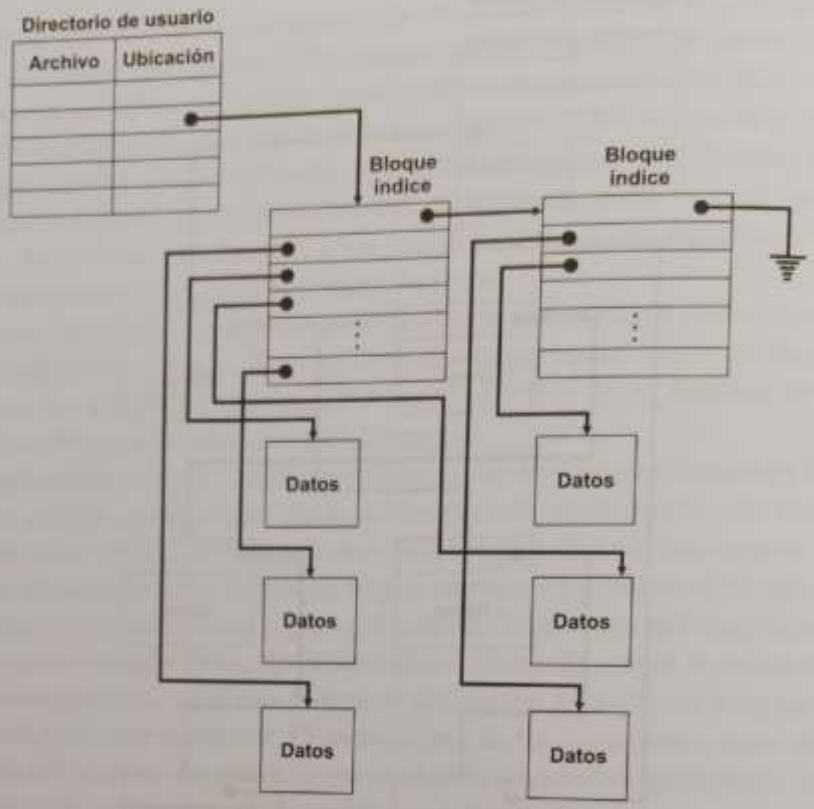


Fig. 6.5 – Indexada.

dispersar los directorios más cercanos a la raíz (*top level directories*) bajo el supuesto de que no están relacionados entre sí.

Asignación demorada. (*delayed allocation*) también llamada *allocate-on-flush*, es una función para el rendimiento, no afecta a la distribución de los datos en el sistema de archivos y consiste, precisamente, en demorar la asignación de bloques lo más posible, contrariamente a lo que hacen tradicionalmente los sistemas de archivos, asignar los bloques tan pronto sea posible. Esta técnica demora la asignación de los bloques cuando el archivo aún se encuentra en caché hasta el momento en que efectivamente vayan a ser escritos a disco. Esto le da al asignador de bloques la oportunidad de optimizar la asignación en situaciones en las que los sistemas tradicionales no podrían hacerlo.

6.2 Implementaciones

6.2.1 Sistemas de archivos en Windows

6.2.1.1 FAT - File Allocation Table

El sistema de archivos FAT (*File Allocation Table*) o Tabla de Asignación de Archivos, debe su nombre a su método de organización porque almacena entradas en una tabla que reside al comienzo del volúmen, tiene sus orígenes a finales de 1970 y comienzos de 1980

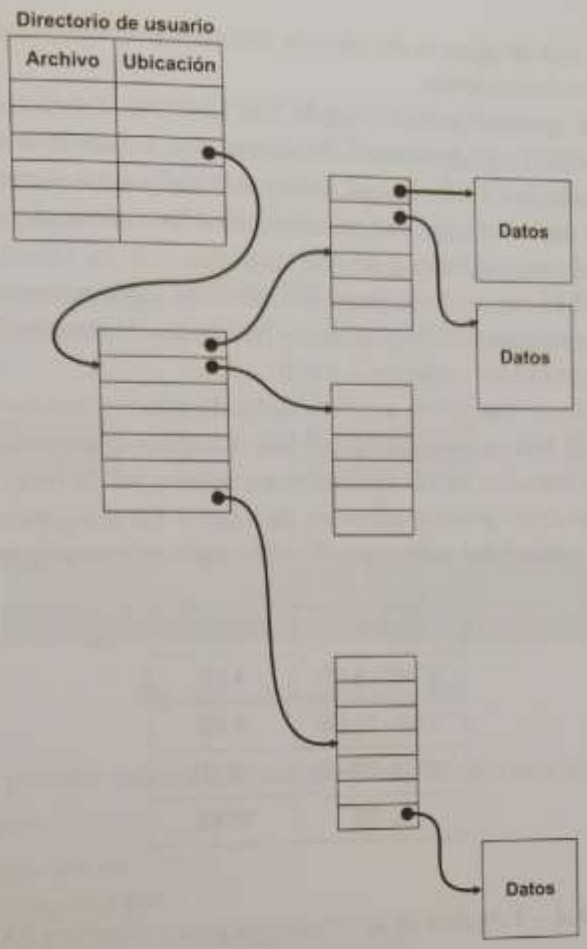


Fig. 6.6 – Indexación multinivel.

y fue el sistema de archivos soportado por el sistema operativo MS-DOS de Microsoft. Fue desarrollado originalmente como un sistema de archivos simple adecuado para dispositivos de discos flexibles con capacidades menores a los 500K. A través del tiempo ha sido mejorado para dar soporte a medios cada vez mayores. Actualmente hay tres tipos de sistemas de archivos FAT: FAT12, FAT16 y FAT32. La diferencia básica entre estos sub tipos de FAT, y el motivo de sus nombres, es el tamaño, en bits, de las entradas en la estructura real de la FAT en el disco, es decir, hay 12 bits en la entrada FAT del FAT12, 16 bits en la entrada FAT de FAT16 y 32 bits en la entrada FAT de FAT32.

La especificación oficial de la FAT de Microsoft está en: `http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx`

Este sistema de archivos se sigue utilizando hasta la fecha en dispositivos *flash* tales como los medios de almacenamiento removibles de las cámaras digitales.

Para proteger el volúmen se mantienen dos copias de la tabla, en caso de que una resulte dañada. Las tablas de asignación de archivos y la carpeta raíz (*root folder*) se deben almace-

nar en una ubicación fija de manera tal que los archivos necesarios para arrancar el sistema puedan ser ubicados correctamente.

El identificador de agrupación de 12 bits de FAT12 limita una partición a almacenar hasta un máximo de 2^{12} (4096) agrupaciones[8]. Windows usa tamaños de agrupaciones que van desde los 512 bytes hasta los 8 KB, lo cual limita al tamaño del volumen de FAT12 a 32 MB, por lo tanto Windows puede utilizar este sistema como formato de discos flexibles de 5 o 3½ pulgadas, que puede almacenar hasta 1.44 MB de datos.

Análogamente FAT16 puede direccionar 2^{16} (65536) agrupaciones pero en Windows los tamaños de agrupaciones para FAT16 van desde 512 bytes (el tamaño de un sector) hasta 64 KB, lo que limita el tamaño del volumen a 4 GB.

FAT32 es el formato de sistema de archivos definido más recientemente. Usa identificadores de agrupación de 32 bits pero reserva los 4 bits más altos de manera que sólo son usables 28 bits y dado que los tamaños de las agrupaciones pueden ser de hasta 32 KB, FAT32 tiene la capacidad teórica de direccionar volúmenes de hasta 8 TB (terabytes). Pero en la práctica Windows limita la capacidad del volumen a 32 GB, según se muestra en el cuadro 6.4.

Partición	Agrupación
32 MB - 8 GB	4 KB
8 GB - 16 GB	8 KB
16 GB - 32 GB	16 KB
32 GB	32 KB

Tabla 6.4 – Tamaños de agrupaciones para volúmenes FAT32.

Además de la ventaja de contar con más agrupaciones, otra ventaja de FAT32 sobre las versiones previas es que el directorio raíz no está almacenado en una ubicación predefinida en el volumen, tampoco tiene un límite superior en su tamaño, y almacena una segunda copia del sector de arranque, para brindar confiabilidad. Pero una limitación que tienen las tres implementaciones es que el tamaño máximo de archivo es de 4 GB porque los directorios almacenan los tamaños de archivos como valores de 32 bits.



Fig. 6.7 – Estructura del volumen FAT.

Un volumen FAT está dividido en varias regiones, que se muestran en la figura 6.7. La primera, el Sector de Arranque de la Partición (*Partition Boot Sector*) contiene información que el sistema de archivos usa para acceder el volumen. En computadoras basadas en arquitectura x86, el Registro de Arranque Maestro (*Master Boot Record - MBR*) usa al Sector de

8. Todos los tipos de sistema de archivos FAT reservan las dos primeras y las últimas 16 agrupaciones de un volumen, de manera que el número de agrupaciones utilizables de un volumen es ligeramente menor

Arranque de la Partición en la partición del sistema para cargar los archivos del núcleo del sistema operativo.

La Tabla de Asignación de Archivos, que le da nombre al sistema, tiene una entrada para cada agrupación en el volumen. Las entradas en la tabla definen cadenas (*chains*) de asignación tanto para archivos como para directorios, en las que los eslabones en la cadena son índices a la próxima agrupación de los datos del archivo, como puede verse en la figura 6.8. La entrada de directorio del archivo almacena la agrupación de comienzo del archivo y la última entrada de la cadena de asignación del archivo es el valor reservado 0xFFFF para FAT16 y 0xFFF para FAT12. Las entradas de la FAT para las agrupaciones sin asignar tienen un valor de 0.

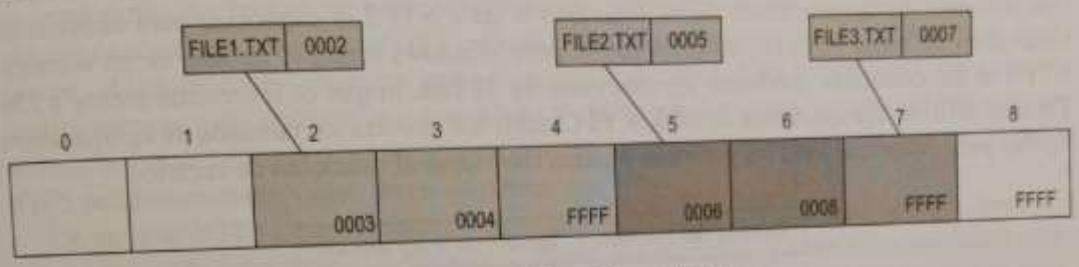


Fig. 6.8 – Asignación en FAT.

Es decir que los posibles valores a encontrar en las entradas para las agrupaciones son:

- sin usar (0x0000)
- asignación usada por un archivo
- asignación dañada (0xFFF7)
- última asignación en un archivo (0xFFF8-0xFFFF)

El método de asignación de archivos que usa FAT es el de la lista encadenada o enlazada (*linked list*) Veamos un ejemplo de asignación:

En la figura 6.8 se muestran tres archivos. El archivo `FILE1.TXT` utiliza tres agrupaciones o unidades de asignación, el segundo archivo `FILE2.TXT` es un archivo fragmentado que también necesita tres agrupaciones, y el archivo `FILE3.TXT` cabe completamente en una sóla unidad.

El directorio raíz de los volúmenes FAT12 y FAT16 tiene espacio suficiente preasignado al comienzo del volúmen, como para almacenar 256 entradas de directorio, lo cual nos pone un límite en el número de archivos y directorios que se pueden almacenar en el directorio raíz (no hay espacio preasignado o límites de tamaño en el directorio raíz de FAT32). Una entrada de directorio en la FAT tiene 32 bytes y almacena el nombre del archivo, el tamaño, la agrupación de comienzo y la fechas de creación, como vemos en la tabla 6.5.

Como vemos en la figura 6.7, en la estructura del volumen FAT tenemos a la carpeta (directorio) raíz. No hay una organización para la estructura de carpetas en la FAT, a los archivos se les da la primera ubicación disponible en el volúmen. El número de comienzo de la agrupación es la dirección del primer *cluster* usado por el archivo.

El problema con el sistema FAT es que los apuntadores de todos los archivos del disco están en la misma tabla. Aunque se abra sólo un archivo, se necesita cargar toda la FAT en memoria.

Nombre	Extensión	Atributos	Reservado	Tiempo	Fecha	1ª agrupación	Tamaño
8 bytes	3 bytes	1 byte	10 bytes	2 bytes	2 bytes	2 bytes	4 bytes

Tabla 6.5 – Entrada de directorio de la FAT.

6.2.1.2 NTFS

El sistema de archivos NTFS es el formato de sistema de archivos nativo de Windows. Usa números de agrupaciones de 64 bits, esto le da a NTFS la capacidad para direccionar volúmenes de hasta 16 exabytes ($16 * 10^{18}$); pero Windows limita el tamaño de un volumen NTFS al direccionable mediante agrupaciones de 32 bits, lo que es levemente menor a 256 TB si se utilizan agrupaciones de 64 KB. El Cuadro 6.6 muestra los tamaños de agrupaciones típicas para volumenes NTFS pero que pueden cambiarse al momento de crearlos.

Tamaño del volumen	Tamaño de la agrupación
512 MB o menor	512 bytes
513 MB a 1024 MB	1 KB
1 GB a 2 GB	2 KB
mayores a 2 GB	4 KB

Tabla 6.6 – Tamaño de agrupaciones para volumenes NTFS.

NTFS también tiene la capacidad de contener $2^{32} - 1$ archivos por volumen. El formato NTFS permite archivos de hasta 16 exabytes de tamaño pero la implementación limita el tamaño máximo de archivo a 16 TB. NTFS incluye un número de características avanzadas tales como la seguridad a nivel de archivo y directorio, cupos (*quota*) en disco, compresión de archivos, enlaces simbólicos basados en el directorio y encriptación.

Una de las características más significativas es la recuperabilidad. Si un sistema se detiene inesperadamente, los metadatos de un volumen FAT pueden quedar en un estado inconsistente, dejando como resultado la corrupción de grandes cantidades de datos de archivos y directorios. NTFS registra los cambios a los metadatos de manera transaccional de manera tal que las estructuras del sistema de archivos pueden repararse a un estado consistente sin pérdida de información en la estructura de archivos o directorios.

Para cumplir con los requerimientos de almacenamiento y acceso confiables a los datos, NTFS provee recuperación del sistema de archivos basados en el concepto de transacciones atómicas (es decir, tiene *journaling*). Si un programa inicia una operación de entrada y salida que altera la estructura de un volumen NTFS, es decir cambios a la estructura de directorios, extender un archivo, asignar espacio a un archivo nuevo, NTFS lo hace mediante una transacción atómica. Además usa almacenamiento redundante para la información vital del sistema de archivos de manera que si un sector del disco se daña, NTFS puede acceder a esos datos

críticos del sistema de archivos del volúmen. A diferencia de lo que ocurre con las estructuras en el disco del sistema de archivos FAT y HPFS, que tiene sectores únicos que contiene datos críticos del sistema de archivos.

La seguridad en NTFS se deriva directamente del modelo de objetos de Windows. Un archivo abierto se implementa como un objeto archivo con un descriptor de seguridad almacenado en el disco como parte del archivo.

Antes de que un proceso pueda abrir un mando (*handle*) a cualquier objeto, inclusive un objeto archivo, el sistema de seguridad de Windows verifica que el proceso tenga la autorización apropiada para hacerlo.

En NTFS, cada unidad de información asociada con un archivo, incluyendo su nombre, su propietario, sus fechas, su contenido y demás, está implementada como un atributo del archivo (atributo de objeto de NTFS). Cada atributo consiste en un flujo (*Alternate Data Stream - ADS*) único, es decir una secuencia simple de bytes. Esta implementación genérica facilita agregar más atributos a un archivo. Y dado que los datos de un archivo son «un atributo más» del mismo y además se le pueden agregar nuevos atributos, los archivos de NTFS pueden contener varios flujos de datos.

Un archivo NTFS tiene un flujo de datos por defecto, el cual no tiene nombre, pero una aplicación puede crear flujos adicionales de datos, con nombres, y accederlos mediante ellos. El nombre del flujo de datos se especifica añadiendo los dos puntos (:) al nombre del archivo, y como los dos puntos es un carácter reservado puede servir como separador entre el nombre del archivo y el nombre del flujo de datos.

Cada flujo tiene un tamaño propio de asignación (el cual define cuánto espacio en disco se le ha reservado), el tamaño real (cuántos bytes ha usado) y la longitud válida de los datos (cuánto del flujo se ha inicializado). Además a cada flujo se le otorga un cerrojo (*lock*) propio que se usa para trabar rangos de bytes y permitir acceso concurrente.

Un componente de Windows que usa múltiples flujos de datos es el soporte de servidor de archivos de Apple Macintosh; Windows Explorer es otra aplicación que utiliza flujos. El software maligno (*malware*) también usa los flujos de datos para ocultar su código.

Otra característica que ofrece NTFS es la reasignación dinámica de las agrupaciones defectuosas (*dynamic bad-cluster remapping*). Si el volumen ha sido creado con tolerancia a fallos, el manejador de la tolerancia a fallos extrae dinámicamente una copia buena de los datos que fueron almacenados en el sector dañado y luego envía una advertencia a NTFS indicando que el sector está dañado.

NTFS también utiliza enlaces. Los enlaces permiten que distintas rutas se refieran al mismo archivo (pero no funciona con directorios) de manera similar a como ocurre en los sistemas Unix. Además NTFS tiene otro tipo de redirección llamado **unión** (*junction*). No se pueden usar uniones para enlazar directorios remotos, sólo para directorios en volúmenes locales. Las uniones están basadas en un mecanismo de NTFS llamado (*reparse points*), que es un archivo o directorio que tiene un bloque asociado a él llamado (*reparse data*), que son datos definidos por el usuario acerca del archivo o directorio, tales como su estado o ubicación, que pueden ser leídos desde el *reparse point* o bien por la aplicación que creó los datos, un manejador filtro del sistema de archivos, o el administrador de la entrada y salida. No deben confundirse los enlaces con los «accesos directos» (*shortcut*), que son archivos con extensión `.lnk`. A partir de Windows Vista, NTFS también tiene enlaces simbólicos (*symbolic link*)

que, a diferencia de las uniones, pueden también apuntar a un archivo o directorio remotos, siempre y cuando el sistema remoto los soporte.

6.2.2 Sistema de archivos de UNIX

Los orígenes del sistema de archivos de UNIX datan del año 1969, según el mismo Dennis Ritchie relata en *The Evolution of the Unix Time-sharing System*:

> «Además de las turbulencias financieras que tuvieron lugar en 1969, también había trabajo técnico. Thompson, R. H. Canady, y Ritchie desarrollaron, en la pizarra y notas manuscritas, el diseño básico de un sistema de archivos que posteriormente se convertiría en el corazón de Unix. La mayor parte del diseño fue de Thompson, como lo fue el impulso para pensar en sistemas de archivos, pero creo que yo contribuí la idea de archivos de dispositivos. El deseo vehemente de Thompson para la creación de un sistema operativo tomó diversas formas durante este período; también escribió (en Multics) una simulación muy detallada del rendimiento del diseño del sistema de archivos propuesto y del comportamiento de la paginación de los programas». ... «Pronto Thompson comenzó a implementar el sistema de archivos de papel (quizás "el sistema de archivos de tiza" sería más preciso) que había sido diseñado con anterioridad. Un sistema de archivos sin una forma de ejercitarlo es una propuesta estéril ...» (D. Ritchie 1980).

Estructuralmente, el sistema de archivos del Unix de la PDP-7 era prácticamente idéntico al actual; comprendía una «lista-i (*i-list*)», un arreglo lineal de nodos-i que describía cada uno a un archivo. Un nodo-i que contenía menos información que la que contiene actualmente, pero en esencia era lo mismo, el modo de protección de los archivos, su tipo y tamaño, y una lista de los bloques físicos que mantenían el contenido del archivo. Los directorios eran una clase especial de archivo que contenía una secuencia de nombres con su número de nodo-i asociado y archivos especiales que describían a dispositivos. D. Ritchie 1980

Posteriormente, la versión 4.2BSD desarrollada por el *Computer Systems Research Group - CSRG*, incluía unos pocos bloques al comienzo de la partición reservados para **bloques de arranque** (*boot blocks*) que debían ser inicializados separadamente del sistema de archivos. También incluía al **superbloque** (*superblock*) que contenía el número mágico que lo identificaba como sistema de archivos, como así también información acerca del tamaño del sistema de archivos, el número de bloques libres en el sistema, una lista de los bloques libres, el número de nodos-i libres, una lista de nodos-i libres, entre otros, las distintas implementaciones de Unix varían. Y muchos de estos valores son configurables al momento de la creación del sistema de archivos.

Los nodos-i se numeran de manera secuencial y los primeros nodos-i están reservados por motivos históricos, por ejemplo el nodo-i del directorio raíz es siempre 2, de manera que es fácil encontrarlo al momento de montarlo. Los subdirectorios se implementan almacenando el nombre del subdirectorio en el campo del nombre y el número de nodo-i en el campo para el nodo-i. Los **enlaces duros** (*hard links*) se implementan almacenando el mismo número de nodo-i con otro nombre de archivo, entonces al acceder al archivo por cualquiera de sus

nombres, accedemos al mismo número de nodo-i y, por lo tanto, a sus datos. Los directorios especiales «.» y «..» se implementan almacenando sus nombres «.» y «..» en el directorio y el número de nodo-i del directorio actual y el del padre respectivamente en el campo para el nodo-i. El único tratamiento especial que reciben estas dos entradas es que se crean de forma automática cada vez que se crea un directorio nuevo y no se pueden borrar.

Según decíamos en 6.1.1.2 el nodo-i (*index node* o *i-node*) de UNIX contiene información acerca del tipo de archivo (normal o regular, directorio, enlace simbólico o *link*, tubería o *pipe*, archivo dispositivo, *socket*), permisos o modos de acceso, el propietario, el grupo propietario, tamaño, fechas de creación, de último acceso y de última modificación, y punteros a bloques de datos. La estructura de punteros del nodo-i es una estructura adoptada por el nodo-i de un archivo en los sistemas de archivos tipo UNIX para listar las direcciones de los bloques de datos del archivo. En el pasado la estructura consistía de 13 punteros, pero las implementaciones modernas usan 15, pero básicamente la estructura consiste de **punteros directos** que apuntan directamente a los bloques de datos, **puntero indirecto simple** que apuntan a un bloque de punteros directos, que a su vez apuntan a los bloques de datos, **puntero indirecto doble** que apunta a un bloque de punteros indirectos simples, que se comportan como ya vimos Bach (1986). Y, en las implementaciones más modernas, **puntero indirecto triple** que apunta a un bloque de punteros dobles .

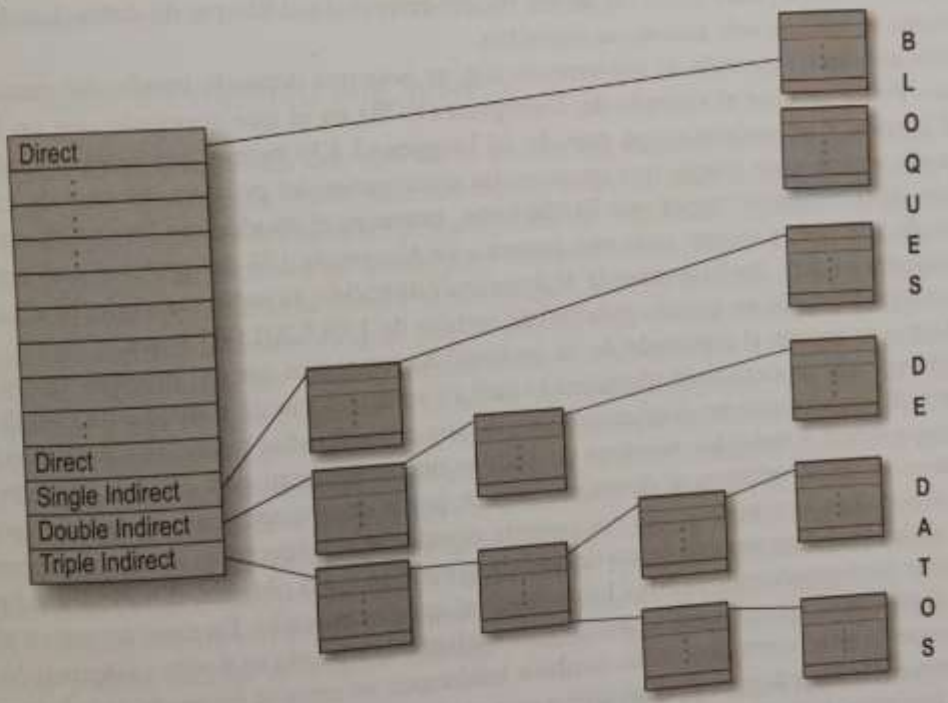


Fig. 6.9 – Estructura de punteros de los nodos-i.

Supongamos en el esquema tradicional del UNIX que contamos con 10 punteros directos y 3 indirectos y que nuestros bloques sean de 1k. Si el archivo tiene 10 K, los bloques del archivo serán accedidos a partir de las direcciones directas que están en el nodo-i. El nodo-i

esta en disco, por lo tanto, hay que leer primero el nodo-i y desde allí leer las direcciones para acceder a los bloques. Cuando el archivo crece a un bloque más, se le asigna un bloque de disco y se coloca un apuntador indirecto. El bloque asignado es un bloque que tendrá direcciones, y cada una de estas referenciará a un bloque donde están los datos que continúan. Si las direcciones son de 32 bits, en un bloque de 1024 bytes, entrarán $\frac{1024*8}{32}$ direcciones de disco, es decir 256.

Si uso los 10 punteros directos más uno indirecto, puedo direccionar 266 bloques de disco (10 directos más 256 direcciones del indirecto). Si el archivo mide más de 266 K, cuando necesito agregar bloques al archivo tendré que usar el puntero indirecto doble. Esta dirección apunta a un bloque cuyo contenido son 256 direcciones de 32 bits. Cada una de estas direcciones referencian un bloque cuyo contenido son 256 direcciones de 32 bits. Cada dirección referencia, ahora si, el bloque de datos que agregan al archivo.

De esta manera, con dos niveles de indirección logro tener un archivo de $10+256+(256*256)$ K, 65802 K sumando los bloques directos, los indirectos y los indirectos dobles.

Si el archivo mide más que este tamaño, se debe usar el tercer nivel de indirección o puntero indirecto triple. Esta dirección apunta a un bloque cuyo contenido son 256 direcciones de 32 bits. Cada una de estas direcciones apuntan a un bloque cuyo contenido son 256 direcciones de 32 bits. Cada dirección referencia un bloque cuyo contenido son 256 direcciones de 32 bits. Ahora sí, cada dirección de ese bloque referencia el bloque de datos. Los bloques indirectos se utilizan sólo cuando se necesitan.

Este es sólo un ejemplo de implementación, se nota que depende mucho del tamaño del bloque. Podemos citar el ejemplo de Thompson (1978) en el que contando con un bloque de 512 bytes, si el archivo ocupa más de 10 bloques (5.120 bytes), entonces la undécima dirección apunta a un bloque que contiene las direcciones del próximo bloque de 128 del archivo. Si el archivo es mayor que 70.656 bytes, entonces el duodécimo bloque apunta a un bloque de 128 que, a su vez, cada uno apunta a un bloque de 128 del archivo. Los archivos aún mayores a 8.459.264 bytes usan la decimotercer dirección, la indirecta triple. El algoritmo finaliza aquí logrando un tamaño máximo de archivo de 1.082.201.087 bytes.

¿Como se accede al contenido de un archivo? Supongamos que un proceso quiere abrir un archivo, cada proceso tiene almacenado cuál es su directorio de trabajo actual, todos los nombres de archivo que no comienzan con «/» son considerados relativos a este directorio de trabajo actual y todos los nombres de archivo que comienzan con «/» son considerados absolutos, es decir, relativos al directorio raíz. A partir de ahí comienza el trabajo recursivo de ir accediendo al nodo-i, que nos permite acceder al subdirectorio si tenemos permiso, luego abrir el subdirectorio en busca del próximo nombre, para obtener el nodo-i y así sucesivamente atravesando todo el árbol hasta llegar al archivo buscado. En caso de que en algún punto el proceso no tenga permiso para seguir adelante, la travesía se detiene informando con el mensaje de error correspondiente, también tendremos un error si luego de toda la travesía no tenemos permiso de acceso al archivo o éste ni siquiera existe.

Si logramos llegar a este punto exitosamente, accederemos al contenido del archivo a través de sus nodos-i.

Podemos preguntarnos cuántos accesos a disco fueron necesarios para acceder a los datos que queríamos. Podríamos suponer un máximo de dos por subdirectorio, uno para buscar el nombre y otro para encontrar el nodo-i, y luego dos más para el nombre de archivo mismo. Este esfuerzo sólo se hace al abrir el archivo, ya que una vez abierto los accesos siguientes

pueden usar los datos del nodo-i sin buscarlo de nuevo. Además, las técnicas de **caching** eliminan muchos de los accesos necesarios para buscar un archivo.

6.2.3 Sistemas de archivos en Linux

En sus primeros días, Linux era desarrollado sobre el sistema operativo Minix (Tanenbaum y Woodhull 2005) y, lógicamente, el sistema de archivos que era capaz de acceder era ése mismo. Recordemos que el objetivo de Minix es didáctico, por lo tanto había dos características de él que se percibían como limitaciones entre los desarrolladores de Linux, a saber, por una parte las direcciones de los bloques se almacenan en enteros de 16 bits, por lo tanto el tamaño máximo de todo el sistema de archivos queda limitado a 64 MB y por otra, los directorios contienen entradas de tamaño fijo y la longitud máxima del nombre de archivo es de 14 caracteres.

Por estos motivos, Rémy Card se abocó al desarrollo del Sistema de Archivos Extendido (*Extended File System, extfs*) y al Segundo Sistema de Archivos Extendido (*Second Extended File System, ext2fs*) y con el objetivo de facilitar el agregado de nuevos sistemas de archivos dentro del núcleo de Linux, fue desarrollada la capa del Sistema de Archivo Virtual (*Virtual File System - VFS*), inicialmente por Chris Provenzano y posteriormente reescrita por Linus Torvalds antes de que fuera integrada al núcleo de Linux.

6.2.3.1 Second Extended File System

Después de la integración del VFS en el núcleo fué implementado el `extfs` en Abril de 1992 y agregado a Linux 0.96c. Este sistema de archivos tenía un tamaño máximo de 2 GB y la longitud máxima del nombre de archivo era de 255 caracteres. Fue una mejora con respecto al sistema de archivos de Minix pero aún presentaba problemas. El sistema usaba listas encadenadas para llevar la cuenta de los bloques libres y los nodos-i y esto daba como resultado un mal rendimiento ya que, con el uso, la lista se desordenaba y el sistema de archivos se fragmentaba. Por esto, se intentaron dos soluciones, por una parte el sistema de archivos Xia y por otra, el Segundo Sistema de Archivos Extendido, con muchas mejoras sobre el `extfs`. Xia era más estable pero, con el uso, se repararon los defectos de `ext2fs` y fué el sistema adoptado finalmente como estándar *de facto* en Linux. El sistema de archivos tenía un tamaño máximo de 4 TB y el tamaño máximo de archivo era de 2 GB.

Cada sistema de archivos de Linux implementa un conjunto básico de conceptos comunes derivados del sistema operativo Unix (Bach 1986). Los archivos están representados por nodos-i, los directorios son simples archivos que contienen una lista de entradas y los dispositivos se pueden acceder solicitando entrada y salida sobre archivos especiales (Card, Ts'o y S. Tweedie 1994)

Como decíamos en 6.2.2, distintos nombres pueden estar asociados al mismo número de nodo-i, es lo que llamamos enlace, el nodo-i contiene un campo que contiene el número asociado con el archivo. Al crear un enlace incrementamos un contador de enlaces en el nodo-i y al borrarlo lo decrementamos. Este tipo de enlace sólo puede apuntar a archivos que, además, deben estar dentro del mismo sistema de archivos. Estos enlaces no pueden apuntar a directorios para prevenir la aparición de ciclos en el árbol. También tenemos los enlaces simbólicos o blandos (*soft links*) que son archivos que contienen un nombre de archivo. Cuando el nú-

cleo encuentra un enlace simbólico en la ruta para obtener un nodo-i, reemplaza el nombre del enlace por su contenido, es decir el nombre del archivo apuntado, y reanuda la interpretación de la ruta. Dado que un enlace simbólico no apunta a un nodo-i, es posible crear enlaces simbólicos entre sistemas de archivos, además pueden apuntar a cualquier tipo de archivo, aún archivos inexistentes.

En sistemas operativos tipo Unix los dispositivos pueden accederse a través de archivos especiales, estos archivos no usan espacio en el medio de almacenamiento ya que sólo son un punto de acceso al manejador de dispositivo. Existen dos tipos de archivos especiales, los orientados al carácter y los orientados al bloque. Los primeros permiten operaciones de entrada y salida en modo caracter mientras que los últimos requieren que los datos sean escritos en forma de bloque a través de las funciones de memoria caché (*buffer cache*). Cuando se realiza una solicitud de entrada o salida a un archivo especial, es redirigida a un pseudo manejador de dispositivo. Un archivo especial está referenciado por un número mayor (*major number*) que identifica al tipo de dispositivo, y un número menor (*minor number*) que identifica a la unidad.

6.2.3.2 Virtual File System

La capa del VFS del núcleo de Linux se usa durante las llamadas a sistema que actúan sobre archivos, es una **capa de indirección** que maneja las llamadas a sistema orientadas a archivos y llama a las funciones necesarias en el código del sistema de archivo físico para realizar la entrada y salida. El mecanismo de indirección se ha usado en sistemas operativos tipo Unix para facilitar la integración y el uso de varios tipos de sistemas de archivos (Kleiman 1986; McKusick y col. 1984). Cuando un proceso efectúa una llamada a sistema orientada a operar un archivo, el núcleo llama a una función contenida en el VFS. Esta función se encarga de las manipulaciones independientes de la estructura y redirige la llamada a una función contenida en el código del sistema de archivos físico, el cual es responsable de encargarse de las operaciones dependientes de la estructura. El código del sistema de archivos usa las funciones de memoria caché para solicitar la entrada y salida a los dispositivos, como se ve en la figura 6.10

El VFS define un conjunto de funciones que todo sistema de archivos debe implementar. Esta interfaz está hecha de un conjunto de operaciones asociadas a tres clases de objetos: sistemas de archivos, nodos-i y archivos abiertos. El VFS conoce cuales son los distintos tipos de sistemas de archivos implementados en el núcleo, utiliza una tabla definida durante la configuración del mismo, cada entrada en esta tabla describe un tipo de sistema de archivos, contiene el nombre del tipo de sistema de archivos y un puntero en una función llamada durante la operación de montaje. Cuando se debe montar un sistema de archivos, se llama a la función de montaje apropiada. Esta función es responsable de leer el superbloque del disco, inicializar sus variables internas y devolver al VFS un descriptor de sistema de archivo montado. Después de que el sistema de archivos ha sido montado, las funciones del VFS pueden usar este descriptor para acceder las rutinas del sistema de archivos físico. Un descriptor de sistema de archivos montado contiene distintos tipos de datos, tales como información común a todos los tipos de sistema de archivos, punteros a funciones provistas por el código en el núcleo del sistema de archivos físico, y datos privados mantenidos por el código del sistema de archivos físico. Los punteros de la función contenida en el descriptor del sistema

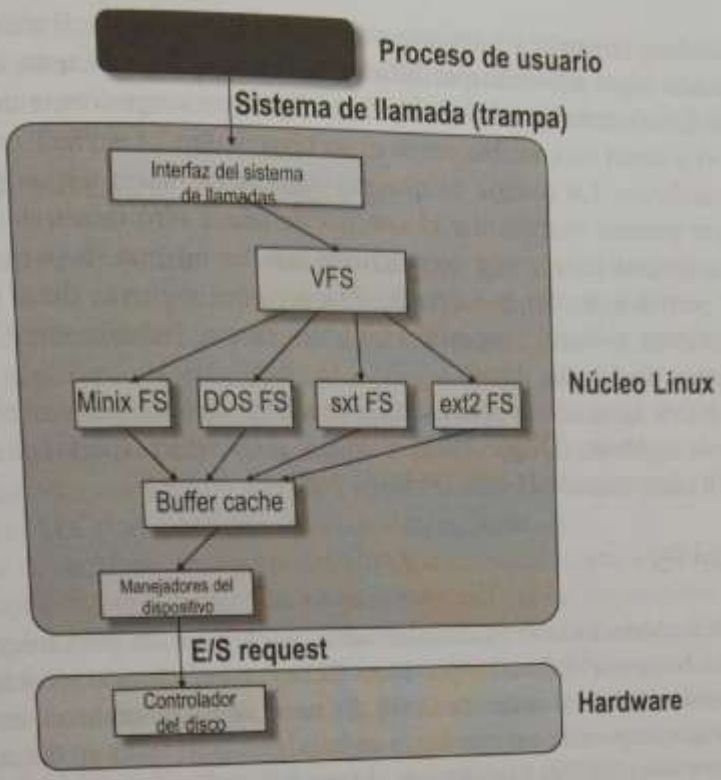


Fig. 6.10 – Sistema de Archivos Virtual.

de archivos le permite al VFS acceder a las rutinas internas del sistema de archivos. El VFS usa otros dos tipos de descriptores, un descriptor de nodo-i y un descriptor de archivo abierto, cada descriptor contiene información relacionada con archivos en uso y un conjunto de operaciones provistas por el código del sistema de archivos físico. Si bien el descriptor de nodo-i contiene referencias a las funciones que se pueden utilizar para actuar en cualquier archivo, los descriptores de archivo contienen punteros a funciones que solo pueden actuar sobre los archivos abiertos.

6.2.3.3 Third Extended File System

A pesar de la robustez de `ext2fs`, persistían algunos problemas tales como la disponibilidad, la integridad de los datos y la velocidad, que tienen como origen en común los malos apagados del sistema, cada vez que se le interrumpe bruscamente el suministro eléctrico al *hardware*, a los sistemas de archivos `ext2fs` no se los podía volver a montar nuevamente sin antes hacerle una extensa (y a veces intensa) comprobación, y este tiempo de comprobación está en relación directa con el tamaño del sistema de archivos y la cantidad de archivos que contenga. A menudo luego del mal apagado se perdían datos a pesar de que `e2fsck`, la herramienta de comprobación de `ext2fs`, hubiera hecho su mejor trabajo.

S. C. Tweedie (1998) comenzó a trabajar sobre `ext2fs` en 1998 para agregarle capacidades de *journaling*, tarea que pudo ser integrada al núcleo de Linux en Noviembre de 2001 con la versión 2.4.15 y no sólo logró abordar estos problemas sino que lo hizo agregando un

bloque de *journaling* al `ext2fs` ya existente, lo que otorga una fácil transición de `ext2fs` a `ext3fs`. Además logró **retrocompatibilidad**, es decir que un sistema `ext3fs` puede ser utilizado por un Linux anterior a la versión 2.4.15 porque simplemente no utiliza el bloque de *journaling*, el *journal* será visible como el archivo oculto `.journal` en el directorio raíz del sistema de archivos. La ventaja de que sea compatible hacia adelante y hacia atrás no es menor, ya que permite transformar el sistema de uno a otro mientras se lo está usando, y las llamadas a sistema para operar con archivos son las mismas. Soporta varios modos de *journaling* que permiten decidir qué tipo de transacciones registrar, datos y/o metadatos, lo que permite encontrar distintos rangos y soluciones de compromiso entre la velocidad y la consistencia e integridad de los datos.

Además de haber agregado la ventaja del *journaling*, `ext3fs` permite el crecimiento *on-line* del sistema de archivos, (Dilger 2002), y utiliza una versión especial de **árbol-B** (B-tree) (D. Knuth 1997b) denominado **H-tree** (Phillips 2001).

6.2.3.4 ReiserFS

El sistema de archivos `ReiserFS` fue diseñado e implementado por un equipo de desarrollo de la empresa Namesys[9] liderados por Hans Reiser, e introducido en el núcleo de Linux a partir de la versión 2.4.1 en enero de 2001. Es un sistema de archivos caracterizado por utilizar *journaling*, que puede contener hasta un máximo de $2^{32} - 3$ archivos, en un sistema que puede llegar a tener 16 TiB (Tebibits) y el tamaño máximo que puede llegar a tener un archivo es de $2^{31} - 1$, es decir, 2 GiB- 1 en la versión 3.5, y de 1 EiB (Eibit) en la versión 3.6. También soporta listas de control de acceso (*Access Control Lists - ACLs*).

Los desarrolladores de `ReiserFS` se focalizaron en lograr un buen rendimiento del sistema con los archivos chicos, al notar que éste era un punto débil de `ext2fs`, que manejaba bien archivos mayores a 20 kb, pero no es ideal para almacenar 2000 archivos de 50 bytes[10] porque asigna espacio en trozos de uno o cuatro kB. La solución aportada por quienes utilizaban `ext2fs` era desarrollar algún tipo de base de datos que ejecutara sobre el sistema de archivos, lo cual, según la visión de Hans Reiser indicaba que el sistema no satisfacía las necesidades de los usuarios. Y `ReiserFS` es de ocho a quince veces más rápido que `ext2fs` cuando maneja archivos menores a 1 kB y no lo hace a expensas del manejo de archivos mayores a ese tamaño. Para lograr este rendimiento, `ReiserFS` utiliza un árbol balanceado B* (*B*-tree*) optimizado especialmente para organizar todos los datos del sistema de archivos lo cual, entre otras ventajas, permite asignar nodos-i de forma dinámica a medida que sean necesarios en lugar de estar limitado a un conjunto fijo de nodos-i definidos al momento de la creación del sistema de archivos. A diferencia de `ext2fs`, `ReiserFS` no reserva espacio de almacenamiento en bloques fijos de uno o cuatro kB sino que puede asignar el tamaño exacto que necesita. Además incluye una optimización especial alrededor de lo que se denomina **colas** (*tails*), un nombre para archivos y porciones finales de archivos que son menores que un bloque, `ReiserFS` es capaz de almacenar archivos dentro de las hojas mismas de los nodos del árbol-B* en lugar de almacenar los datos en otro lugar del disco y apuntarlos. Esto mejora notablemente el rendimiento porque normalmente pueden accederse en una única operación de entrada o salida.

9. Inactiva desde fines de 2007
10. Por ejemplo, los archivos almacenados en la memoria caché de un gestor o *proxy* siguen este patrón

En el año 2004 hace su aparición `Reiser4` que tiene un manejo más eficiente del *journaling* respecto de su predecesor, como también es más eficiente en el manejo de archivos pequeños gracias a la **subasignación de bloques** para el manejo de colas (Reiser 2002).

6.2.3.5 Fourth Extended File System

El Cuarto Sistema de Archivos Extendido (*Fourth Extended File System - ext4fs*) es la siguiente versión de `ext3fs` y tiene compatibilidad con éste, hacia atrás y hacia adelante, incorporado en forma estable[11] en el núcleo de Linux en diciembre de 2008 a partir de la versión 2.6.28 y mejorando la escalabilidad y confiabilidad, además de agregar nuevas funcionalidades. El direccionamiento del bloque es de 48 bits, por lo tanto el sistema de archivos puede llegar a ocupar hasta 1 EB[12] y el tamaño máximo que puede llegar a tener un archivo es de 16 TB, el número máximo de sub directorios por directorio que caben en `ext3fs` es de 32000 pero en `ext4fs` el número de subdirectorios es ilimitado.

Los sistemas de archivos derivados del UNIX tradicional, como el `ext3fs`, usan un esquema de correspondencia indirecta de bloques para realizar un seguimiento de cada bloque usado respecto a los bloques correspondientes a los datos de un archivo. Esto es ineficiente para archivos grandes, especialmente en las operaciones de borrado y truncado de archivos grandes porque se debe mantener una entrada para cada bloque individual y, como los archivos grandes ocupan muchos bloques, la tabla que mantiene la correspondencia también es grande y esto afecta a la velocidad. En cambio se utiliza un enfoque diferente basado en las **extensiones** (*extents*) que es un grupo de bloques físicos contiguos y que básicamente indica que «los datos están en los próximos *n* bloques», por ejemplo, un archivo de 100 MB puede estar asignado a una única extensión de ese tamaño en lugar de necesitar crear la tabla de correspondencia indirecta para 25600 bloques (4 KB por bloque). Los archivos muy grandes se dividen en varias extensiones. Las extensiones mejoran el rendimiento y ayudan a reducir la fragmentación, dado que la extensión alienta el diseño contiguo en el disco.

Cuando `ext3fs` necesita escribir datos nuevos al disco, hay un asignador de bloques que decide qué bloques libres se usarán para escribir los datos. Pero el asignador de bloques de `ext3fs` sólo asigna un bloque de 4 KB por vez. Eso significa que si el sistema necesita escribir los 100 MB de datos a los que nos referíamos, necesitará invocar al asignador de bloques 25600 veces. Esto no sólo es ineficiente sino que tampoco le permite al asignador de bloques optimizar la política de asignación porque no conoce cuantos datos se asignarán en total, sólo conoce acerca de un único bloque, en cambio, `ext4fs` utiliza un **asignador multibloques** que asigna varios bloques en una única llamada a la rutina en lugar de un único bloque por llamada evitando mucha sobrecarga. Esto mejora notablemente el rendimiento y es particularmente útil con la asignación demorada (ver 6.1.7.2) y las extensiones (Kumar, Mingming Cao y Santos 2008).

Otra mejora que agrega `ext4fs` es la suma de comprobación (*checksum*) del *journal*, ya que siendo una parte muy utilizada del sistema, si ocurre un fallo de *hardware* en los bloques que forman parte de él, el sistema de archivos correrá el riesgo de corromperse seriamente.

11. La versión presente a partir del núcleo 2.6.19 no era estable
12. 1 EB=1024 PB, 1 PB=1024 TB, 1 TB=1024 GB

6.2.3.6 Sistemas de archivos sin almacenamiento

Es muy común en Linux la utilización de sistemas de archivos que no se utilizan en medios de almacenamiento masivo, es decir que no se utilizan sobre ningún dispositivo de almacenamiento. Desde los comienzos de Unix ha existido la filosofía «todo es un archivo» y, así, se han modelado muchos recursos como si fueran archivos para poder utilizar en estos recursos las mismas llamadas a sistema que se usan en los archivos, simplificando enormemente el modelo del sistema operativo. Linux ha heredado esta filosofía, la ha hecho propia y la ha extendido en gran manera, como veremos a continuación. Por tratarse de sistemas de archivos, los mencionaremos brevemente acá, pero los desarrollaremos en los capítulos del libro en los que corresponda y haciendo referencia a esta parte.

6.2.3.6.1 procfs

Como habíamos adelantado en 1.5.1.1, el sistema de pseudo archivos[13] `proc` tiene su origen en Killian (1984) «Processes as Files» y su objetivo era el de mostrar información acerca de los procesos en ejecución. En Linux, actúa como una interfaz a estructuras internas de datos en el núcleo, se puede usar tanto para obtener información del sistema como para cambiar ciertos parámetros del núcleo durante su ejecución. Contiene, entre otras cosas, un directorio para cada proceso en ejecución cuyo nombre es el número identificador del proceso (PID), un enlace llamado *self* que apunta al proceso que está accediendo al sistema de archivos `proc` en ese instante. Muestra el uso que hace de la memoria cada proceso y sus estados. De manera similar también muestra información acerca del núcleo en ejecución, asignación de la memoria, información del procesador, sistemas de archivos, uso de los puertos de entrada y salida y de las interrupciones, controladoras de los dispositivos periféricos conectados, información de los dispositivos de red, así como también del tráfico en la misma y los distintos parámetros de configuración. Una parte muy importante de `/proc` es el directorio `/proc/sys` el cual no sólo ofrece información muy valiosa sino también que es modificable. Al cambiar los valores contenidos dentro de los pseudo archivos se cambian los parámetros de operación del núcleo de forma inmediata, mientras permanece ejecutándose.

6.2.3.6.2 debugfs

De manera similar a `procfs` existe `debugfs` que les facilita a los programadores depurar el núcleo, pero a diferencia de `procfs` que sigue la regla de almacenar un valor por archivo de forma más o menos organizada, `debugfs` no tiene reglas, los directorios y archivos se crean mediante declaraciones dentro del código fuente del núcleo, en las partes en las que el programador necesite visualizar cierta información para depurar su código. Su autor es Greg Kroah-Hartman.

13. Se prefiere esta expresión a la de «archivos virtuales» para no confundirla con VFS

6.2.3.6.3 sysfs

De acuerdo con su autor Patrick Mochel, `sysfs` es una prestación del núcleo Linux versión 2.6, que le permite al código del mismo exportar información a procesos de usuario a través de un sistema de archivos en memoria. La organización de la jerarquía de directorios del sistema de archivos es estricta y está basada en la organización interna de las estructuras de datos del núcleo. Los archivos que se crean en el sistema de archivos son, en su mayoría, de tipo ASCII y, a menudo, con un valor por archivo. Estas características aseguran que la información exportada sea precisa y accesible fácilmente, logrando que el `sysfs` sea una de las prestaciones más intuitivas y útiles de la versión 2.6 del núcleo Linux (Mochel 2003b).

`sysfs` muestra toda la información referida a los manejadores, colectores y clases de los dispositivos presentes en el sistema. También muestra mucha otra información que no está directamente relacionada con los manejadores, información que incluye a todos los tipos de sistemas de archivos, todos los dispositivos de bloques, la información de la CPU y hasta una lista de las primitivas de *firmware* presentes en el sistema. Con el tiempo, mucha de la información en el sistema de archivos `/proc` que no tenga que ver con los procesos, que fue la finalidad inicial, será totalmente desplazada a `sysfs`.

`sysfs` es un mecanismo para la representación de objetos del núcleo, sus atributos y sus relaciones entre ellos. Provee dos componentes: una interfaz de programación del núcleo para exportar estos ítems a través de `sysfs` y una interfaz de usuario para ver y manipular estos ítems que se corresponden con los objetos del núcleo que representan, según podemos ver en la tabla siguiente:

Interna	Externa
Objetos del núcleo	Directorios
Atributos del objeto	Archivos normales
Relaciones del objeto	Enlaces simbólicos

Tabla 6.7 – Correspondencia entre estructuras del núcleo y sistema de archivos.

Por tratarse de un sistema de archivos muy relacionado con el sistema de entrada y salida, volveremos a tratarlo en el capítulo correspondiente.

6.2.3.6.4 ramfs

Es un sistema de archivos muy simple que exporta los mecanismos de caché de discos de Linux (la caché de páginas y la caché de entradas de directorio o *dentry*) como un sistema de archivos basado en RAM de tamaño variable y dinámico. Normalmente Linux mantiene a los archivos en uso almacenados en memoria caché. Las páginas de datos leídas de *backing store* (generalmente el dispositivo por bloques en el que está montado el sistema de archivos) se mantienen en caso de que se necesiten de nuevo pero marcadas como «limpias» (es decir, liberables) en caso de que el sistema de memoria virtual necesite de esa memoria para algo más. De manera similar, los datos escritos a archivos se marcan como «limpios»

tan pronto como sean escritos a *backing store* pero se los mantiene en memoria caché hasta que el sistema de memoria virtual reasigne la memoria. Un mecanismo similar (la caché *dentry*) acelera en gran medida el acceso a los directorios. Con `ramfs` no hay *backing store*, los archivos escritos a `ramfs` asignan caché de páginas y *dentry* como siempre pero no hay dónde escribirlos. Esto significa que las páginas nunca se marcan como «limpias» de manera que no pueden ser liberadas por el sistema de memoria virtual cuando está buscando reciclar memoria. La cantidad de código requerido para implementar `ramfs` es muy pequeña porque todo el trabajo está hecho por la infraestructura de almacenamiento de caché de Linux, básicamente se monta la caché de discos como un sistema de archivos. El viejo mecanismo de **disco RAM** (*RAM disk*) creaba un dispositivo de bloques sintético dentro de un área de RAM y lo utilizaba como sistema de archivos, este dispositivo por bloques tenía un tamaño fijo, de manera que el sistema de archivos también. El uso de un disco RAM también requería copiar innecesariamente memoria desde el falso dispositivo por bloques a la caché de páginas y a la inversa, así como crear y borrar entradas de directorio. Además necesitaba un manejador para el sistema de archivos (tal como el `ext2`) para poder acceder a los datos. Esto desperdiciaba memoria y le creaba un trabajo innecesario a la CPU.

6.3 Trabajos prácticos

6.3.1 Particiones y sistemas de archivos

Para ofrecer una imagen única del sistema, el sistema operativo debe ofrecer servicios que permitan asociar, y desasociar, unos sistemas de otros de forma transparente en un árbol de nombres único. Además, las utilidades de interpretación de nombres deben ser capaces de saltar de un sistema de archivos a otro sin que sea aparente en ningún momento el nombre del dispositivo físico o lógico que almacena el sistema de archivos. Las dos llamadas al sistema de UNIX que realizan estas operaciones son `mount` y `umount`. La operación de montado permite conectar un sistema de archivos, almacenado en un volumen o partición, a una entrada de directorio del árbol de directorios existente.

Por ejemplo, si en Linux colocamos el comando:

```
$ mount
```

Nos muestra los volúmenes y particiones montadas en el sistema (la primera columna) y el directorio en donde han sido conectados, denominado «punto de montaje» (la segunda columna), también podemos ver el tipo de sistema de archivos que dicho volumen o partición tiene, por ejemplo: vfat para sistemas de archivos de Windows 95 ó 98, ntfs para sistemas de archivos de Windows NT, ext2 ó ext3 (Extended Second, Extended Third) para sistemas de archivos de Linux. Y además algunas opciones del montado. Por ejemplo: un sistema de archivos iso9660, propio de los CD-ROM, estará montado como «sólo lectura» o *read-only*. En cambio otros pueden estar montados para lectura/escritura o *read-write*.

Los comandos mount y umount tienen muchas opciones, puede verlas en la documentación en línea, con el comando:

```
$ man mount
$ man umount
```

Estos comando toman las opciones por defecto del archivo `/etc/fstab` que contiene una descripción de los distintos volúmenes/particiones, con sus respectivos puntos de montaje, tipo de sistema de archivo y opciones adicionales. Véalo con el comando:

```
$ cat /etc/fstab
```

Este archivo menciona varias particiones de nuestro disco rígido ¿Cómo podemos verlas? Coloque:

```
# fdisk -l /dev/hda
```

Y obtendrá un listado de cada partición, si está marcada como «activa» para que pueda arrancar desde ahí (boot), los cilindros que abarca (desde/hasta), y el tipo de partición de la que se trata. De acuerdo con el tipo de partición será el tipo de sistema de archivos que podrá tener (Cuidado: no confundir tipo de partición con tipo de sistema de archivos).

¿Cómo se crea un sistema de archivos? Lo que en la jerga del DOS se llama «formatear» en directa alusión al programa FORMAT que realiza esa operación, puede hacerse en Linux con el comando genérico `mkfs` (es decir, justamente, *make filesystem*) seguido de un punto y el sistema de archivos específico, por ejemplo `mkfs.vfat` ó `mkfs.ext2`.

```
# mkfs.vfat /dev/fd0
```

También podemos usar el comando específico, por ejemplo si queremos crear un sistema de archivos *extended second* (es decir, para Linux) en un diskette, deberíamos colocar:

```
# mke2fs /dev/fd0
```

En lugar de:

```
# mkfs.ext2 /dev/fd0
```

Que también es válido. Si quisiéramos comprobar el sistema de archivo, deberíamos colocar, análogamente:

```
# fsck.ext2 /dev/fd0 (o su equivalente)
# e2fsck /dev/fd0
```

Si el sistema está marcado como «limpio», será necesario «forzar» la comprobación con la opción «-f».

```
# e2fsck -f /dev/fd0
```

Para ver el contenido, es decir, para poder acceder al sistema de archivos que hay en él, es necesario montarlo, como dijimos anteriormente. Por ejemplo:

```
# mount -t ext2 /dev/fd0 /mnt/floppy
```

Con todas sus opciones, estamos diciendo que el tipo de sistema de archivos es `ext2`, que el dispositivo es `/dev/fd0` y que debe ser montado en `/mnt/floppy`. Ahora podemos listar el contenido del sistema de archivos del diskette con:

```
$ ls /mnt/floppy
```

Aunque está vacío, porque acabamos de crearle el sistema de archivos, hay un directorio llamado `lost+found`, es decir «perdidos+encontrados», en ese directorio, típico de este sistema de archivos, se guardan archivos (o porciones de archivos) que se han recuperado luego de una falla (`e2fsck` los coloca ahí).

Podemos realizar varias operaciones con archivos y directorios, primeramente nos cambiaremos al directorio en el que montamos el diskette, para trabajar sobre él.

```
# cd /mnt/floppy
```

6.3.2 Directorios, archivos y enlaces

Acá van una pequeña lista no exhaustiva, por ejemplo, para crear un directorio llamado «nuevo»: (aunque no es necesario en muchos casos, seguiremos usando la cuenta del superusuario `root`)

```
# mkdir nuevo
```

Para cambiarnos dentro de ese directorio:

```
# cd nuevo
```

Podemos con el comando `touch` crear un archivo si es que éste no existe, o actualizarle la fecha de acceso en caso de que el archivo exista, por ejemplo:

```
# touch miarchivo
```

Podemos crear un enlace simbólico con:

```
# ln -s miarchivo mienlace
```

Veámoslos en formato largo, con color:

```
# ls -l --color
```

Si intentamos editar el enlace con:

```
# vi mienlace
```

Podemos agregar el siguiente texto si entramos en modo inserción:
Es este `miarchivo` o `mienlace`?
Ahora veamos el contenido de ambos:

```
# cat mienlace
# cat miarchivo
# ls -l
```

Y borrarlo con:

```
# rm miarchivo
```

Piense: ¿A dónde apunta el enlace ahora?

```
# ls -l
# cat mienlace
```

¿Qué conclusiones obtiene?
Piense: ¿Qué ocurrirá con un enlace no simbólico (duro)?

```
# rm mienlace
# touch miarchivo
# ln miarchivo mienlace
# ls -l --color
# cat >miarchivo
es este miarchivo o mienlace?
^D
#
```

¿Qué conclusiones obtiene?
Como la estructura de archivos/directorios es la de un «árbol», podemos cambiarnos al nivel precedente que se denomina «directorio padre» y se simboliza con dos puntos seguidos «..»[14], entonces con:

```
# cd ..
```

Nos cambiamos al directorio padre, luego con:

```
# rmdir nuevo
```

Borraríamos el directorio que habíamos creado (¿Qué ocurre? ¿Cómo lo soluciona?).
¿Y si intentáramos desmontar el diskette mientras estamos cambiados sobre su punto de montaje?

```
# umount /mnt/floppy
```

Cámbiese a su «casa» e inténtelo nuevamente:

```
# cd
# umount /mnt/floppy
```

¿Qué conclusiones saca?
Puede saber cuál es el uso del espacio en las distintas particiones/volúmenes con el comando *disk free*:

```
# df -h
```

La opción «-h» muestra los valores en unidades significativas para los «humanos», es decir, en kilobytes, megabytes, etc. Observe cuidadosamente cuánto espacio se ocupa en los distintos tipos de sistema de archivos, por ejemplo puede crear en el diskette distintos «formatos», montarlo, observar el espacio, desmontarlo, crear otro tipo de «formato», montarlo, comparar, etc.

14. De manera análoga el directorio actual se simboliza con un punto «.»

Observe que en el CD ubuntu vienen varios tipos de sistema de archivos. Coloque el comando `mkfs` y apriete el tabulador dos veces para que el intérprete `bash` le muestre cuáles son los comandos que empiezan de esa manera.

Experimente creando un sistema de archivos tipo `ext2` en el diskette y en lugar de montarlo, coloque este comando:

```
# dumpe2fs /dev/fd0
```

Que hace un vuelco de la información del sistema de archivos.

6.3.3 Permisos

Como se trata de un sistema multiusuario, los archivos y directorios tienen modos o permisos de acceso, para el propietario, para el grupo al que pertenece el archivo o directorio, y para los restantes grupos (el «mundo»). Estos permisos básicamente son de lectura (r: *read*), escritura (w: *write*) y ejecución (x: *execute*). Como el permiso de ejecución carece de sentido en un directorio, la interpretación del mismo es «cambiarse a».

Es decir, si un directorio no tiene el permiso «x» no podemos colocar el comando «`cd`» para cambiarnos a ese directorio.

Para cambiar el modo de acceso utilizamos el comando `chmod` (change mode), cuyo forma genérica sería:

```
chmod {u, g, o, a}{+, -, =}{r, w, x} archivo o directorio
```

Por ejemplo, si queremos habilitar el permiso de lectura al archivo «`miarchivo`» por parte del grupo, colocaríamos el comando:

```
$ chmod g+r miarchivo
```

Note que al cambiar los permisos, éstos tienen una posición fija, que está habilitado o inhabilitado. ¿Qué le recuerda esto? ¡Acertó! Son bits. Por lo tanto, el patrón de bits «`rwxrw-r-`» equivale a «765» (rwx=7, rw-=6, r–=5). Pruebe lo siguiente:

```
$ chmod 777 miarchivo
$ chmod 000 miarchivo
$ chmod 111 miarchivo
$ chmod 222 miarchivo
$ chmod 444 miarchivo
```

Sistemas operativos

MARTÍN SILVA



Alfaomega

Sistemas de entrada y salida

7

7.1 Dispositivos de entrada y salida

Con respecto a la arquitectura de una computadora, hasta el momento nos hemos referido a sólo dos de sus componentes principales; es decir, al procesador y a la memoria. Los dispositivos de entrada y salida constituyen el tercer componente principal. Son los responsables de la comunicación con el mundo exterior y del almacenamiento de datos para su posterior uso. También son el cuello de botella de todo el sistema de componentes porque son mucho más lentos que los otros dos; el sistema operativo debe lidiar con esta situación ya que el procesador no se detiene, entonces aquel debe buscar formas para compensar esta dificultad y elaborar estrategias que mejoren el rendimiento total del sistema. Es función del sistema operativo el control de todos los dispositivos de entrada y salida, función que lleva a cabo enviando comandos a los dispositivos, detectando las interrupciones y controlando los errores.

Además, los usuarios cada vez quieren conectar (y desconectar) a la computadora una mayor cantidad y diversidad de dispositivos. El sistema operativo debe ser capaz de detectar el dispositivo nuevo y proporcionar el manejador adecuado, para luego revertir todo el procedimiento realizado cuando al usuario le plazca desconectarlo. Esto es notoriamente claro desde la aparición del USB (*Universal Serial Bus*). La administración de los dispositivos de entrada y salida debe ser, tal vez, el mayor desafío que enfrentan los sistemas operativos porque existe una gran diversidad de dispositivos con grandes diferencias de propósito, velocidad, volumen de información procesada, dirección o flujo de la información, protocolo de comunicación y disponibilidad, ya que no todos los dispositivos están conectados todo el tiempo y cada vez son más comunes los dispositivos removibles. El problema de la detección y carga de los manejadores de dispositivos puede entenderse mejor si pensamos al software del sistema operativo durante el proceso de arranque, que parte de no tener un conocimien-

to previo de la mayoría de los dispositivos instalados en la computadora y secuencialmente se va enterando de la presencia de cada uno de ellos, hasta finalizar el proceso de arranque. Años atrás, acá habría finalizado el proceso de carga de manejadores de dispositivos pero, en la actualidad, los sistemas operativos modernos son capaces de detectar la conexión y desconexión de distintos tipos de dispositivos y, como consecuencia, de cargar y descargar los manejadores apropiados.

Tampoco es menos importante el nombre por el cual, tanto el usuario como las aplicaciones y el sistema operativo, vayan a referirse al dispositivo; es decir, los sistemas operativos tienden a tratar a los dispositivos tales como los *pen-drive*, las cámaras fotográficas digitales o los teléfonos celulares como si se tratasen de medios de almacenamiento masivo, como un disco rígido removible, pero son dispositivos de distinto tipo. Esto es así porque el sistema operativo provee una abstracción de los dispositivos. Es decir que es función del sistema operativo proporcionar una interfaz sencilla y fácil de usar entre los dispositivos y el resto del sistema y que, preferentemente, debe ser la misma para todos los dispositivos. Otro ejemplo de resolución conflictiva de nombres podemos ofrecer si tenemos dos impresoras distintas conectadas a los puertos USB; si las desconectáramos y reconectáramos en otro orden luego de un reinicio del sistema, el sistema operativo debería ser capaz de reconocerlas unívocamente aun estando conectadas a puertos distintos, y referirse a ellas por el mismo nombre que les habíamos dado la vez anterior.

Otro aspecto a tener en cuenta es que ciertos tipos de dispositivos tienen su propio procesador y memoria locales; es decir que, como si todo este escenario no fuera ya suficientemente complejo, ahora debemos sumarle el paralelismo con el cual deben operar.

Sólo haremos una breve revisión de la tecnología utilizada en los dispositivos de entrada y salida como para dejar en claro algunos términos que necesitaremos para referirnos al software del sistema operativo que los administra. Pero como para hacer una lista no exhaustiva de los distintos dispositivos que podemos conectar en estos días a una computadora, podemos mencionar a las terminales, teclado, *mouse*, impresora, *scanner*, discos rígidos, discos ópticos, memorias *flash* (*pen-drive*), adaptadores de red, módems telefónicos, placas de audio, micrófonos, controles remotos infrarojos, relojes y temporizadores internos, la propia energía eléctrica y todo lo que podamos conectar a colectores (*buses*) USB, IEEE 1394 (conocidos comercialmente como Firewire, i.LINK o Lynx), PCI (*Peripheral Component Interconnect*), PCMCIA (*Personal Computer Memory Card International Association*), SCSI (*Small Computer System Interface*) o IDE (*Integrated Drive Electronics*).

7.2 Organización de las funciones de entrada y salida

Los dispositivos actuales de entrada y salida no están conectados al procesador ni a la memoria de manera directa sino a través de un colector de entrada y salida (*I/O Bus*), como puede apreciarse en el diagrama de bloques de la figura 7.1 de un sistema típico de computación. Estos colectores se comunican con el procesador a través de un módulo conmutador. Permiten la conexión de un rango amplio de dispositivos de entrada y salida, y proveen la especificación acerca de cómo deberían comunicarse con el procesador transfiriendo datos y comandos y cómo deberían competir los distintos dispositivos entre sí por el uso del colector. Tal vez ésta sea la principal desventaja de los colectores de entrada y salida: deben ser

compartidos por todos los dispositivos conectados a él y poseen un ancho de banda limitado. Cada tipo de colector, tal como el PCI o el SCSI, define un protocolo para accederlo.

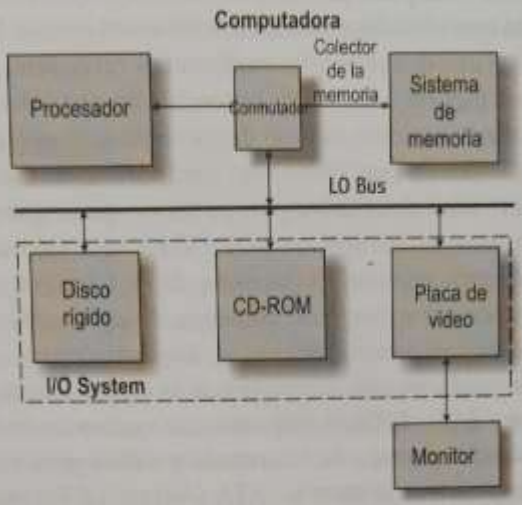


Fig. 7.1 – Colectores de entrada y salida.

Uno de los términos clave es la **política de arbitraje** (*arbitration policy*) usada para decidir qué dispositivo puede acceder al colector en un momento dado. Muchos colectores permiten que cualquier dispositivo solicite el uso del mismo y tienen una política para decidir cuál dispositivo lo utilizará. Por ejemplo, en SCSI, cada dispositivo tiene un identificador único (el SCSI ID) y cuando varios dispositivos solicitan a la vez el colector, la política de arbitraje indica asignarle el colector al dispositivo con el número de identificador más alto. Es fácil decidirlo pero tiene la desventaja de que puede provocar inanición (*starvation*) de los dispositivos con número de identificador más bajo.

Los dispositivos suelen comunicarse con la máquina a través de un punto de conexión denominado **puerto** (*port*). Pueden estar compuestos por un componente mecánico y un componente electrónico o **controlador** (*controller*), que es una colección de circuitos electrónicos que pueden operar un puerto, un colector o un dispositivo. Un controlador de puerto serial es un ejemplo de controlador de dispositivo sencillo: es un solo *chip* del computador que controla las señales en los alambres de un puerto serial (por ejemplo, el UART 16550). En contraste, un controlador de canal SCSI a menudo se implementa como una tarjeta de circuitos aparte que se enchufa en un zócalo de la placa madre de la computadora y que por lo regular contiene un procesador, microcódigo y un poco de memoria privada para poder procesar los mensajes del protocolo SCSI. Estos procesadores de entrada y salida difieren de la CPU porque utilizan instrucciones distintas tanto en formato como en significado.

Un puerto de entrada y salida suele tener cuatro registros, llamados de **estado**, de **control**, de **entrada de datos** y de **salida de datos**. El registro de estado contiene bits que el adaptador puede leer. Estos bits indican, por ejemplo, si la orden actual ya se ejecutó, si hay un byte disponible para leerse en el registro de entrada de datos y si ha habido un error de dispositivo. El adaptador puede escribir en el registro de control para iniciar una orden o cambiar el modo

de un dispositivo. El adaptador lee el registro de entrada de datos para obtener entradas y escribe en el registro de salida de datos para enviar salidas.

El adaptador puede estar en **espera activa** (*busy waiting*) o **escrutinio** (*polling*). La espera activa es una técnica en la que el código revisa repetidamente en un lazo (por eso también se le suele llamar *busy loop*) si una determinada condición es verdadera, por ejemplo esperar una entrada desde un teclado o esperar a que esté disponible un candado (*lock*). Si bien sabemos que la espera activa es inherentemente tiempo desperdiciado y, por lo tanto, desaconsejable como técnica de programación, en este ámbito del hardware suele dar mejores resultados que sus alternativas. Es una actividad **sincrónica** porque debe esperar que termine la operación de entrada y salida para continuar (*synchronous I/O*) y si hubiese sido iniciada por un proceso, éste habría quedado bloqueado esperando (*blocking I/O*). La técnica del escrutinio (*polled I/O*) consiste en que el procesador, luego de comenzar la operación de entrada y salida, retorne a la ejecución, pero revise periódicamente al dispositivo para saber si ya se completó la operación de entrada o salida; es más eficiente que la espera activa. En la entrada y salida **programada** (*programmed I/O* o PIO) el software que ejecuta en la CPU usa instrucciones que acceden al espacio de direcciones de la entrada y salida para realizar transferencia de datos desde o hacia el dispositivo; la interfaz ATA (*Advanced Technology Attachment*) utilizaba PIO. La entrada y salida manejada por **interrupciones** (*interrupt driven I/O*) es una actividad **asíncrona** porque los módulos de entrada y salida al finalizar su tarea envían una señal que interrumpe a la CPU (*asynchronous I/O*) y si hubiese sido iniciada por un proceso, éste no se habría quedado bloqueado esperando (*non-blocking I/O*). Las interrupciones normalmente están implementadas en un circuito controlador de interrupciones tal como el PIC (*Programmable Interrupt Controller*) de la PC de IBM. La interfaz IEEE 1394 es un estándar de colector serial para comunicaciones de alta velocidad y transferencia de datos **isócronas** de tiempo real (cuya especificación está publicada por la IEEE Computer Society (2008)). En las transmisiones isócronas (de igual tiempo) los datos deben ser transmitidos dentro de ciertas limitaciones de tiempo. En la transmisión serial isócrona cada carácter tiene un bit de arranque y uno de detención, pero el equipo de emisión y de recepción se sincronizan en cuanto a la longitud de cada tiempo; es decir, cada byte está sincronizado o enmarcado entre los bits de arranque y detención.

Para usar el sistema de entrada y salida, el procesador debe ser capaz de enviar comandos a los dispositivos de entrada y salida, y leer los datos que provengan de ellos. Para ello, la mayoría de los sistemas utilizan o bien un espacio de direcciones especial para la entrada y salida, asignando a cada controlador una parte de él, o bien un mecanismo llamado entrada y salida **mapeada en memoria** (*memory-mapped I/O*) en la que los registros de control para cada dispositivo de entrada y salida aparecen al programador como ubicaciones de memoria convencional. Cuando el sistema operativo lee o escribe estas ubicaciones de memoria, el hardware transforma las operaciones en una transacción sobre el colector o *bus* que lee o escribe los registros apropiados en el dispositivo. En el caso de una lectura, el resultado de la operación es transferido al procesador sobre el colector de entrada y salida y escrito sobre los registros de destino.

De esta manera, la CPU tiene que copiar cada trozo de datos desde el origen al destino, no quedando disponible para otras tareas. Esta situación se agrava si consideramos que el acceso a los dispositivos de entrada y salida conectados a un colector de periféricos es mucho más lento que a la memoria RAM. En cambio, el **acceso directo a memoria** (*Direct Memory*

Access o DMA) es el mecanismo hardware que le permite a los componentes periféricos transferir sus datos de entrada y salida directamente desde y hacia la memoria principal sin necesidad de involucrar al procesador del sistema. El uso de este mecanismo puede incrementar en gran medida el rendimiento del dispositivo porque se elimina una gran sobrecarga computacional.

7.3 Aspectos del diseño

7.3.1 Objetivos

Como decíamos al comienzo del capítulo, el acceso a los dispositivos de entrada y salida es el cuello de botella de todo el sistema de componentes, tal como podemos apreciar en la tabla 3.2. Por lo tanto, un objetivo que debería alcanzar el software que los gestione es la **eficiencia**. Y dada la gran variedad de dispositivos que debe gestionar el sistema operativo, sería deseable que lograra alcanzar una cierta **generalidad** en el tratamiento de los mismos, ocultando sus particularidades, sin embargo debido a la diversidad de características de los dispositivos, en la práctica es difícil conseguir una auténtica generalidad. Lo que puede hacerse es emplear un enfoque jerárquico y modular para el diseño de las funciones de entrada y salida.

7.3.2 Mecanismo y política

A los manejadores de dispositivos a veces los escriben los programadores de aplicaciones para hacer funcionar su propia aplicación. Por ejemplo, lo escriben para el puerto paralelo porque quieren que genere pulsos que manejen a un motor de pasos (*stepper motor*). Este enfoque conduce a desarrollar manejadores de dispositivos que dependen mucho de la aplicación en particular y no son suficientemente genéricos como para que los pueda utilizar otra aplicación. Lo que se debe tener en cuenta al desarrollar un manejador de dispositivos es separar **mecanismo** y **política**. Con mecanismo nos referimos a la representación fidedigna de las funcionalidades puras del dispositivo, independientemente de qué partes de la funcionalidad usará una aplicación. Una vez que el manejador de dispositivo ofrece una interfaz de software al mecanismo del dispositivo, un programador de aplicaciones puede usar esta interfaz para usar el dispositivo de una manera en particular.

Si el mecanismo se refiere a qué capacidades deben proporcionarse, la política (*policy*) hace referencia, en cambio, a cómo se pueden usar esas mismas capacidades: se refiere a la forma en la que un programador puede usar esta interfaz del mecanismo para usar el dispositivo de una manera en particular. El rol del manejador de dispositivos, según Rubini y Corbet (2001), es el de proporcionar mecanismos, no se deberían proporcionar políticas.

7.3.3 Estructura lógica de las funciones

Los sistemas operativos intentan ofrecer al resto del sistema una interfaz estándar, simple y uniforme para el uso de un dispositivo. Para ello, el software encargado de administrar la entrada y salida se organiza en forma de **capas**. Las capas más cercanas al hardware se encargan de ocultar las particularidades del mismo a las capas superiores y éstas deben presentar

una interfaz agradable, limpia y regular a los usuarios, como se ilustra en la figura 7.2, en la que observamos cómo se organizan los componentes del sistema de entrada y salida en Windows.

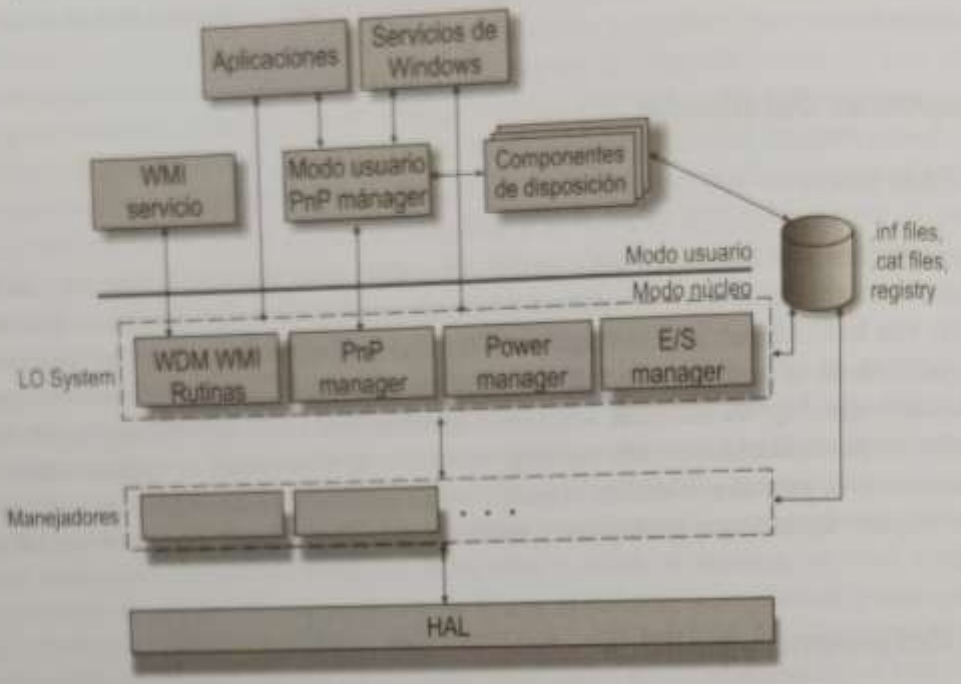


Fig. 7.2 – Componentes del sistema de entrada y salida en Windows.

Al existir una variedad tan amplia de dispositivos –y si no existiera el sistema operativo–, los programas deberían ser capaces de conocer en alto grado de detalle las particularidades de cada uno de ellos para poder operarlos, lo cual agregaría una gran complejidad a los programas, aumentaría mucho su tamaño y sería necesario modificarlos para cada dispositivo nuevo que apareciera en el mercado. El concepto clave aquí es la **independencia** del dispositivo, es decir, debería ser posible escribir programas que se pudieran utilizar con archivos en distintos dispositivos, sin tener que modificar los programas para cada tipo de dispositivo; este problema debe ser resuelto por el sistema operativo. Entonces, una interfaz define un conjunto de funciones estandarizadas que permiten la abstracción, el encapsulamiento y la división del software en capas.

7.4 Subsistema de entrada y salida del núcleo

Según Silberschatz y Galvin (1997) el núcleo del sistema operativo debería ofrecer servicios relacionados con la entrada y salida, apoyándose en el hardware y en la infraestructura de manejadores de dispositivos. Estos servicios son la planificación de la entrada y salida, el uso de memorias de almacenamiento intermedio (*buffers*), cachés y *spool*, la reservación de dispositivos y el manejo de errores; servicios que trataremos a continuación.

7.4.1 Planificación de la entrada y salida

Siendo la administración de los recursos uno de los objetivos del sistema operativo, se requiere una cuidadosa planificación para el acceso controlado a los mismos. Así como un planificador de procesos virtualiza el recurso de tiempo de procesador entre distintos procesos en el sistema, un planificador de entrada y salida (*I/O scheduler*), según Love (2007), virtualiza los recursos de entrada y salida al disco, entre distintas solicitudes de acceso, con el fin de maximizar el rendimiento global. A diferencia de un planificador de procesos, el de entrada y salida no es un componente obligatorio del sistema operativo pero sí lo es el rendimiento, que es la única razón de ser del planificador de la entrada y salida. La planificación, según Iyer (2001), provee la infraestructura que recibe solicitudes desde las capas superiores del núcleo, despacha solicitudes al manejador del disco y gestiona la finalización de estas solicitudes.



Fig. 7.3 – Diagrama de transición de estados de un planificador.

Lo podemos modelar mediante un diagrama de transición de estados, como vemos en la figura 7.3. Básicamente, un planificador opera entre dos estados: «ocioso» y «ocupado», reflejando así el estado de ejecución del disco.

7.4.2 Almacenamiento intermedio

Cuando dos procesos, locales o remotos, se comunican intercambiando mensajes, suele ocurrir que uno de ellos sea más lento que el otro, ya sea en la transmisión en la recepción. Este es un problema clásico del modelo de productor y consumidor, que se presenta frecuentemente en Ciencias de la Computación y que resolvemos mediante la utilización de una **memoria de almacenamiento intermedio** o *buffer*. Sinha (1996) hace una exposición muy completa de este tema desde la perspectiva de los sistemas operativos distribuidos en los que este problema se presenta acrecentado.

Los mensajes pueden transmitirse desde un proceso a otro copiando el cuerpo del mensaje desde el espacio de direcciones del proceso transmisor al espacio de direcciones del proceso receptor (posiblemente a través del espacio de direcciones del núcleo).

En la comunicación entre procesos, la estrategia de almacenar mensajes está fuertemente relacionada con la estrategia de sincronización. Los nodos sincrónicos y asincrónicos de comunicación corresponden respectivamente a los dos extremos del *buffering*: un *buffer* nulo, o sin *buffer* y un *buffer* con capacidad ilimitada. Otras dos estrategias comúnmente usadas son las del *buffer* de mensaje simple y límite finito o mensaje múltiple.

7.4.2.1 Memoria intermedia nula

En este caso no hay lugar para almacenar temporalmente el mensaje. Entonces puede usarse una de las siguientes estrategias de implementación:

1. Cita o *rendez-vous*. El mensaje permanece en el espacio de direcciones del proceso transmisor y la ejecución del `send()` se demora hasta que el receptor ejecuta el correspondiente `receive()`. Para hacer esto, el proceso transmisor se respalda y suspende de manera tal que cuando se desbloquea, comienza por reejecutar la sentencia `send()`. Cuando el receptor ejecuta `receive()`, se envía un acuse de recibo al núcleo del transmisor diciendo que éste puede ahora enviar el mensaje. Al recibir el mensaje de acuse de recibo, el remitente se desbloquea, provocando que se ejecute el `send()` otra vez. Esta vez el mensaje se transfiere satisfactoriamente desde el espacio de direcciones del transmisor al espacio de direcciones del receptor porque el receptor está esperando recibir el mensaje.
2. Descarte. El mensaje simplemente se descarta y el mecanismo de tiempo límite (*timeout*) se usa para reenviar el mensaje después de un período de tiempo límite. Esto es, después de ejecutar el `send()`, el proceso transmisor espera un acuse de recibo del proceso receptor. Si no se recibe ningún acuse de recibo dentro de un período de tiempo límite, asume que el mensaje ha sido descartado e intenta otra vez esperando que esta vez el receptor haya ejecutado el `receive()`. El transmisor puede tener que intentar varias veces antes de tener éxito. El transmisor se da por vencido después de reintentar un número predeterminado de veces.

7.4.2.2 Memoria intermedia sencilla

Generalmente, la estrategia del *buffer* nulo no es apta para comunicaciones sincrónicas entre dos procesos en un sistema distribuido porque si el receptor no está listo, el mensaje debe transferirse dos o más veces, y el receptor del mensaje debe esperar todo el tiempo que toma transferir el mensaje a través de la red.

En un sistema distribuido, la transferencia de un mensaje a través de la red puede requerir un tiempo significativo en algunos casos. Entonces, en lugar de usar una estrategia de *buffer* nulo, los mecanismos de comunicación en sistemas de red/distribuidos usan una estrategia de *buffer* de mensaje único. En ésta, un *buffer* que tiene la capacidad de almacenar un único mensaje se usa en el nodo del receptor. Esto es porque en sistemas en comunicación sincrónica, un módulo de aplicación puede tener como máximo un mensaje pendiente por vez. La idea de la estrategia del *buffer* de mensaje simple es mantener el mensaje listo para su uso en el lugar del receptor. Por lo tanto, en este método, el mensaje de pedido se almacena en el nodo del receptor si el receptor no está listo para recibir el mensaje. El *buffer* del mensaje puede estar en el espacio de direcciones del núcleo o en el espacio de direcciones del proceso receptor.

La clase de apoyo más simple que el sistema operativo puede ofrecer es la memoria intermedia sencilla. Por ejemplo, podría ofrecer un *buffer* entre el puerto serial (UART) y el módem, ya que la velocidad del primero puede alcanzar 38400 bps mientras que la del último

puede tener una portadora de 14400 bps. Otro ejemplo similar lo tenemos con las tarjetas de red (100 Mbps) conectadas a dispositivos ADSL (2 Mbps).

En el caso de una lectura del disco es esperable, después de haber completado la transferencia se soliciten datos presentes en el bloque (o sector) siguiente, por lo que generalmente el sistema operativo reservaba un *buffer* en el que almacenaba el bloque siguiente. A esta técnica se la llama **lectura por adelantado** o **entrada anticipada** (*read-ahead*) y, dada su probada ventaja, actualmente se la implementa en el hardware de la controladora (*disk buffer*), por motivos de eficiencia.

7.4.2.3 Memoria intermedia doble

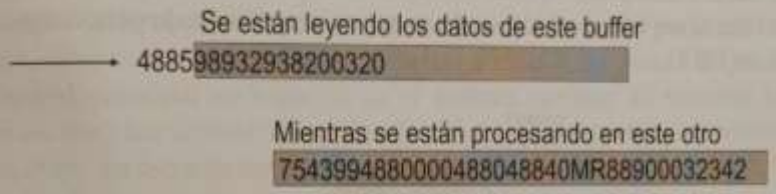


Fig. 7.4 – Memoria intermedia doble.

Se utilizan dos áreas de almacenamiento de manera que mientras una está siendo leída la otra está siendo escrita, como se muestra en la figura 7.4. Por ejemplo, al reproducir audio o video, mientras los datos en un *buffer* se envían a la tarjeta de sonido (o a la de video) el otro *buffer* se está llenando con más datos desde su fuente, como el caso de un reproductor de CD o DVD.

7.4.2.4 Memoria intermedia circular

Un *buffer* circular es una estructura de datos que utiliza varios almacenamientos intermedios simples, de tamaño fijo, lógicamente contiguos y en la que al último elemento le sigue el primero generando, conceptualmente, un anillo, tal como se ve en la figura 7.5.

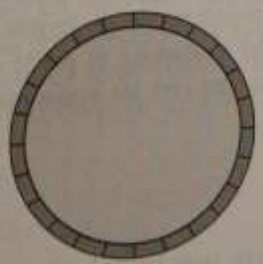


Fig. 7.5 – Memoria intermedia circular.

De manera que el *buffer* parece no tener fin, si bien tiene una longitud predeterminada. Pero en la gestión de memoria no hay manera de definir un anillo (físicamente hablando) y por eso se usa una representación lineal del mismo.

Comienza vacío y se lo va llenando a partir de cualquiera de sus elementos (da lo mismo porque conceptualmente no tiene inicio ni fin) y de ahí en más de forma consecutiva. El consumidor va extrayendo datos del anillo, comenzando por el primero ingresado a la estructura (FIFO), mientras que el productor la sigue llenando, hasta que el *buffer* esté completamente lleno, como se muestra en la figura 7.6.

Si esto ocurre y el productor sigue ingresando datos a la estructura, se producirá una sobrescritura de los elementos más «viejos» y que no pudieron ser consumidos a tiempo. Hay varias alternativas para lidiar con esta situación: por ejemplo, utilizar un *buffer* suficientemente grande como para no llenarlo rápidamente, o hacer que el productor maneje un tamaño menor de elementos, o aceptar resignadamente la pérdida de algunos datos en aras de la funcionalidad.

La ventaja del almacenamiento intermedio circular es que los elementos no deben ser desplazados en la lista al ser consumidos, es decir que es muy adecuado para consumir elementos de una estructura FIFO.

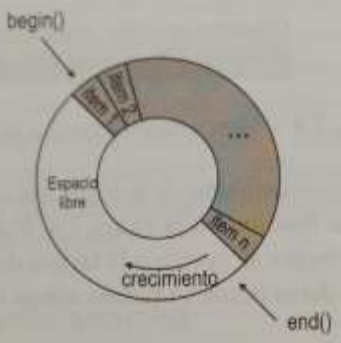


Fig. 7.6 – Uso de la memoria intermedia circular.

Siguiendo con el ejemplo de reproducción de audio y video, en ciertas ocasiones puede ser preferible la utilización de una memoria intermedia circular, por ejemplo si se trata de una reproducción típica de Internet, en forma de flujo continuo, sin interrupciones, a la que denominamos *streaming*, en lugar de descargar el archivo completo para reproducirlo con posterioridad. En este caso, es en cierto modo preferible y aceptable la pérdida de algunos datos porque lo que nos interesa del *streaming* es poder disfrutar de la reproducción a medida que se va generando, como ocurre con las emisoras de radio y TV que transmiten su programación por Internet.

7.4.3 Caché

La caché también es una memoria, pero más rápida (y cara) que la convencional en la que se mantiene una **copia** de los datos originales, de manera que es más eficiente el acceso a esta copia que al original. El objetivo es el de mejorar el desempeño mediante el almacenamiento transparente de datos. Normalmente, los datos que están almacenados en la caché pueden ser valores que han sido obtenidos recientemente mediante un cálculo previo o duplicados de originales almacenados en otra parte, de modo que al intentar volver a leerlos, si es exitoso,

tenemos un *cache hit* y sino un *cache miss*. La diferencia con un *buffer* es que una caché es una copia del dato original, en cambio el *buffer* puede ser la única versión del dato.

7.4.4 Spool

El término *spool* es un término que data desde hace mucho tiempo, acrónimo del inglés *Simultaneous Peripheral Operation On Line*, (operación periférica simultánea en línea). Se utilizó por primera vez en el sistema IBM 7070 y, de acuerdo con Bashe y col. (1981), para permitir el solapamiento entre el procesamiento de la CPU y múltiples operaciones de entrada y salida, el 7070 tenía una capacidad de interrupción llamada «procesamiento prioritario». Permitía que un programa principal fuera interrumpido cuando un dispositivo de entrada y salida requiriera servicios, el cual era provisto mediante la permutación a una «rutina prioritaria» que no admitía interrupción. El procesamiento prioritario se usó para realizar operaciones con periféricos, tales como conversiones de tarjeta a cinta y de cinta a impresora, todo esto en la computadora principal en lugar de en el sistema auxiliar. El término SPOOLing (se usaba en mayúsculas) fue acuñado para esta función, que representó una forma temprana de multiprogramación. Su uso está documentado en el manual IBM 7070 SPOOL System - form J28-6047-1, y su diseño está estudiado en Needham y Hartley (1969).

Actualmente, el término se refiere al proceso de colocar en un *buffer* salidas destinadas a un dispositivo, por ejemplo una impresora, que no puede aceptar corrientes de datos intercaladas; los datos (el archivo) son colocados en una fila o cola hasta que el dispositivo pueda procesarlos. De esta manera, se compensan las distintas velocidades a las que pueden operar el dispositivo y la CPU. Pero también se pueden crear archivos de *spool* para procesos relativamente lentos como, por ejemplo, los que utilizan los procesos servidores de correos electrónicos. El uso de *spool* es una forma muy ingeniosa en que los sistemas operativos pueden coordinar salidas concurrentes.

7.4.5 Manejo de errores

Según Silberschatz y Galvin (1997), un sistema operativo que emplea memoria protegida puede evitar muchos tipos de errores de hardware y de las aplicaciones, de modo que un problema mecánico menor no cause un fallo total del sistema. Tanto los dispositivos como las transferencias de entrada y salida pueden fallar y el sistema operativo puede compensar los fallos transitorios como, por ejemplo, el de un controlador de disco descompuesto. Algunos componentes de hardware pueden proporcionar información de error muy detallada, como el caso de los dispositivos SCSI.

7.4.6 Estructuras de datos

El núcleo necesita mantener información de estado acerca del uso de los componentes de entrada y salida; esto lo hace a través de diversas estructuras de datos internas, como la estructura de tabla de archivos abiertos. El núcleo de un sistema operativo emplea muchas estructuras similares para seguir la pista a las conexiones de red, comunicaciones con los dispositivos y otras actividades de entrada y salida.

7.5 Planificación del acceso a discos

7.5.1 Estructura de un disco

Para comprender el rol de un planificador de entrada y salida, repasemos algo de información básica y veamos cómo se comporta un sistema sin un planificador de entrada y salida. Los discos rígidos almacenan sus datos utilizando el familiar direccionamiento basado en la geometría de cilindros, cabezas y sectores (*cylinders, heads, sectors*). Una unidad de discos rígidos está compuesta por varios **platos** (*platters*), un eje y una cabeza lecto-grabadora. Cada plato está subdividido en regiones circulares concéntricas semejantes a anillos y llamadas **pistas** (*tracks*). Finalmente, cada pista está compuesta por un cierto número entero de sectores.

Para ubicar una unidad específica de datos en un disco rígido, la lógica del mecanismo requiere de tres piezas de información: el cilindro, la cabeza y el sector. El cilindro especifica la pista en la cual residen los datos. Si ponemos los platos uno encima del otro (que es como están en la unidad de disco rígido), una pista dada forma un cilindro a través de cada plato. Luego, la cabeza identifica la cabeza lecto grabadora exacta (y por lo tanto al plato exacto). La búsqueda ahora se ha restringido a una única pista en un único plato. Finalmente, el valor de sector denota al sector exacto sobre la pista, y así finaliza la búsqueda. La unidad de discos rígidos ahora sabe en qué sector, pista y plato reside el dato. Puede posicionar la cabeza lecto-grabadora del plato correcto sobre la pista correcta y leer el sector adecuado. Por suerte, los discos rígidos modernos no fuerzan a las computadoras a comunicarse en términos de cilindros, cabezas y sectores, sino que hacen corresponder un número único de bloque con cada terna cilindro/cabeza/sector (en inglés, C/H/S). El número único identifica a un valor específico de cilindro/cabeza/sector. Los sistemas operativos modernos entonces pueden ubicar datos utilizando este número de bloque, llamado direccionamiento de bloque lógico (*logical block addressing*) y la unidad de discos rígidos traduce el número de bloque al valor correcto de cilindro/cabeza/sector.

Algo para hacer notar respecto a este número de bloque, si bien nada lo garantiza, es que la correspondencia física tiende a ser secuencial. Es decir, el bloque lógico n tiende a estar físicamente adyacente al bloque lógico $n + 1$. Discutiremos más adelante por qué esto es importante.

Se le llama búsqueda (*seeking*) al movimiento de la cabeza del disco y es una de las operaciones más caras en una computadora. El tiempo de búsqueda (*seek time*) en las unidades modernas de discos se miden en décimas de milisegundos. Al tiempo total de acceso (*access time*) hay que sumarle, además del de búsqueda, el tiempo que tarda el sector deseado en llegar a la cabeza del brazo en la rotación; es llamado retardo o latencia de giro (*rotational latency*) y, con las velocidades de rotación de las unidades de discos rígidos actuales, es notablemente menor que el tiempo de búsqueda.

La tasa de transferencia de datos (*data transfer rate*) en las unidades de discos rígidos que giran a 7200 revoluciones por minuto (rpm) puede llegar a los 1030 Mbits/sec utilizando *buffers*.

La cabeza lecto-grabadora no tiene contacto con la superficie, flota sobre un colchón de aire del orden de los micrones. Cuando la cabeza toca la superficie, la daña y ocurre un «aterrizaje de cabezas» o *head crash*.

7.5.2 Planificación del acceso al disco

Pensemos por un momento en un típico sistema UNIX. Aplicaciones tan variadas como bases de datos, clientes de correo electrónico, servidores Web y editores de texto emiten solicitudes de lectura y escritura al disco, tales como «lea este bloque» y «escriba este otro». Los bloques tienden a estar ubicados físicamente sobre todo el disco. El *spool* del correo electrónico puede estar ubicado en una región completamnente diferente del disco de los datos HTML del servidor de Web o del archivo de configuración del editor de texto. En efecto, incluso un único archivo puede estar diseminado por todo el disco si está fragmentado, es decir, no dispuesto en bloques secuenciales.

Dado que los archivos están partidos en bloques individuales y las unidades de disco rígido se acceden por bloques y no mediante el concepto mucho más abstracto de archivo, la lectura o escritura de datos al archivo esta partida en un flujo de muchos pedidos individuales de entrada y salida, cada una a un bloque diferente. Con suerte, los bloques son secuenciales o al menos físicamente cercanos. Si los bloques no están cercanos unos a otros, la cabeza lecto-grabadora se debe mover a otra ubicación en el disco y esto, como ya dijimos, es una de las operaciones más caras en una computadora. Esta es la razón por la cual los archivos son convenientes.

Dicha ventaja es notoria en un sistema operativo monotarea y monousuario pero, desafortunadamente, no importa si los archivos están desfragmentados en un sistema multiusuario, porque el sistema está generando solicitudes de entrada y salida de distintas aplicaciones y usuarios por todo el disco, es decir, el cliente de correo electrónico quiere un poco de acá, y el servidor de Web quiere un poco de allá y el editor de texto quiere leer un archivo completo de más allá. El efecto neto es que la cabeza del disco se hace saltar por todo el disco. En el peor de los casos, con solicitudes intercaladas de entrada y salida a distintos archivos, la cabeza puede pasarse todo el tiempo saltando desde un lugar a otro, lo que no es bueno para el rendimiento global del sistema.

Aquí es donde hace su aparición el planificador de entrada y salida.

7.5.3 Distintos tipos de algoritmos de planificación

El planificador regula las solicitudes pendientes de entrada y salida para minimizar el tiempo dedicado a mover la cabeza del disco. Esto, a su vez, minimiza el tiempo de búsqueda y maximiza el rendimiento del disco rígido. Esta magia se logra mediante dos acciones principales: ordenar y mezclar (*sort and merge*).

Primero, el planificador de entrada y salida mantiene la lista de solicitudes pendientes de entrada y salida ordenadas por número de bloque. Cuando aparece una solicitud nueva se la inserta, en el sentido de los bloques (*block-wise*) en la lista de las solicitudes pendientes. Esto previene que la cabeza busque por todo el disco para atender las solicitudes de entrada y salida. En lugar de eso, al mantener la lista ordenada, la cabeza se mueve en una línea recta alrededor del disco (como si barriera desde el borde al centro y volviera). Si la unidad de discos rígidos está ocupada atendiendo una solicitud en una parte del disco y llega un pedido nuevo en la misma parte del disco, esa solicitud podrá ser atendida antes de pasar a otras partes del disco. La mezcla tiene lugar cuando se presenta una solicitud a una región

adyacente e idéntica del disco. En lugar de emitir una solicitud nueva, se junta con la solicitud adyacente o idéntica. Esto minimiza el número de solicitudes pendientes.

Veamos un ejemplo: Consideremos el caso en que dos aplicaciones solicitan (a través de llamadas a sistema, por supuesto) los siguientes números de bloques, de manera que llegan al núcleo en este orden: 10, 500, 12, 502, 14, 504 y 12.

7.5.3.1 FCFS

La forma más sencilla de planificación es, desde luego, el servicio por orden de llegada (*FCFS, first-come, first-served*). Casi podríamos decir que es un enfoque sin planificador, estos bloques se procesarán en ese orden. Es decir, siete búsquedas de una a otra parte del disco. Este algoritmo es intrínsecamente justo, pero generalmente no proporciona el servicio más rápido.

7.5.3.2 SSTF

Si el núcleo ordenara estas solicitudes, no obstante, y las procesara en otro orden los resultados serían muy diferentes, en nuestro ejemplo: 10, 12, 12, 14, 500, 502 y 504.

Parece razonable atender todas las solicitudes cercanas a la posición actual de la cabeza antes de moverla a una posición lejana para atender las más lejanas. Este supuesto es la base del algoritmo de tiempo de búsqueda más corto primero (*SSTF, shortest-seek-time-first*), que selecciona la solicitud que tiene el menor tiempo de búsqueda a partir de la posición actual de la cabeza.

El problema es que puede provocar inanición (*starvation*) pues un requerimiento a un bloque cercano a los extremos puede quedar siempre postergado por la llegada de requerimientos nuevos cercanos a la ubicación en ese momento de la cabeza (es probable que se den un conjunto de solicitudes de acceso a bloques cercanos entre sí). El SSTF, si bien es más injusto, introduce una mejora con respecto al FCFS, pero no es óptimo.

7.5.3.3 SCAN

En este algoritmo la cabeza se posiciona en un extremo del disco y va avanzando hacia el otro extremo, atendiendo los requerimientos de acceso a bloques que va encontrando por el camino.

También se le llama «el algoritmo del ascensor» pues su comportamiento es semejante al que realiza un ascensor automático: a medida que va subiendo o bajando va atendiendo las llamadas en orden.

Consideremos que, a medida que la cabeza va avanzado, van llegando requerimientos de acceso a bloques que ya pasaron y deben esperar que la cabeza vuelva luego de llegar al extremo hacia el que se dirige.

Cuando llega al extremo, al volver encuentra inicialmente pocos requerimientos (recién fue atendida esa zona). La mayor cantidad de requerimientos pendientes estará en el extremo opuesto, así que ¿por qué no ir ahí primero? El siguiente algoritmo se basa en esta idea.

7.5.3.4 C-SCAN

Para mantener uniforme el tiempo de espera de los requerimientos, se modifica el SCAN y se usa el C-SCAN (*Circular SCAN*).

En este algoritmo la cabeza, al llegar al extremo final se vuelve a posicionar en el inicio, para atender primero los requerimientos que, se supone por lógica, llevan más tiempo esperando. Este algoritmo trata a las solicitudes como una lista circular que continúa del último bloque al primero.

7.5.3.5 LOOK

De acuerdo con la descripción de los algoritmos vistos, se observa que el brazo va de un extremo al otro del disco, aún sin requerimientos; en la práctica no se implementa así sino que sólo llega hasta la última solicitud.

Por lo tanto podemos verlo como modificaciones al SCAN y C-SCAN, que, en este caso, pasan a llamarse LOOK y C-LOOK.

Los primeros antecedentes de un análisis comparativo de las distintas políticas de planificación se pueden encontrar en el trabajo de Teorey y Pinkerton (1971).

7.5.3.6 El ascensor Linus

El planificador de entrada y salida a la unidad de discos rígidos que encontramos en la versión 2.4 del núcleo Linux se llama «El Ascensor Linus» (*The Linus Elevator*). Fue desarrollado por Andrea Arcangeli y Jens Axboe, y funciona casi exactamente como el clásico planificador descripto anteriormente. Se utilizó con mucho éxito durante toda la versión 2.4 porque era simple y funcionaba bien. Desafortunadamente, en la búsqueda de un planificador para maximizar el rendimiento global se hizo una compensación para mejorar la latencia de solicitud.

Por ejemplo, consideremos el flujo de solicitudes a los bloques lógicos 20, 30, 700 y 25. El algoritmo de ordenamiento del planificador de la entrada y salida agregará a la cola y emitirá las solicitudes, tal como se espera, en el siguiente orden: 20, 25, 30 y 700. Y supongamos que a mitad de camino, mientras se está procesando el pedido al bloque 25, llega otra petición a la misma parte del disco, y luego otra y otra. Es factible que el pedido del bloque 700 no sea atendido por un buen tiempo. Peor aún ¿qué ocurriría si el pedido al bloque 700 fuera de lectura?

Los pedidos de lectura generalmente son sincrónicos. Cuando una aplicación hace un pedido de lectura de datos, a menudo se bloquea y espera hasta que el núcleo le retorne los datos. Las escrituras, por otra parte, a menudo no son sincrónicas sino asincrónicas. Cuando una aplicación escribe datos, el núcleo copia los datos y los metadatos dentro de su espacio de direcciones, prepara un *buffer* para contenerlos y retorna a la aplicación. A la aplicación realmente no le importa o ni siquiera sabe cuándo los datos llegan a quedar grabados efectivamente en el disco.

Sin embargo, se vuelve peor para el lejano pedido de lectura. Dado que las escrituras son asincrónicas, tienden a fluir, a volverse un *stream*. Es decir, es común que se produzca una reescritura de una gran cantidad de datos. Esto implica que muchos pedidos individuales de

escritura se envíen a un área cercana del disco. Por ejemplo, pensemos que cuando se guarda un archivo grande, la aplicación vuelca los pedidos de escritura sobre el sistema y la unidad de discos rígidos tan rápido como se pueda planificar.

Los pedidos de lectura, por el contrario, no tienden a fluir, sino que las aplicaciones emiten las solicitudes de lectura en pequeños pedazos uno a uno y cada trozo es dependiente del anterior (del último emitido). Piense en la lectura de todos los archivos en un directorio: La aplicación abre el primer archivo, emite un pedido de lectura de un trozo del mismo, espera a los datos, emite el pedido de lectura del siguiente trozo, espera y continúa y vuelve a esperar de la misma forma hasta que haya leído el archivo completo. Luego se cierra el archivo, se abre el próximo archivo y se repite el proceso. Todo pedido posterior tiene que esperar al previo, lo que significa demoras sustanciales en esta aplicación si los pedidos son a bloques lejanos.

Al fenómeno del «efecto flujo» de las solicitudes de escritura que hace demorar a las solicitudes de lectura se le llama lecturas hambreadas por escrituras (*writes-starving-reads*). Esta inanición en las solicitudes de lectura, aun por lapsos moderados de tiempo, dan como resultado altas latencias en la aplicación (para aquellas que realizan solicitudes de lectura durante otras actividades del disco). Esta alta latencia en las lecturas afecta negativamente al rendimiento del sistema y a la sensación que percibe el usuario.

7.5.3.7 El planificador con fecha límite

La prevención de la inanición de solicitudes en general y de solicitudes de lectura en particular fue un objetivo de los nuevos planificadores de entrada y salida propuestos para la versión 2.6 del núcleo Linux. El planificador con fecha límite (*deadline I/O scheduler*) fue propuesto por Jens Axboe para resolver el problema de la inanición que afectaba al planificador de entrada y salida de la versión 2.4 y de los algoritmos del ascensor en general. El Ascensor Linus fue reemplazado finalmente en la versión 2.5.39, según Axboe (2004).

Tal como dijimos, El Ascensor Linus mantiene la lista ordenada de las solicitudes de entrada y salida pendientes en una única cola. La solicitud de entrada y salida al comienzo de la cola es la próxima en ser atendida. El planificador con fecha límite continúa manteniendo esta cola, pero introduce dos colas adicionales: la cola FIFO de lectura y la cola FIFO de escritura, tal como podemos observar en la figura 7.7.

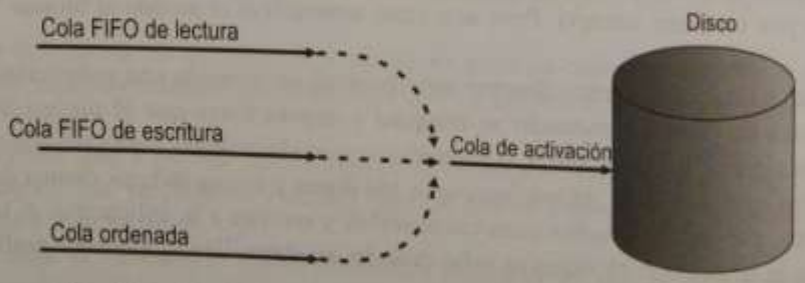


Fig. 7.7 – Planificador de entrada y salida con fecha límite.

El planificador con fecha límite mantiene a las solicitudes en estas colas ordenadas por orden de llegada (efectivamente, la primera en llegar será la primera en ser procesada). La cola FIFO de lectura, como su nombre indica, contiene sólo solicitudes de lectura y la cola FIFO de escritura contiene sólo solicitudes de escritura. A cada cola FIFO se le asigna una fecha de vencimiento, la cola FIFO de lectura tiene un tiempo de expiración de 500 milisegundos y la de escritura un tiempo de expiración de 5 segundos. Cuando se emite una solicitud de entrada o salida nueva, es ordenada e insertada en la cola estándar y ubicada al final de su respectiva cola FIFO (de lectura o escritura). Normalmente a la unidad de discos rígidos se le envían solicitudes de entrada y salida desde la cabecera de la cola estándar ordenada. Esto maximiza el rendimiento global al disminuir las búsquedas dado que la cola normal está ordenada por número de bloque, como en El Ascensor Linus.

Cuando la solicitud a la cabecera de una de las colas FIFO, no obstante, se vuelve más vieja que el valor de expiración asociado con su cola, el planificador de entrada y salida deja de despachar solicitudes desde la cola estándar y, en su lugar, atiende las solicitudes en la cabecera de la cola FIFO más un par extra. El planificador de entrada y salida necesita revisar y manejar sólo las solicitudes en la cabecera de las colas FIFO porque son las solicitudes más viejas en la cola.

Si traemos el ejemplo pasado a este algoritmo, notaremos que a pesar de la avalancha de pedidos de escritura a los bloques lejanos, luego de 500 milisegundos el planificador con fecha límite dejará de atender a aquellas solicitudes y atenderá la lectura al bloque 700. El disco buscará el bloque 700, atenderá el pedido de lectura y luego continuará atendiendo las otras solicitudes pendientes.

De esta manera, el planificador con fecha límite puede forzar un plazo sobre solicitudes de entrada y salida. A pesar de que no promete que una solicitud de entrada y salida sea atendida antes de la fecha límite, este planificador generalmente atiende solicitudes cerca de esa fecha. Así, continúa proveyendo un buen rendimiento global sin hambrear ni una solicitud por un tiempo inaceptablemente largo. Y dado que a las solicitudes de lectura se les dan tiempos cortos de expiración, el problema de las lecturas hambreadas por las escrituras está minimizado.

7.5.3.8 El planificador de entrada y salida anticipatorio

Consideremos qué pasa con nuestra solicitud ficticia al bloque 700, el cual presumiblemente es el primero de muchas lecturas dependientes a esa área del disco. Luego de atender la solicitud de lectura, el planificador de entrada y salida con fecha límite continúa atendiendo los pedidos de escritura de los bloques anteriores. Esto está bien, hasta que la aplicación que está leyendo emita su próxima solicitud de lectura (digamos, al bloque 710). En 500 milisegundos, esa solicitud expira y el disco busca (la cabeza se mueve) hasta el bloque 710, atiende la solicitud, vuelve al lugar donde estaba antes y continúa atendiendo al flujo de escrituras. Y luego llega otra lectura.

El problema surge de nuevo con las lecturas, debido a que éstas son solicitadas en trozos dependientes y la aplicación solicita la próxima sólo cuando ha recibido la previa. Pero para el momento en el que recibe los datos leídos, y se planifica para ejecutar y enviar a la próxima lectura, el planificador de entrada y salida ya ha seguido adelante y ha comenzado

a atender otros pedidos. Esto deja como resultado un par de búsquedas perdidas por cada lectura: búsqueda para la lectura, atención y búsqueda hacia atrás.

Se le llama «ociosidad engañosa» (*deceptive idleness*) a la situación en la que un proceso parece haber finalizado la lectura desde el disco cuando en realidad está preparándose para la próxima lectura. Iyer y Druschel (2001) tratan este comportamiento sincrónico en el que el planificador supone incorrectamente que el último proceso que solicitó la lectura ya no tiene más para efectuar y se ve forzado a conmutar a la solicitud de otro proceso.

Si sólo hubiera alguna manera de que el planificador de entrada y salida supiera –más bien se anticipara– que otra lectura pronto se efectuará a la misma parte del disco. En lugar de ir a buscar de un lado a otro, podría esperar anticipándose a la próxima lectura. Para ahorrarse esas terribles búsquedas sin duda valen la pena esos pocos milisegundos de espera.

Por supuesto, esto es lo que hace el planificador de entrada y salida anticipatorio (*Anticipatory I/O Scheduler*), cuyo autor principal es Nick Piggin, según Love (2004). Implementa la misma planificación basada en plazos que usa el planificador con fecha límite, pero fue dotado con el agregado de un mecanismo de anticipación. Cuando se solicita un pedido de lectura, el planificador de entrada y salida anticipatorio lo atiende dentro de sus plazos. No obstante, a diferencia del planificador con fecha límite, el anticipatorio espera hasta seis milisegundos. Es muy probable que la aplicación emita otra solicitud de lectura a la misma parte del sistema de archivos durante esos seis milisegundos. Si eso ocurre, esa solicitud será atendida inmediatamente y el planificador anticipatorio esperará seis más. Si pasaran los seis milisegundos sin una solicitud de lectura, significará que el planificador anticipatorio supuso incorrectamente y volverá a lo que estaba haciendo antes.

Si se pueden anticipar correctamente aunque sea un número moderado de solicitudes, se ahorra una gran cantidad de tiempo, como se ve en la tabla 7.1. Y como la mayoría de las lecturas son dependientes, la anticipación da resultado la mayoría de las veces. Para mejorar aún más las posibilidades de una anticipación correcta, el planificador anticipatorio utiliza una heurística para mejorar la suposición que se hace respecto a cuáles procesos esperar. Con este fin, el planificador mantiene estadísticas de entrada y salida de cada proceso para seguirle su comportamiento. Gracias a estas estadísticas y heurísticas inteligentes, el planificador anticipa correctamente las acciones de las aplicaciones en una cantidad de tiempo suficientemente grande que hace que valga la pena la sobrecarga.

Planificador	Prueba 1	Prueba 2
Ascensor Linus en 2.4	45 segundos	30 minutos, 28 segundos
con fecha límite en 2.6	40 segundos	3 minutos, 30 segundos
anticipatorio en 2.6	4.6 segundos	15 segundos

Tabla 7.1 – Planificadores en Linux.

Al minimizar las búsquedas innecesarias y atender más rápidamente las solicitudes de lectura, en muchas cargas de trabajo el planificador anticipatorio provee tanto una latencia

mejorada de solicitudes como un rendimiento global superior al del planificador con tiempos límite y el Ascensor Linus.

7.5.3.9 CFQ

Es otro planificador de la entrada y salida presente en el núcleo de Linux a partir de la versión 2.6.18, en inglés significa *Completely Fair Queuing* que podríamos traducir como «enfilación completamente justa», y también fue desarrollado por Axboe (2004). Trabaja colocando a las solicitudes sincrónicas en colas por proceso y luego asignando rodajas de tiempo a cada cola para acceder al disco. La longitud de la rodaja de tiempo y la cantidad de solicitudes que una cola puede emitir dependen de la **prioridad** en la entrada y salida que tenga el proceso. En cambio, a las solicitudes asincrónicas de todos los procesos se las agrupa en menos colas, una por prioridad. Si bien el algoritmo CFQ no hace una planificación anticipatoria explícita, logra el mismo efecto de tener un buen rendimiento para todo el sistema en su conjunto al permitir que la cola de un proceso quede ociosa al final de una operación de entrada y salida sincrónica, «anticipando» de esta manera mas cerca de la entrada y salida de ese proceso.

7.5.4 Manejadores de dispositivos

Todo el código que depende de los dispositivos aparece en los **manejadores de dispositivos** (*device drivers*), que son módulos del núcleo del sistema operativo, específicos del mismo y dependientes del hardware y que, si bien internamente diferencian entre los distintos tipos de dispositivos, ofrecen al resto del sistema y a las aplicaciones una interfaz estándar; dicho en palabras simples, es un programa que controla a un dispositivo. Pero los sistemas operativos modernos no sólo proveen una abstracción de los dispositivos físicos (hardware), sino también de los virtuales o lógicos (software).

Un manejador de dispositivos provee servicios de entrada y salida para un dispositivo. Por ejemplo, los siguientes son considerados manejadores de dispositivos para Windows: el manejador del canal IEEE 1394, un manejador de la clase video que administre datos de entrada en forma de flujo para una variedad de dispositivos de video, un manejador de la miniclase video que se comunique con el manejador de la clase para dar soporte a un dispositivo de video específico y a un manejador filtro que filtre los datos en forma de flujo. Algunos manejadores de dispositivo, particularmente los que son para dispositivos de audio, video e impresoras, ejecutan en modo usuario pero muchos ejecutan en modo núcleo.

Un manejador de sistema de archivos (*file system driver*) manipula la entrada y salida de manera independiente al dispositivo físico. Los manejadores de sistema de archivos incluyen a los manejadores para los sistemas de archivos suministrados por el sistema: NTFS y FAT. En los sistemas operativos basados en el núcleo NT, los manejadores del sistema de archivos operan en modo núcleo. Los manejadores filtro de los sistemas de archivos (*file system filter drivers*) proveen prestaciones adicionales a los manejadores de sistemas de archivos estándares, por ejemplo mediante la implementación de servicios de valor agregado tales como la detección de virus.

En los comienzos de los sistemas operativos –y durante varios años–, los dispositivos y sus manejadores eran bastante «estáticos», en el sentido de que no era muy habitual la co-

nexión de dispositivos nuevos. Con el tiempo, esta situación comenzó a cambiar y empezó a ser cada vez más necesaria la capacidad de detección automática de los dispositivos nuevos, es decir, el poder apagar el equipo, conectar el dispositivo nuevo, encenderlo nuevamente, que el sistema operativo detectara dicho dispositivo y lo dejara disponible inmediatamente para su uso. La tecnología que hizo esto una realidad se denominó en inglés *Plug and Play*, o **PnP** por sus siglas, y fue llevada adelante por un consorcio de fabricantes. Se incorporó en las últimas versiones del colector ISA (*Industry Standard Architecture*), pero no fue hasta la llegada del colector PCI cuando el estándar comenzó a funcionar correctamente. Para que un dispositivo sea considerado PnP debe estar libre de *jumpers* y de interruptores; debe ser capaz de «anunciarse» por sí mismo al sistema operativo, para que éste pueda cargar los manejadores adecuados, y debe ser configurable desde el mismo manejador, esto implica que debe ser capaz de asignarle sus propios espacios de direcciones de memoria y números de interrupción (IRQ), resolviendo el conflicto con los dispositivos previamente instalados. Muy relacionada con esta tecnología –al punto que a menudo se las confunde– está la capacidad de «conexión en caliente» o **hotplug**, como se la denominó originalmente en inglés, y que difiere de PnP en que no es necesario apagar el equipo, el usuario puede conectar y desconectar el dispositivo sin una interrupción significativa del sistema. También existe el término «reemplazo en caliente» o **hot swap**, sobre todo relacionado con los discos rígidos.

La labor de un manejador de dispositivos es la de aceptar las solicitudes abstractas que le hace el software independiente del dispositivo, emitir comandos al dispositivo y verificar la ejecución de dichas solicitudes; si al recibir una solicitud el manejador está ocupado con otra solicitud, agregará la nueva solicitud a una cola de solicitudes pendientes. Por ejemplo, un manejador de discos rígidos debería estimar el lugar donde se encuentra el bloque solicitado, verificar si el motor de la unidad funciona bien, enviar los comandos al controlador al escribir en los registros de dispositivo del mismo, tal vez bloquearse hasta que el controlador realice el trabajo hasta que una interrupción lo libere de ese estado, verificar si hubo errores y transferir los datos para que queden disponibles al software que los solicitó. El manejador puede efectuar estas operaciones con el procesador en modo privilegiado, lo que le otorga un cierto nivel de seguridad, porque así el sistema operativo se protege de las aplicaciones que ejecutan con el procesador en modo usuario; pero, por otra parte, pueden provocar un desastre si están mal codificadas.

Por ejemplo, M. E. Russinovich y D. A. Solomon (2004) afirman que Windows no provee protección alguna para la memoria del sistema de lectura o escritura privadas que está siendo usada por componentes que están ejecutando en modo núcleo. En otras palabras, una vez que se entró en modo núcleo el código del sistema operativo y los manejadores de dispositivo tienen acceso completo al espacio de memoria del sistema y pueden saltarse la seguridad de Windows para acceder a objetos; debido a que la mayor parte del código del sistema operativo Windows ejecuta en modo núcleo es vital que los componentes que ejecutan en dicho modo estén cuidadosamente diseñados y probados para asegurar que no violarán la seguridad del sistema. Esta falta de protección también aumenta la necesidad de tener cuidado cuando se carga un manejador de dispositivo de terceras partes porque una vez que se está en modo núcleo el software tiene acceso completo a todos los datos del sistema operativo. Esta vulnerabilidad fue una de las razones detrás del mecanismo de firmas de los manejadores introducido en Windows que le advierte al usuario si se hace un intento de agregar un manejador no autorizado (no firmado).

Ciertos manejadores pueden operar con el procesador en modo usuario también; es decir, podemos subdividir a los manejadores entre los que operan estrictamente sobre los dispositivos (o sus controladoras) físicos (*physical device driver* o PDD), y otros que operan en un nivel más lógico (*logical device driver* o LDD). los primeros los desarrollan los fabricantes del hardware porque tienen un conocimiento mucho más detallado, y los segundos los codifican los desarrolladores del sistema operativo. Por ejemplo, Microsoft ha creado un marco o *framework* para el desarrollo de manejadores llamado Windows Driver Foundation (WDD) que incluye al User-Mode Driver Framework (UMDF) frente al Kernel-Mode Driver Framework (KMDF). Como desventaja, generalmente las transiciones entre el espacio de usuario y el de núcleo imponen una sobrecarga, con lo cual no hacen adecuado a este modelo si tenemos requerimientos de baja latencia o alto rendimiento.

Cuando aparecieron las primeras computadoras, quienes las construían también desarrollaban el software, tanto las aplicaciones como el sistema operativo, con los manejadores para sus propios dispositivos. En la actualidad, hay muchos fabricantes y muchos sistemas operativos que intentan dar soporte a una gran diversidad de hardware. Son los fabricantes quienes tienen el conocimiento más detallado del hardware que venden y a veces no están dispuestos a divulgar estos detalles para proteger sus derechos de propiedad intelectual; entonces ellos mismos deben desarrollar el manejador para sus productos, si lo hacen para un sistema operativo de código cerrado, sus derechos están garantizados, pero si lo hacen para un sistema operativo de código abierto con licencia GPL tendrán un conflicto de intereses. Entonces se deben utilizar estrategias para que este sistema operativo pueda manejar el dispositivo sin conocer en detalle su arquitectura interna. Este es otro motivo para utilizar el modelo de desarrollo de manejadores físicos y lógicos. En Linux existe el **soporte dinámico de módulos del núcleo** (*Dynamic Kernel Module Support* o DKMS) cuya intensión es permitir que el código fuente del manejador de dispositivos resida en cualquier lugar del sistema de archivos y no sólo en el árbol de los programas fuente del núcleo.

Los manejadores de dispositivos virtuales también son un caso especial, se utilizan para imitar a dispositivos hardware en un ambiente de máquina virtual. Tal vez el caso más sencillo podría ser el de una aplicación para DOS que ejecuta en sistema operativo Windows XP y posteriores. El sistema operativo DOS brinda la ilusión de acceder al manejador de dispositivo virtual. Es fácil suponer que el sistema operativo DOS no tenía manejadores para el hardware moderno que apareció muchos años después de que se lo dejó de desarrollar.

7.6 Ventanas al núcleo

Todos los sistemas Unix proveen acceso al hardware a través de manejadores de dispositivos. Estos manejadores necesitan proveer puntos de entrada para aplicaciones en espacio de usuario y herramientas del sistema para acceder el hardware. Siguiendo la filosofía «todo es un archivo» de Unix, estos puntos de entrada se exponen en el espacio de nombres de archivos y se los llama **archivos especiales de dispositivos**, **archivos dispositivos**, **archivos especiales**, o **nodos de dispositivos** (*device node*). Son una interfaz para el manejador de dispositivo que aparece en el sistema de archivos como si se tratara de un archivo más. Esta prestación abstrae las particularidades inherentes del dispositivo y considera sólo la comunicación, de manera que las mismas llamadas a sistema que se utilizan para operar sobre archivos pueden utilizarse sobre estos archivos especiales, entonces al realizar una operación

de escritura sobre el archivo dispositivo de la impresora obtenemos nuestro impreso. Otro ejemplo lo tenemos si queremos reproducir un archivo de audio: lo logramos efectuando una escritura sobre el archivo dispositivo de nuestra placa de audio. También hay archivos especiales en Windows, como los hubo también en DOS, pero la idea no fue llevada al extremo, como lo fue en Unix.

Hay dos clases generales de archivos dispositivos en los sistemas operativos tipo Unix: los **orientados al carácter** y los **orientados al bloque**. La diferencia radica en cómo se leen y escriben los datos a los dispositivos: en los primeros, se utiliza un flujo de bytes, es decir, se acceden leyendo o escribiendo una **secuencia** de bytes, mientras que en los últimos se transfiere un **bloque** fijo de bytes al manejador; estos bloques se pueden direccionar de manera aleatoria.

Tradicionalmente, en UNIX, todos los archivos dispositivos se encuentran en el directorio `/dev`. Las distribuciones Linux pueden optar por adherir a las normativas de FHS (*Filesystem Hierarchy Standard*) que establece cuáles son los directorios, sus posiciones en el árbol de directorios y las funciones que deben cumplir. En este caso, FHS establece que el directorio `/dev` debe contener los nombres de los archivos dispositivos.

Obviamente, no todos los dispositivos pueden clasificarse en estas dos categorías; por ejemplo, el reloj. Linux agrega a estas categorías tradicionales la de dispositivos orientados a **sockets** de Internet, donde un socket no es un dispositivo sino una forma de comunicación entre procesos.



A menudo se le suele llamar socket (*socket address*) a la combinación de una dirección IP y un número de puerto, siendo una de las puntas de un flujo de comunicación entre procesos locales o remotos y, en este caso, a través de una red de computadoras que utilizan el protocolo IP. Según W. R. Stevens (1993), este término apareció en la especificación original de TCP (RFC 793 editado por Postel (1981)) y posteriormente comenzó a utilizarse como el nombre de la interfaz de programación (API) derivada de Berkeley. Pensemos en un zócalo o enchufe (*socket*) de pared para electricidad. Cualquier aparato eléctrico se puede enchufar en él. Por analogía, las llamadas al sistema de la interfaz de *sockets* permiten a una aplicación crear un *socket*, conectar un *socket* local a una dirección remota (que «enchufa» esta aplicación en un *socket* creado por otra aplicación), detectar si una aplicación remota se «enchufa» en el *socket* local, y enviar y recibir paquetes para la conexión. En cambio, se le suele llamar **par socket** (*socket pair*) a la cuádrupla que consiste de la dirección IP del cliente, el número de puerto del cliente, la dirección IP del servidor y el número de puerto del servidor, y es el que especifica a las dos puntas terminales que identifican unívocamente cada conexión IP en una internet.[1]

Además, existen otros archivos de dispositivos en Linux en el directorio `/dev` a los que se les suele llamar **dispositivos virtuales** porque no se utilizan para comunicarse con un dispositivo real.

Los sistemas Unix convencionales manejan el concepto de **número de dispositivo**. A cada instancia de un manejador y componente hardware se le asigna un número único de dispositivo. Dentro del núcleo, este número de dispositivo se usa para referirse al hardware y

a la instancia del manejador. El número de dispositivo está codificado en el nodo[2] dispositivo correspondiente y éstos se almacenan en sistemas de archivos normales. Para ayudar en la administración de números de dispositivos se los divide en dos componentes: los números **mayores** y **menores**. A cada manejador se le asigna un número mayor y usa el número menor para determinar qué instancia particular de hardware se está accediendo.

Los archivos de dispositivos, por convención, residen en el directorio `/dev`, como dijimos, y era el administrador del sistema el encargado de crear el nodo apropiado del dispositivo antes de acceder a la pieza de hardware[3]. Como archivos que son, además de tener los atributos comunes con los archivos ordinarios, como un nombre, permisos, propietario, grupo, tamaño, fechas de creación y modificación; también se les atribuye un **tipo** (orientado al carácter o bloque), y los números mayor y menor. El número mayor (*major number*) identifica al tipo de objeto y el número menor (*minor number*) identifica a un dispositivo específico dentro del grupo de los dispositivos que comparten el mismo número mayor. La lista completa de los números y los dispositivos está en un archivo de documentación del código fuente del núcleo de Linux, que es `Documentation/devices.txt` y, según ahí mismo se indica, también en *The Linux Assigned Names And Numbers Authority* `http://www.lanana.org/docs/device-list/` y en `ftp://ftp.kernel.org/pub/linux/docs/device-list/`

Estamos en el punto en el que comienza a vislumbrarse la diferencia entre el tratamiento de un archivo normal y el de un archivo dispositivo. Como decíamos, esta interfaz nos permite tratar a los dispositivos como si se tratara de archivos, pero no todas las operaciones son aplicables a dispositivos y, además, a éstos es preciso aplicarles otro tipo de operaciones para controlarlos. La norma POSIX define a la llamada a sistema `ioctl()` para controlar dispositivos, pero la implementación Linux difiere ligeramente, su prototipo es:

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...)
```

En la que `d` es un descriptor de archivo abierto, por ejemplo el devuelto por una llamada a sistema `open()`. El segundo argumento `request` es un código de solicitud dependiente del hardware y el tercer argumento es un puntero sin tipo a la memoria. Por ejemplo, la siguiente podría ser una llamada hipotética para ajustar el volumen de una placa de sonido:

```
ioctl(fd, MIXER_WRITE(SOUND_MIXER_MIC), &vol);
```

Los dispositivos de red no son dispositivos orientados al bloque ni al carácter, por lo tanto, no tienen un nombre en el directorio `/dev`. Para comunicarse con un dispositivo de red, por ejemplo para hacer llamadas a sistema con `ioctl()`, debemos utilizar un **descriptor de socket**, en lugar de un descriptor de archivo.

La forma tradicional que tenía Unix –y que Linux mantuvo hasta su versión 2.3– de relacionar los números mayores y menores con su nombre en el directorio `/dev` y su asociación con el manejador de dispositivo en el núcleo del sistema operativo era a través de una llamada a sistema. En Linux se efectuaba mediante las llamadas `register_blkdev()` o `register_chrdev()`, según se quisiera registrar el nombre de un archivo dispositivo orientado al bloque o al carácter, respectivamente, y especificando como argumento un número

2. Se le llama nodo porque es una estructura de datos que contiene el nombre, los números y el tipo.
3. Mediante el comando `mknod`.

entero sin signo de ocho bits (es decir, entre 0 y 255). En el núcleo de Linux los números de dispositivo se almacenan en enteros de 16 bits, 8 bits se le asignan al número mayor y 8 bits al número menor, entonces a cada manejador le corresponde uno de los 256 números mayores posibles.

Obviamente, existían cientos de archivos dispositivo poblando el directorio `/dev`, la mayoría de los cuales no se utilizaba, sólo estaban ahí por si alguna vez se los necesitaba. Los números válidos que se podían utilizar eran los «oficiales» reconocidos en `Documentation/devices.txt`, como habíamos mencionado. De manera que si un programador desarrollaba un manejador para un dispositivo nuevo, podía utilizar cualquier número que no estuviera previamente asignado si tal desarrollo no salía de su computadora, pero si pretendía que ese número fuera incorporado al núcleo Linux estable, debía contactar a los otros desarrolladores del núcleo y solicitar un número «oficial».[4] En realidad el nombre del dispositivo era irrelevante, sólo el número era importante, porque era el que el núcleo manejaba internamente y era el que debía corresponderse exactamente con el del nodo dispositivo identificado por estos números mayor y menor.

Cada manejador debe registrar el número mayor que desea utilizar y los **métodos de operación del manejador** que se deben llamar cuando se requiere realizar las operaciones de archivo para ese manejador. Estas operaciones incluyen, por ejemplo, abrir, cerrar, leer y escribir. Los métodos del manejador se almacenan en una tabla indexada por número mayor para uso posterior. A esta tabla se la llama **tabla mayor** (*major table*).

Cuando se abre un nodo de dispositivo (es decir, cuando un programa ejecuta la llamada al sistema `open()` en ese nodo), se extrae el número mayor y se usa para indexar en la tabla mayor y determinar los métodos del manejador. Estos métodos se graban en la **estructura del archivo**, que es la representación en el núcleo del nuevo mando (*handle*) del descriptor de archivo dado al proceso.

Las operaciones siguientes sobre el descriptor de archivo (tales como lectura o escritura) darán como resultado que sean invocados los métodos del manejador y así es como el control es transferido al manejador. Entonces el manejador usará el número menor para determinar qué instancia del hardware se debe manipular.

7.6.1 Desventajas

Este enfoque, si bien es tradicional y sencillo, tenía varias desventajas, que fueron consideradas por Gooch (2001) y que podemos resumir brevemente. Los números mayor y menor existentes estaban limitados a 8 bits cada uno. Por ejemplo, el manejador de discos SCSI originalmente consumió un número mayor: 4 bits se asignaron al índice de la partición y se dejaron 4 bits para el índice del disco, por lo tanto, sólo se soportaron 16 discos. Un cambio posterior reservó otros 7 números mayores, lo que incrementó el número de discos a 128. Si bien era suficiente para empresas medianas, era insuficiente para grandes sistemas. Los grandes arreglos de almacenamiento pueden presentar actualmente miles de volúmenes lógicos.

El esquema convencional requería la asignación de números mayores y menores de dispositivos. Esto significa que se necesitaba de una autoridad central que coordinara la entrega de estos números de dispositivos. El nombre de cada nodo dispositivo también debía ser coordi-

4. Peter Anvin era la autoridad responsable de la asignación de números y nombres en el núcleo Linux.

nado con esta autoridad central, de manera que las aplicaciones se refirieran al mismo nombre de dispositivo y los administradores de sistemas y las distribuciones de Linux pudieran crear los nodos de dispositivo aprobados.

Gooch (2001) observó mucha duplicación de información almacenada: los nodos de dispositivo se almacenaban en el sistema de archivos (no volátil) y los debía crear el administrador de sistemas. Para ello teníamos el programa `/dev/MAKEDEV`[5] y cada vez que era necesario efectuar un cambio en cualquiera de los manejadores de dispositivo, era necesario también efectuar el cambio en este programa. Había varias bases de datos separadas de números mayores, menores y nombres de dispositivos. Los números y los nombres de dispositivo estaban almacenados en el programa `/dev/MAKEDEV` y en el directorio `/dev` en muchos millones de computadoras. Además, los números de dispositivo estaban almacenados dentro del código fuente del núcleo y el manejador; y los nombres de dispositivo estaban almacenados en el código fuente de miles de aplicaciones que accedían a los nodos de dispositivos.

Un directorio `/dev` típico tenía más de 1200 nodos y, en la mayoría de los sistemas, estos dispositivos simplemente no existían porque el hardware no estaba conectado o ya ni siquiera estaba disponible. Estaban definidos para proveer la posibilidad de que fuera instalado el hardware. Un `/dev` inmenso incrementaba el tiempo para acceder a los dispositivos porque las operaciones de búsqueda de directorio y nodo-i necesitaban leer más datos del disco.

Otra desventaja que podemos mencionar respecto al enfoque tradicional es que el orden en el cual el usuario conecta el dispositivo determina cuál archivo de dispositivo le asigna a cada dispositivo el núcleo. Por ejemplo, el primer disco SCSI detectado queda vinculado al archivo de dispositivos `/dev/sda`, el siguiente disco SCSI detectado quedará vinculado al `/dev/sdb` y así sucesivamente. Entonces, con este enfoque, el mismo dispositivo es accesible a través de diferentes archivos de dispositivos en diferentes momentos y el sistema los montará en diferentes directorios.

7.6.2 devfs

La solución presentada por Gooch (2001) para paliar estos problemas fue el desarrollo de **devfs** o *device file system*.

En `devfs` los manejadores de dispositivos en el núcleo podían registrar dispositivos por su nombre en lugar de hacerlo por su número y estos dispositivos registrados aparecían automáticamente en el sistema de archivos. Es decir que, si bien se mantenían los números por compatibilidad, ya no era necesario el trámite de solicitud de los mismos, era el nombre el que importaba. El sistema de archivos les permitía a los manejadores crear, modificar y borrar entradas.

El sistema se completaba con un proceso demonio (del inglés *daemon*) que ejecutaba en espacio de usuario, el **devfsd**, que permitía desde crear espacios de nombres de dispositivos virtuales hasta administrar dispositivos *hotplug*. Richard Gooch comenzó a desarrollar a `devfs` en enero de 1998 y fue aceptado para su inclusión en la versión 2.3.46 del núcleo Linux en febrero de 2000.

No obstante estas ventajas respecto del enfoque tradicional, `devfs` no tuvo la aceptación esperada porque el núcleo decidía cómo se llamarían los archivos en `/dev` (en lugar de

5. Es un script que hace numerosos llamados al programa mknod, como vimos en el apartado 7.6.

decidirlo el administrador), forzando la política de nombres en el núcleo[6], pero la principal desventaja eran las condiciones de carrera que se presentaban y que hubieran requerido numerosas modificaciones al núcleo para que no ocurrieran. Un análisis completo puede verse en el trabajo de Kroah-Hartman (2003).

7.6.3 Conexión en caliente

La conexión en caliente (*hotplug*) comenzó a implementarse en el núcleo Linux durante el desarrollo de la versión 2.4, en enero de 2001, para permitir la conexión en caliente de los dispositivos PCMCIA, CardBus y Firewire. Casi simultáneamente se empezó a desarrollar una implementación para el pasado (pero no abandonado) núcleo versión 2.2. Si un núcleo soporta *hotplug* tendrá un archivo llamado `hotplug` dentro del directorio `/proc/sys/kernel`. Este archivo contendrá la ruta completa al comando que el núcleo invocará para manejar el «evento»; es decir, la conexión o desconexión de un dispositivo. Puede llegar a contener, por ejemplo, la línea `/sbin/hotplug`, que es el programa que ejecuta en espacio de usuario. Ya en la versión 2.6 está totalmente integrada y disponible para todas las clases de manejadores y colectores adaptados a `sysfs`. Greg Kroah-Hartman implementó para la versión 2.4.15 del núcleo Linux un manejador para la conexión en caliente de dispositivos en el colector PCI, al que llamó `pcihpfs` y que, al ser montado, permitía obtener un árbol con los directorios llamados 3, 4, 5, etc., que eran los número físicos de las ranuras (*slots*) de expansión del colector PCI. Se podía leer cada archivo en el directorio de la ranura correspondiente para encontrar el valor de ese bit de información acerca de la ranura. A los archivos `power` y `attention` se los podía escribir con el objeto de establecerles el bit en 1 o en 0, para conectarlo o desconectarlo (eléctricamente). Es decir, un comando como:

```
echo 1 > 5/power
```

Desde la raíz donde estaba montado `pcihpfs`, encendía el suministro eléctrico en la ranura número cinco. Para mayores detalles se puede consultar el trabajo completo de Kroah-Hartman (2002).

7.6.4 El nuevo modelo de dispositivos de Linux

En la versión de desarrollo 2.5 del núcleo Linux, se implementó un modelo unificado de manejadores de dispositivos que hizo más fácil el desarrollo de manejadores en la posterior versión 2.6. Mochel (2003a) fue quien creó el marco de desarrollo de un modelo unificado de manejadores de dispositivos en esta serie de desarrollo. La intención original fue la de proveer un medio de representación y operación genéricos para todos los dispositivos de una computadora.

Este marco consta de un número de funciones y estructuras comunes a las que todos los subsistemas de manejadores de dispositivos se han debido adaptar para poder usarlas. También consta de algunas estructuras genéricas que se usan fuera del código del manejador en otras partes del núcleo.

6. Recordemos que sólo debería manejar mecanismos, según lo visto en 7.3.2.

El modelo de manejadores y `sysfs` están muy relacionados, al punto tal que muchas veces se los confunde, pero son entidades distintas: `sysfs` es el sistema de archivos virtual que provee una representación visible en espacio de usuario del modelo de manejadores, el modelo funciona muy bien sin `sysfs`, pero no a la inversa.

7.6.4.1 Colectores, dispositivos y clases

La propuesta de Mochel desglosa todo en **colectores, dispositivos y clases**. Usando estas primitivas, controla cómo los manejadores se relacionan con los dispositivos físicos y virtuales y le muestra al usuario cómo están interconectadas todas estas cosas.

Un colector puede ser descrito como algo con dispositivos conectados, un dispositivo en sí mismo que sirve como punto de conexión para otros dispositivos. Ejemplos de colectores son el PCI, el USB, el i2e, PCMCIA y SCSI. Por lo general, sólo un **manejador de colector** (*bus driver*) controla la actividad en el mismo y provee un tipo de **puente** (*bridge*) desde el colector en el que está al colector que controla.

Un ejemplo de puente es el controlador USB que reside en el colector PCI: éste se comunica con el colector PCI como dispositivo PCI y se muestra ante el núcleo como un manejador PCI, pero controla todos los accesos a ese colector USB específico comunicándose con los diferentes dispositivos USB enchufados al mismo.

Los colectores están representados en el núcleo mediante la definición `struct bus_type`, que se encuentra en `include/linux/device.h`. Todos los colectores en el sistema se muestran al usuario en forma de subdirectorios dentro del directorio `/sys/bus` del sistema de archivos `sysfs`, que ya tratamos en el apartado 6.2.3.6.3.

Los dispositivos en este modelo son los dispositivos físicos o virtuales que residen en el colector. Están representados por la definición `struct device`, y son creados por el colector cuando éste detecta que están presentes en el sistema. Por lo general, sólo un manejador controla a un dispositivo específico en un instante dado. Pueden verse en el directorio `/sys/devices` como un inmenso árbol de todos los dispositivos en el sistema o en el directorio `/sys/bus/TIPO-DE-COLECTOR/devices/` para un tipo específico de dispositivo.

Los dispositivos también tienen sus propios manejadores asignados, que controlan cómo comunicarse con el dispositivo a través de un colector específico. Algunos manejadores saben cómo comunicarse con varios colectores, por ejemplo, el manejador del adaptador de red Tulip que puede comunicarse con los dispositivos Tulip conectados al colector ISA o PCI.

La clase, en este contexto, no sigue la definición clásica de la Programación Orientada a Objetos, sino que es más bien algo que provee una función a un usuario y de la que se espera que se desempeñe de cierta manera, el usuario debería percibirla como un tipo de dispositivo, por ejemplo dispositivos de audio, discos, puertos seriales o terminales. Las clases están representadas en el núcleo Linux mediante la definición `struct device_class`, y pueden verse como subdirectorios del sistema de archivos `sysfs` debajo de `/sys/class/`.

7.6.5 sysfs

Tal como decíamos en el apartado 6.2.3.6.3, el sistema de archivos `sysfs`, que fue desarrollado a partir del año 2001, enumera los dispositivos y canales conectados al sistema en una jerarquía de archivos a la que se puede acceder desde el espacio del usuario. Está dise-

ñado para manejar las opciones específicas al dispositivo y al controlador que previamente residían en `/proc`, y cubre la adición dinámica de dispositivos ofrecida anteriormente por `devfs`. Mochel (2003b) lo llamó originalmente `ddfs` (*Device Driver Filesystem*) y fue escrito para depurar el nuevo modelo de manejadores a medida que se desarrollaba, y al que nos referiremos más adelante. Ese código de depuración había usado originalmente a `procfs` pero, a pedido de Linus Torvalds, se lo convirtió para que utilizara un sistema nuevo de archivos basado en `ramfs`. Para la época en la que el nuevo modelo de manejadores fuera integrado en la versión 2.5.1 del núcleo de Linux, se le había cambiado el nombre a `driverfs` para que fuera un poco más descriptivo. Durante el siguiente año de desarrollo de la versión 2.5, las capacidades de infraestructura del modelo de manejadores y `driverfs` comenzaron a demostrar su utilidad a otros subsistemas de entrada y salida. Se desarrolló entonces el concepto de `kobject` en el núcleo para proveer un mecanismo central de administración de objetos y `driverfs` fue convertido a `sysfs` para representar su independencia en los subsistemas.



Un `kobject` es una representación de datos relevantes a cualquier objeto encontrado en el sistema, en un lenguaje orientado a objetos sería la clase a partir de la cual heredan otras clases. Los `kobject` contienen los atributos que, se espera, necesiten la mayoría de los objetos del sistema, a saber, un nombre, un contador de referencias, un apuntador a un padre (lo cual permite organizarlos en una estructura jerárquica) y un tipo. Casi cualquier objeto relacionado con el modelo de dispositivos tendrá un `kobject` enterrado profundamente en algún lugar dentro de él. En el código fuente del núcleo Linux un `kobject` está definido mediante una declaración `type struct`.

El sistema de archivos `sysfs` puede ser montado como cualquier otro; el directorio estándar de facto es `/sys` adoptado por la mayoría de las distribuciones. En el nivel máximo del punto de montaje de `sysfs` hay directorios. Estos directorios representan los subsistemas principales que están registrados con `sysfs`, como se ve en la figura 7.8 y como ya se adelantara en la tabla 6.7. Estos directorios se crean al iniciarse el sistema cuando los subsistemas se registran a sí mismos con el núcleo de `kobject`. Después de que están inicializados comienzan a descubrir objetos que están registrados dentro de sus respectivos directorios. Veremos a continuación el significado de estos directorios, según se muestran en la figura 7.8.

```
/ sys /
| -- block
| -- bus
| -- class
| -- devices
| -- firmware
| -- module
, -- power
```

Fig. 7.8 – Directorios en `/sys`.

El directorio `/sys/block` contiene subdirectorios para cada dispositivo de bloques que haya sido descubierto en el sistema. En cada directorio de dispositivo de bloque hay atribu-

tos que describen varias cosas, incluso el tamaño del dispositivo y el número de dispositivo (`dev_t`, en la estructura de datos definida en el código fuente) al cual apunta. Cada partición de cada dispositivo de bloques está representada como un subdirectorio del dispositivo de bloques. Incluidos en estos directorios hay atributos de sólo lectura acerca de las particiones.

El directorio `/sys/bus` contiene subdirectorios para cada tipo de colector físico que tenga soporte registrado en el núcleo, ya sea estáticamente compilado o cargado a través de un módulo. Cada tipo de colector representado tiene dos subdirectorios: `devices` y `drivers`. El directorio `devices` contiene un listado de cada dispositivo descubierto en ese tipo de colector en todo el sistema. Los dispositivos listados son, en realidad, enlaces simbólicos que apuntan al directorio del dispositivo en el árbol global de dispositivos.

El directorio `drivers` contiene directorios para cada manejador de dispositivo que haya sido registrado con el tipo de colector. Dentro de cada directorio de manejador hay atributos que permiten ver y manipular parámetros del manejador y enlaces simbólicos que apuntan a los dispositivos físicos (en el árbol global de dispositivos) a los que el manejador está ligado.

```
class /
|-- graphics
|-- input
|-- net
|-- printer
|-- scsi_device
|-- sound
`-- tty
```

Fig. 7.9 – Clases en `/sys`.

El directorio `class` contiene representaciones de cada clase de dispositivo que está registrado en el núcleo, entendiendo por clase de dispositivo el que describe un tipo funcional de dispositivo, por ejemplo, los que se muestran en la figura 7.9. Cada clase de dispositivo contiene subdirectorios para cada clase de objeto que haya sido asignado y registrado con esa clase de dispositivo. Para la mayoría de los objetos de esa clase de dispositivo, en sus directorios hay enlaces simbólicos a los respectivos directorios del dispositivo y manejador (en la jerarquía global de dispositivos y en la de colectores respectivamente) que están asociados con esa clase de objeto.

El directorio `devices` contiene la jerarquía global de dispositivos, cada dispositivo físico que haya sido descubierto por el tipo de colector registrado en el núcleo. Los representa de una manera ancestral: donde cada dispositivo se muestra como un dispositivo subordinado del dispositivo al cual está físicamente (eléctricamente) subordinado. Hay dos tipos de dispositivo que son una excepción a esta norma, los dispositivos periféricos inherentes a una plataforma en particular y los dispositivos de sistema. Los dispositivos de una plataforma en particular tienen algunos puertos de entrada y salida o MMIO que existen en una ubicación conocida y fija, por ejemplo antiguos dispositivos de la arquitectura x86 tales como un controlador serial, el controlador de una disquetera o dispositivos embutidos de un SoC (*system on chip*). Los dispositivos de sistema (*system devices*) son dispositivos no periféricos que son componentes integrales del sistema y no se semejan a ningún otro dispositivo. Pueden tener algún registro en hardware para acceder a su configuración pero no disponen de la capacidad para transferir

datos, por ejemplo los temporizadores (*timers*) y APIC (*Advanced Programmable Interrupt Controller*).

El directorio `firmware` contiene interfaces para ver y manipular objetos específicos de *firmware* y sus atributos. En este contexto, el término *firmware* se refiere al código específico de la plataforma que se ejecuta al encender el equipo, como ocurre con la BIOS de los equipos x86, OpenFirmware en plataformas PPC y EFI en las plataformas ia64.

El directorio `module` contiene subdirectorios para cada módulo cargado en el núcleo. El nombre de cada directorio es el nombre del módulo, que puede ser tanto el nombre el archivo como el nombre interno del mismo.

El directorio `power` representa al subsistema de alimentación eléctrica. Contiene dos atributos: `disk` que controla el método por el cual el sistema se suspenderá (volcando la información a disco), y `state` que le permite a un proceso hacer ingresar el equipo a un estado de bajo consumo eléctrico. Esta es una prestación que utilizan mucho las notebooks y netbooks.

Como complemento, `sysfs` provee una biblioteca escrita en lenguaje C llamada `libsysfs` para proporcionar una interfaz que permita acceder a objetos y atributos en `sysfs`. Por tratarse de una pieza importante de la infraestructura del núcleo Linux en la versión 2.6, varios programas, ya sea que ejecuten en espacio de usuario o de núcleo, utilizan a `sysfs`, por ejemplo `udev` que veremos a continuación.

7.6.6 udev

A partir de la versión de desarrollo 2.5 del núcleo de Linux, todos los dispositivos físicos y virtuales en un sistema son visibles en espacio de usuario de una forma jerárquica gracias a `sysfs` y, por otra parte, el programa `/sbin/hotplug` provee una notificación a espacio de usuario cuando se agrega o quita un dispositivo del sistema. Gracias al uso de estas dos prestaciones, ahora es posible una implementación en espacio de usuario de un `/dev` ágil, que puede proveer una política de nombres de dispositivos muy dinámica. El programa `udev`, desarrollado por Kroah-Hartman (2003), reemplaza la funcionalidad provista por `devfs` proporcionando entradas en `/dev` sólo para los dispositivos que están presentes en el sistema en un momento dado, y provee prestaciones con las que no se podía contar previamente con `devfs`, tales como:

- Nombres fijos o persistentes de dispositivos cuando se mueven en el árbol de dispositivos. Por ejemplo, se puede llamar `camara` a una cámara fotográfica que se conecta a algunos de los puertos USB, indistintamente de cuál sea.
- Notificación de sistemas externos de cambios de dispositivos; mediante mensajes D-Bus para que cualquier programa pueda enterarse cuando un dispositivo se conecta o desconecta.
- Un esquema flexible de nombres de dispositivos, por ejemplo `camara`, como ya se dijo.
- Le permite al núcleo el uso de números mayores y menores dinámicos. El programa `udev` no usa el número mayor o menor para reconocer al dispositivo, puede funcionar con números al azar, por lo tanto ya dejó de ser una preocupación el agotamiento de los números mayores y menores.
- Mueve toda la política de nombres fuera del núcleo.

Es un programa de unos 70K que ejecuta enteramente en espacio de usuario, sin código dentro del núcleo Linux, por lo tanto puede intercambiarse a disco. Opera disparado por **eventos** en el núcleo (`uevent`).



Un evento es un mensaje de software que indica que algo ha pasado, por ejemplo al presionar una tecla o al hacer clic con el *mouse*; es una acción que se inicia fuera del ámbito de un programa pero que es manejada por una pieza de código dentro del mismo.

El programa `udev` sigue **reglas** que están escritas en archivos de texto en el directorio `/etc/udev/rules.d/`. Cuando el proceso demonio `udevd` recibe un `uevent` compara sus reglas configuradas con respecto a los atributos de dispositivo disponibles en el sistema `sysfs`. Si encuentra una coincidencia entre las reglas y la información disponible en `sysfs`, realiza ciertas acciones tales como crear o borrar el nodo de dispositivo, instalar *firmware*, etc.

7.6.7 HAL en Linux

La capa de abstracción de hardware o HAL (del inglés *hardware abstraction layer*) es un elemento del sistema operativo que funciona como una interfaz entre el software y el hardware del sistema. Esta capa abstracta de hardware permite que las aplicaciones se independicen del hardware abstrayendo información tal como las cachés, los colectores de entrada y salida y las interrupciones.

El proyecto HAL en Linux partió de una idea propuesta por Pennington (2003) y desarrollada posteriormente por el proyecto Freedesktop.org. En HAL existe un proceso denominado `hald` que es el responsable de descubrir, enumerar y mediar el acceso a la mayoría del hardware. Las aplicaciones se comunican con HAL a través de mecanismos de comunicación entre procesos (o IPC, acrónimo del inglés *Inter-process Communication*) que abstraen el hardware detrás de mecanismos RPC (*Remote Procedure Call*) basados en objetos.

Cada dispositivo hardware lógico está representado por un objeto D-Bus y su dirección de colector se usa como identificador único (UDI o *Unique Device Identifier*). Por dispositivos se entienden abstracciones tales como particiones de disco y redes inalámbricas visibles. La funcionalidad de un dispositivo está expuesta a través de **interfaces** D-Bus y sus estados se acceden a través de **propiedades**, es decir, un conjunto de pares «clave-valor».

HAL difunde eventos de hardware como **señales** en estos objetos de manera que las aplicaciones pueden escucharla para reaccionar frente a eventos tales como una cámara digital que se enchufa, un disco óptico que se coloca o la tapa de una notebook que se cierra.

En Linux, HAL utiliza al sistema de archivos que reside en `/sys` para descubrir hardware y escuchar eventos de conexión en caliente (*hotplug*) en el núcleo. También se suele utilizar a HAL en combinación con `udev`. Pero desde el año 2009 se está en proceso de dejar de lado a HAL migrando sus funciones a `udev`, ya sea al mismo programa `udev` a su biblioteca `libudev` o su paquete de programas accesorios `udev-extras`.

7.6.7.1 DeviceKit

Es también una capa de abstracción de hardware que intenta reemplazar a HAL. En mayo de 2008 el desarrollador de HAL, David Zeuthen, ya había anunciado su intención de dejar de lado a HAL debido a su complejidad y redundancia con otras bibliotecas Linux. DeviceKit se encargaría de realizar sólo las funciones que no están presentes en esas bibliotecas: la enumeración de los dispositivos existentes, señalizar la conexión y desconexión de dispositivos y combinar y clasificar la información del hardware disponible en un solo punto. A partir de ese momento algunas distribuciones comenzaron a incluir a DeviceKit. Pero con el correr del tiempo, algunas de las funcionalidades de DeviceKit también se están desplazando a `udev`, por lo que se espera que HAL y DeviceKit desaparezcan por completo como tales porque su tarea quedará absorbida totalmente por `udev`.

7.6.8 D-Bus

D-Bus (*Desktop Bus*) es un sistema de comunicación entre procesos (IPC). Su diseño tiene tres capas: una biblioteca (`libdbus`), un proceso demonio –que utiliza a la biblioteca `libdbus`– y envoltorios (*wrappers*) de bibliotecas para su uso en ambientes específicos de desarrollo.

Para la comunicación utiliza objetos, que pueden estar definidos en diferentes lenguajes de programación como por ejemplo en C++ o Python. Normalmente, en los lenguajes de programación se utilizan punteros para hacer referencia al destino del mensaje que se implementan como direcciones de memoria relativas al espacio de direcciones de la aplicación, por lo cual no pueden accederse de una aplicación a otra. En cambio D-Bus utiliza un nombre para cada objeto, este nombre se expresa como si se tratara de una ruta en un sistema de archivos, por ejemplo `/org/kde/3/4/5`. Los desarrolladores que utilicen D-Bus son libres de crear objetos y nombrarlos como lo deseen, estos nombres se aglutinan en un espacio de nombres, de manera ordenada, como se desarrollará en el apartado 6.1.3, si bien los espacios de nombres generalmente están prestablecidos en función de los componentes utilizados por los propios programadores, por ejemplo, `/org/kde`.

D-Bus se utiliza tanto para la comunicación entre aplicaciones de escritorio en la misma sesión como para la comunicación entre el sistema operativo –ya sea el núcleo mismo del sistema operativo como los procesos demonios ejecutados por el núcleo– y la sesión de escritorio.

7.6.8.1 Netlink sockets

Los *sockets* (zócalos, enchufes) netlink son otra forma de comunicación entre procesos que se utilizan para transferir información entre el núcleo y los procesos que ejecutan en espacio de usuario. Proveen un enlace de comunicación *full-duplex* entre los dos mediante interfaces (API) de *socket* estándares, para los procesos que ejecutan en espacio de usuario, y llamadas a API especiales del núcleo para sus módulos. Utilizan la familia de direcciones AF_NETLINK en lugar de la AF_INET utilizada por los sockets de TCP/IP. Cada prestación de los *sockets* netlink define su propio tipo de protocolo en el archivo de cabecera del núcleo `include/linux/netlink.h`.

Lo que motiva a utilizar *sockets* netlink en lugar de las tradicionales llamadas a sistema o el sistema de archivos `/proc` para la comunicación entre los espacios de memoria del núcleo y de usuario, es que no es tan sencillo agregar nuevas llamadas a sistema o archivos en `/proc` sin correr el riesgo de contaminar y dañar la estabilidad del sistema. Los *sockets* netlink de Linux están documentados en el RFC 3549 publicado por Salim y col. (2003).

7.6.9 Cómo funciona todo junto

El software del nuevo modelo de dispositivos de Linux viene sufriendo muchas modificaciones en todas las versiones de la serie 2.6, por lo que ahora intentaremos hacer una descripción acerca de cómo interactúan sus componentes a sabiendas de que no puede ser una descripción acabada ni definitiva. El modelo de desarrollo de software de Linux, que enfatiza la importancia de publicar temprana y frecuentemente (*release early, release often*), provoca una revolución que muchas veces no deja tiempo para documentar adecuadamente las ideas que apoyaron un cambio en el código. Muchas de estas ideas, propuestas de mejoras, han sido brevemente comentadas e ilustradas mediante trozos de código en lenguaje C en los correos electrónicos intercambiados entre los distintos desarrolladores del código del núcleo Linux, y que cualquiera puede leer en sitios en Internet, pero no cualquiera puede entender de qué se está discutiendo. Las propuestas que mostraron ser suficientemente sólidas y perseveraron durante un cierto tiempo fueron expuestas al mundo académico a través de congresos.

En este sentido, Zeuthen (2007) explicó cómo se compagina la labor de cada uno de los componentes, tanto en el núcleo como fuera de él, describiendo hasta qué función cumplen las aplicaciones de escritorio con las que interactúa el usuario, mostrando cómo es la interfaz con el usuario desde la línea de comandos y desde las distintas interfaces gráficas en Linux.

La carga de un módulo, el presionar una tecla o el cerrar la tapa de la notebook disparan eventos en varios componentes. Por ejemplo, al cargar un módulo, el manejador llama a la función `kobject\_uevent()` o `kobject\_uevent\_env()` y, utilizando sockets netlink, los eventos se transmiten a `udevd`. HAL lee los eventos desde el socket que reside en `/org/f` `reedesktop/hal/udev\_event`, luego HAL crea objetos de dispositivos y emite la señal `DeviceAdded`. A partir de ahí las aplicaciones reciben las señales desde HAL a través de D-Bus. El esquema general de este proceso lo podemos ver en la figura 7.10

Rebe, Frommel y Brendel (2006) muestran otra disposición que se ilustra en la figura 7.11: cuando se conecta o desconecta un dispositivo, el controlador del colector (ISA, USB o PCI) señala el evento mediante una interrupción. En respuesta a ella, el núcleo indaga los detalles del dispositivo añadido recientemente ya sea mediante la utilización de un protocolo específico para consultar al controlador del hardware o mediante la búsqueda de información almacenada en direcciones preestablecidas (*hard coded*), tales como el área de configuración de una tarjeta PCI; más específicamente busca los identificadores (ID) del fabricante y del producto. El núcleo luego pasa a crear objetos `kobject` para cada dispositivo nuevo. A este objeto `kobject` se le pasan y almacenan datos acerca del tipo de dispositivo, por ejemplo si es orientado al carácter o al bloque, y los números mayor y menor, a través del espacio de nombres global que es parte del núcleo. Luego deja disponible esta información a través de `sysfs`. El programa udev maneja el evento directamente desde el socket netlink mediante la regla `RUN + = "socket:/org/freedesktop/hal/udev_event"`

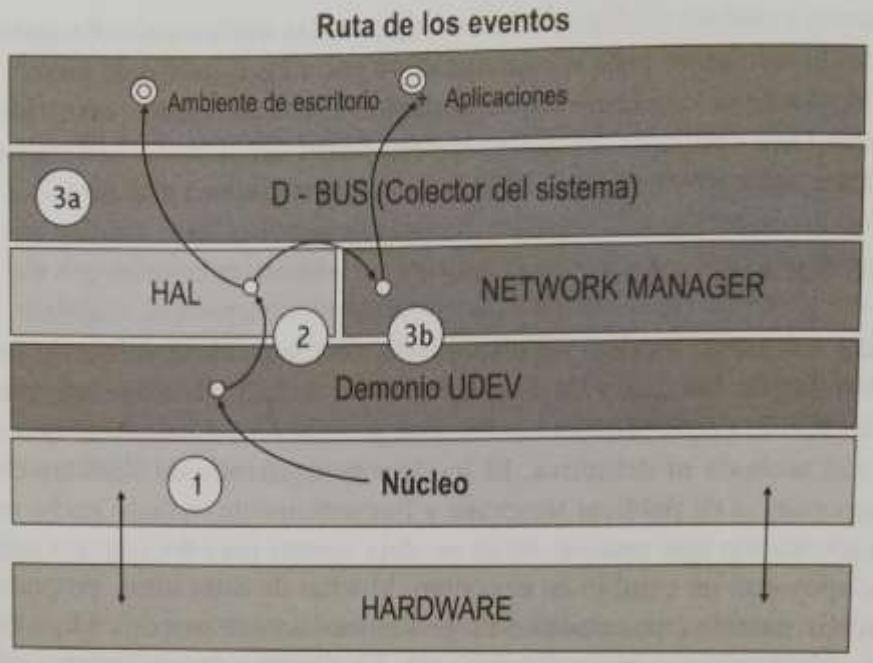


Fig. 7.10 – La ruta de los eventos.



Fig. 7.11 – Gestión dinámica de los dispositivos con `udev`.

Según Rebe, Frommel y Brendel (2006), los sistemas que usan D-Bus tienen dos colectores implementados mediante un único proceso servidor: el colector de sistema y el de sesión. el primero comienza a ejecutar durante el arranque, el segundo comienza a ejecutar cuando se crea una sesión con interfaz gráfica (GUI). La figura 7.11 muestra el rol que desempeñan estos dos colectores en las comunicaciones entre los componentes del sistema operativo. Los colectores de sesión les dan a las aplicaciones que pertenecen a la sesión de escritorio la capacidad para comunicarse entre ellas. Las aplicaciones pueden ser servicios provistos por el ambiente del escritorio. El colector del sistema está diseñado principalmente para ayudar a los programas de escritorio a comunicarse con las capas subyacentes. Por ejemplo, una aplicación puede usar al colector del sistema para registrarse con una clase de hardware tal como

una cámara digital. Por otro lado, Zeuthen (2007) muestra otra disposición que se ilustra en la figura 7.12.

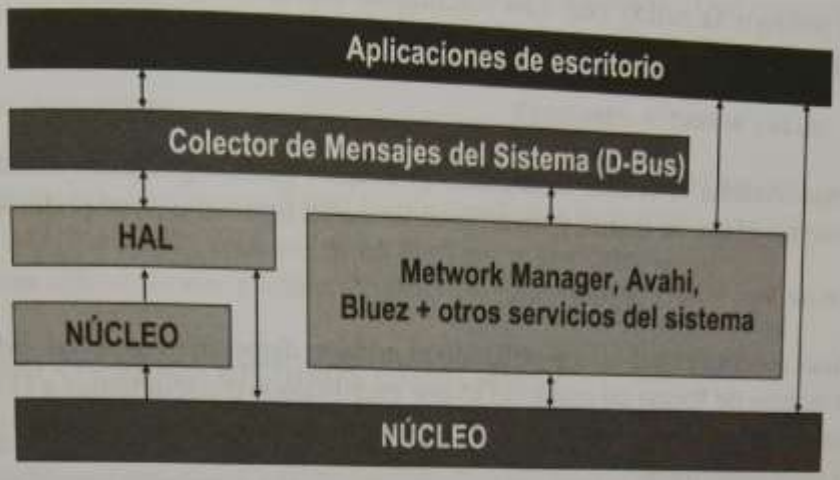


Fig. 7.12 – D-Bus en un sistema moderno.

7.7 Trabajos prácticos

Como decíamos al principio de la materia, UNIX modela a los dispositivos de entrada y salida como si fueran archivos, de esta forma las mismas operaciones (llamadas a sistema) utilizadas para acceder a un archivo (leer, escribir, buscar y posicionarse) se utilizan sobre los dispositivos. Entonces «imprimir» implica «escribir en la impresora», mostrar un carácter (o un texto) en la pantalla implica escribir en ella, para escuchar nuestro tema preferido de música tenemos que escribir sobre nuestro dispositivo de audio. Análogamente, hacemos operaciones de lectura del teclado y del *mouse*.

Estos «archivos dispositivos», por razones organizativas, están en el directorio `/dev`. Por ejemplo, si ingresamos al sistema como usuario «ubuntu» y colocamos el comando

```
$ ls -l /dev/tty1
```

vemos

```
crw--w---- 1 ubuntu tty 4,1 Oct 25 11:30 /dev/tty1
```

La «c» nos indica que es un dispositivo por caracteres, luego vienen los permisos que tiene el archivo, nosotros somos los propietarios del mismo y tenemos permiso de lectura y escritura, luego vienen los números mayor y menor (*major and minor numbers*). El mayor es el genérico y el menor el específico, es decir «4» significa tty, terminal o consola. Si ejecutamos:

```
$ ls -l /dev/tty2
$ ls -l /dev/tty3
```

etc.

Vemos que el número mayor es el mismo, pero cambia el menor.

El comando «echo» muestra una línea de texto por pantalla, por ejemplo:

```
$ echo "Hola, mundo"
```

Podemos redirigir la salida con «>» intentando escribir sobre otra terminal o consola (distinta a la nuestra):

```
$ echo "Hola, mundo" > /dev/tty3
```

y el sistema nos contesta «Permiso denegado» ¿Por qué? ¿Cómo podemos solucionarlo?

Si tenemos una placa de audio, y un archivo de audio tipo .au o .wav podemos hacer:

```
$ cat ding.wav > /dev/audio
```

En prácticas anteriores habíamos utilizado el archivo dispositivo especial «/dev/null» como «agujero negro» de forma tal que todo lo que escribíamos o copiábamos a él se perdía sin remedio, como si hubiésemos echado a la «Papelera de Reciclaje» de Windows y la hubiésemos vaciado simultáneamente.[7]

Análogamente tenemos un dispositivo «/dev/zero» que es un repositorio de ceros o mejor dicho de bytes nulos, que pueden ser útiles para generar rápidamente un archivo del tamaño que deseemos y cuyo contenido no nos importe demasiado, por ejemplo:

```
$ dd bs=1024 count=1 if=/dev/zero of=unka
```

El comando dd convierte y copia un archivo, la opción bs le indica cuántos bytes debe leer/escribir, la opción count le indica cuántos bloques debe copiar, la opción if le indica el archivo de entrada (en lugar de `stdin`) y análogamente la opción of el archivo de salida (en lugar de `stdout`).

Podemos utilizar este programa para obtener el sector de arranque (*boot*) de un disquete o disco rígido. Por ejemplo:

```
$ dd if=/dev/fd0 of=boot-fd bs=512 count=1
$ dd if=/dev/hda of=boot-hd bs=512 count=1
```

y podemos analizar estos archivos con el depurador (debug del DOS o gdb de Linux).

Y hablando de `stdin`, `stdout` y `stderr` si colocamos, por ejemplo:

```
$ ls -l /dev/stdin
```

vemos

```
lrwxrwxrwx 1 root root 17 Jun 14 17:21 /dev/stdin->../proc/self/fd/0
```

y si le seguimos la pista al enlace

```
$ ls -l /proc/self/fd/0
```

vemos

7. Si hacemos clic con el botón derecho del *mouse* sobre el icono de la «Papelera de Reciclaje» y seleccionamos «Propiedades», podemos seleccionar la opción «No mover archivos a la Papelera, purgarlos al eliminarlos» y no seleccionar «Mostrar diálogo para confirmar eliminación» obtendrá un comportamiento similar a «/dev/null».

```
lrwx------ 1 ubuntu ubuntu 64 Oct 25 18:26 /proc/self/fd/0->/dev/tty1
```

Pruebe con `/dev/stdout` y `/dev/stderr` y siga los enlaces ¿Qué conclusiones saca?.

Normalmente hay enlaces (nombres) simbólicos de los dispositivos «mouse», «cdrom», «modem» (si tiene uno). Pruebe con:

```
$ ls -l /dev/mouse
```

Si tiene un *mouse* serial conectado al primer puerto serial (COM1 en DOS) verá que es un enlace a «/dev/ttyS0» (es decir el Serial 0). Recuerde que es común ver que las numeraciones comiencen en 0 ¿Por qué?. Verá que es un dispositivo por caracteres.

Pruebe con «/dev/cdrom» y «/dev/modem» (si tiene uno).

7.7.1 ¿Cómo creamos archivos especiales?

Los números mayores y menores están reservados, de manera que si queremos crear un dispositivo debemos referirnos a la documentación del núcleo de Linux (en este caso). En la documentación se establece que el número mayor, 42, es para uso de muestra (demo), para usar en código de muestra, como mero dispositivo de «ejemplo» y no debería usarse para controlador de dispositivo (*device driver*) que se está distribuyendo. Entonces:

```
# mknod /dev/prueba c 42 1
```

Crea un dispositivo por caracteres llamado «prueba» cuyo número mayor es 42 y cuyo número menor es 1. Para borrarlo, simplemente:

```
# rm /dev/prueba
```

y análogamente podemos crear uno por bloques

```
# mknod /dev/prueba b 42 1
```

En este punto, se recomienda repasar el capítulo de comunicación y sincronización de procesos. Recordará que podíamos comunicar procesos mediante tuberías (*pipes*) sin nombre o con nombre, entonces con

```
# mknod /dev/prueba p
```

creará la tubería (FIFO) «prueba», entonces si desde una consola colocamos el comando

```
# echo "Hola, mundo" > /dev/prueba
```

y desde otra colocamos el comando

```
# cat /dev/prueba
```

¿qué pasó?

En rigor de verdad no necesita ser superusuario para crear una FIFO pero necesita permiso de escritura en el directorio en el que la cree, en este caso «/dev». Aunque puede como usuario «ubuntu» crearla en su propio directorio

```
$ mknod otraprueba p
$ echo "Hola, otra vez" > otraprueba
```

y desde otra consola como usuario «ubuntu» estando en el mismo directorio:

```
$ cat otraprueba
```

7.8 Autoevaluación

1. Para una terminal o consola CTRL/D significa «fin de archivo»:
   a) Siempre.
   b) En modo crudo (raw).
   c) En modo cocido (cooked) y cbreak.
   d) Sólo en cbreak.
2. Indique cuál de las siguientes no es misión de un manejador (controlador o driver) de disco.
   a) Gestionar el espacio libre.
   b) Ofrecer una interfaz de manejo simple.
   c) Reintentar la operación en caso de error transitorio.
   d) Ser capaz de manejar varios dispositivos idénticos.
3. ¿Cuál de las siguientes afirmaciones es cierta?
   a) Los directorios son archivos especiales orientados al carácter.
   b) Un archivo especial orientado a bloque modela un dispositivo de acceso directo.
   c) En MS-DOS existen los archivos especiales.
   d) Un directorio es un archivo normal.
4. ¿Qué es cierto respecto a un manejador (controlador o driver) de un dispositivo de bloques?
   a) En un sistema monoproceso no necesita ser reentrante.
   b) Debe existir un manejador por cada dispositivo.
   c) Debe incluir una caché del dispositivo.
   d) Debe gestionar el espacio libre del dispositivo.
5. ¿Qué labor no realiza el driver de diskette en un sistema operativo?
   a) Copia el bloque traído por el controlador a la caché del sistema de archivos.
   b) Trata interrupciones de operación del controlador.
   c) Da orden de reinicio al controlador en caso necesario.
   d) Da orden de arrancar y parar el motor.

Multiprocesamiento

8

8.1 Múltiples procesadores

En los últimos años, hemos presenciado una migración progresiva desde los chips con un único procesador hasta los dispositivos de computación con múltiples núcleos de procesamiento, tanto en el mercado de los chips de propósito general como en el de los sistemas integrados (*embedded*). El desarrollo de los sistemas con múltiples procesadores ya es una actividad nuclear para las compañías más importantes de hardware. Se ha propuesto una gran diversidad de soluciones para sobrellevar los límites físicos de los dispositivos con un único núcleo y para hacer frente a la creciente demanda computacional de las aplicaciones modernas.

La Ley de Moore (1965) expresa que aproximadamente cada dos años se duplica el número de transistores en un circuito integrado. Se trata de una ley empírica, formulada por el cofundador de Intel, Gordon E. Moore, cuyo cumplimiento se ha podido constatar hasta hoy.

Tradicionalmente, hemos visto a las computadoras como máquinas secuenciales y con una única CPU para las que los programadores han concebido programas partiendo un problema en secuencias de instrucciones, que el procesador ejecuta secuencialmente siguiendo el modelo que vemos en la figura 8.1.

Además, cada instrucción de máquina en la CPU se ejecuta como una secuencia de operaciones a partir del modelo de la máquina de Von Neumann, es decir de la computadora de programa almacenado: el ciclo de obtención de las instrucciones, obtención de los operandos, realización de la operación y almacenamiento de los resultados o, como se lo conoce en inglés: *fetch instruction, fetch operands, perform operation, store result*, según apreciamos en la figura 8.2.

Según Stallings (2008), esta visión de las computadoras no ha sido nunca completamente cierta. A nivel de las microoperaciones se generan muchas señales de control al mismo tiem-

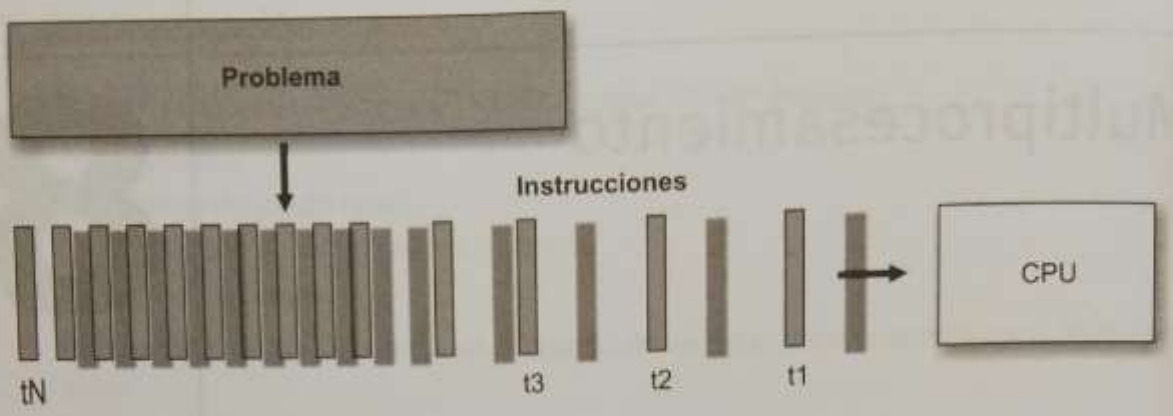


Fig. 8.1 – Modelo de procesamiento secuencial.

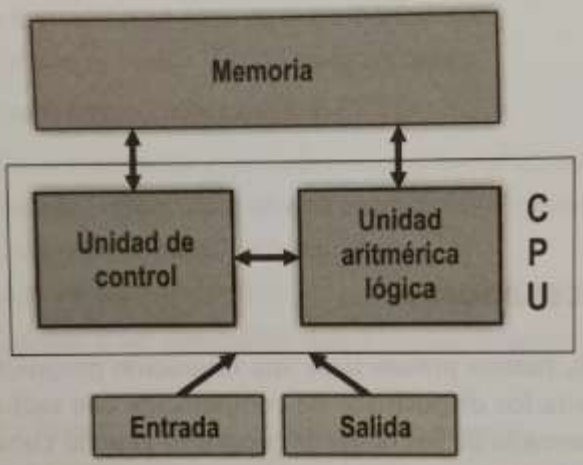


Fig. 8.2 – Arquitectura Von Neumann.

po. Además, conforme ha ido evolucionando la tecnología y ha ido disminuyendo el costo del hardware, los diseñadores de computadoras han buscado más y más ocasiones para el paralelismo, normalmente para incrementar el rendimiento, por ejemplo con los procesadores que siguen el paradigma de ejecución fuera de orden (o *out-of-order execution*, en inglés).

A decir verdad, no hay una frontera definida entre la arquitectura monoprocesador y las masivamente paralelas. De hecho, las actuales arquitecturas monoprocesador son realmente máquinas paralelas a nivel de instrucción. La evolución de la arquitectura basada en monoprocesadores ha venido ligada con la creación de más y mejores supercomputadores que tenían que liberarse del concepto de monoprocesador para poder hacer frente a las demandas de computación.

Con respecto a los microprocesadores, para incrementarles el rendimiento una solución obvia es aumentarles la velocidad mediante un reloj más rápido. Sin embargo, si bien permanentemente salen al mercado procesadores más rápidos, sabemos que no podemos seguir en esa línea indefinidamente porque hay razones de la Física que nos imponen restricciones.

8.2 Aplicaciones paralelas

Entonces, la otra solución que se nos ocurre es la de utilizar varios procesadores. Partiendo del principio de que un problema grande se puede dividir en partes más pequeñas que pueden procesarse de forma paralela en varios procesadores, nace como técnica de programación la computación paralela, cuyo modelo vemos en la figura 8.3.

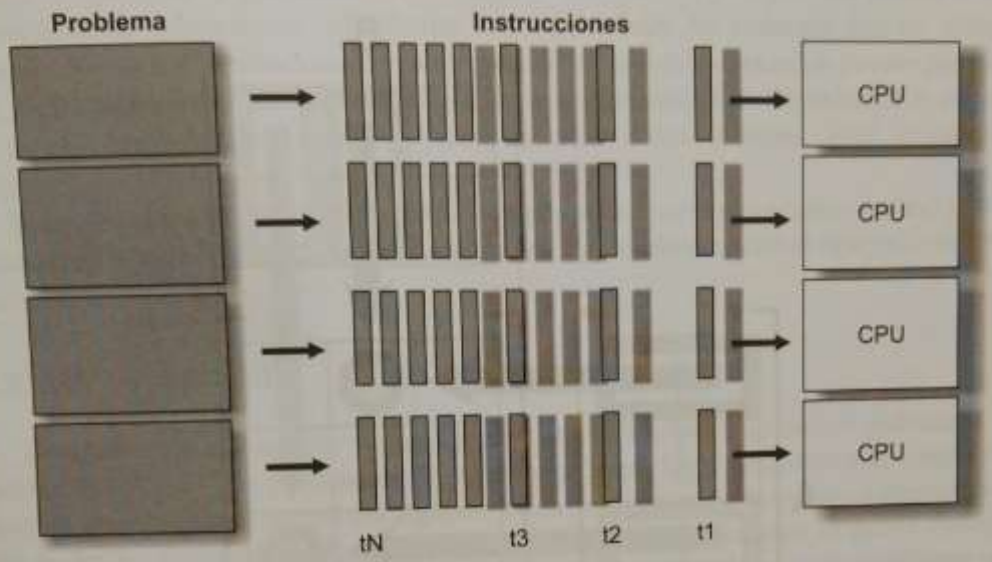


Fig. 8.3 – Modelo de procesamiento paralelo.

En este modelo, el problema se divide en varias partes y el programa se desarrolla de manera explícitamente paralela gracias a lenguajes y bibliotecas especializadas. Pongamos como ejemplo el producto de dos matrices.

Sabemos que dadas dos matrices A y B, tales que el número de columnas de la matriz A sea igual al número de filas de la matriz B, el producto de A por B será una nueva matriz C en la que cada elemento $c_{i,j}$ estará definido por

$$c_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj}$$

Como apreciamos en la figura 8.4, cada procesador puede ocuparse de obtener un elemento $c_{i,j}$ si le brindamos una fila de A y una columna de B. Esto lo codifica de manera explícita el programador.

Por ejemplo, si se parte la aplicación en múltiples procesos éstos se comunicarán utilizando formas de comunicación inter procesos (IPC) tales como tuberías (*pipes*), semáforos o memoria compartida. Los procesos deben ser capaces de bloquearse a la espera de eventos tales como mensajes de otros procesos, y deben coordinar el acceso a objetos compartidos con algo como cerrojos (*locks*).

Otra forma es utilizar los hilos POSIX, que tienen problemas de coordinación similares a los que tienen los procesos y mecanismos similares para enfrentarlos. Así, un proceso único

puede tener cualquier número de sus hilos ejecutando simultáneamente en diferentes procesadores. Coordinarlos y serializar el acceso a los datos compartidos es una responsabilidad del desarrollador.

También puede desarrollar un programa en Fortran o C utilizando la biblioteca MPI (de *Message Passing Interface*, en inglés) para programación paralela mediante pasaje de mensajes.

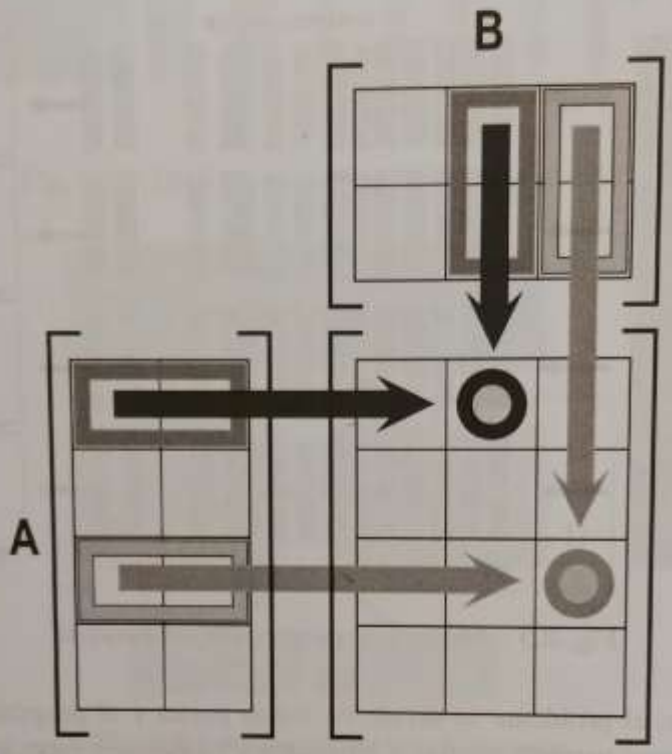


Fig. 8.4 – Diagrama de multiplicación de matrices.

En cualquier caso, el compilador cumple un rol fundamental, ya que es el responsable de la generación de hilos paralelos. Pero también necesita la ayuda del programador para identificar qué es lo que se quiere paralelizar. Por ejemplo, mediante la interfaz de programación de aplicaciones (API, *Application Programming Interface*) de OpenMP[1] para la programación multiproceso de memoria compartida, podemos añadir concurrencia a los programas escritos en los lenguajes C, C++ o Fortran. OpenMP se compone de un conjunto de directivas de compilador, rutinas de biblioteca, y variables de entorno que influyen el comportamiento en tiempo de ejecución.

No obstante, el anterior es un tipo especial de problema. No todos los problemas pueden paralelizarse; en términos de tareas humanas, el agregar gente puede ser una buena idea si la tarea es responder llamados en un centro de atención al cliente (*call center*), pero no lo es si se trata de manejar un auto.

Tampoco pueden programarse todos los problemas en este modelo, y aún cuando es factible hacerlo, la paralelización también tiene un límite.

1. `http://www.openmp.org/`

Muchas aplicaciones –por no decir la mayoría– que utilizamos no están diseñadas de esta manera, sino que están implementadas mediante hilos que el programador supuso que se ejecutarían en un sistema con un único procesador.[2] El núcleo del sistema operativo ejecutando en un sistema con múltiples procesadores toma fragmentos de hilos de distintas aplicaciones, susceptibles de ser asignados a un procesador y los ejecuta de manera asíncrona. La migración a un multiprocesador puede mejorar el rendimiento global de un sistema y el tiempo de ejecución de aplicaciones complejas y multihiladas pero también mejora el tiempo de respuesta de aplicaciones individuales de un único hilo. Sin embargo, esto no siempre es así dado que las aplicaciones se siguen ejecutando secuencialmente en un pseudo paralelismo que, si no está bien planteado, puede degradar su rendimiento. Esta degradación es provocada por la sobrecarga creada cuando el sistema operativo conmuta tareas, como ya vimos en el capítulo 2.

Todo esto provoca, que el software del sistema operativo se torne más complejo y, por otra –lamentablemente– el rendimiento tampoco escala linealmente con el agregado de procesadores, como veremos.

8.3 Ley de Amdahl

Decíamos que era posible suponer que al utilizar varios procesadores podríamos incrementar la velocidad de procesamiento de manera lineal. En el escenario más simple supondremos que el sistema se está encargando de un solo proceso o hilo. Supondremos también que podemos modificar el algoritmo de manera que cada procesador se pueda encargar de ejecutar su propia porción de código de manera paralela a los otros y sin interferirse mutuamente. Podríamos expresar esta mejora (*speedup*, en inglés) como la tasa:

$$S_P = \frac{T_1}{T_P}$$

Es decir, siendo P el número de procesadores, T_1 el tiempo de ejecución con el algoritmo secuencial con un procesador y T_P el tiempo de ejecución con el algoritmo paralelo con P procesadores, el *speedup* S de P procesadores será el cociente entre ambos. El *speedup* lineal o ideal se obtiene cuando $S_P = P$. De manera que al ejecutar un algoritmo con *speedup* lineal, si duplicamos el número de procesadores, duplicamos la velocidad.

Al incrementar el número de procesadores, esperamos realizar más trabajo en un tiempo más corto. Sin embargo, la proporción de aumento de la velocidad con P procesadores no es P, sino más bien menor que P. Cuando varios procesadores cooperan para llevar a cabo una tarea, se incurre en cierto «gasto extra» para mantener todos los componentes funcionando correctamente. Este gasto extra, aunado a la contención por los recursos compartidos, reduce la ganancia que cabría esperar de los procesadores adicionales.

Amdahl (1967), en la ley que lleva su nombre, demostró que existe un máximo en la mejora esperada en el sistema cuando hay una mejora en una porción de éste. Esta ley se emplea para calcular la máxima mejora teórica en el rendimiento al utilizar múltiples procesadores.

2. Además, es una buena estrategia para que la aplicación pueda ejecutar tanto en un sistema con procesador único como en uno con múltiples procesadores.

En el caso especial de la paralelización, la Ley de Amdahl declara que si *F* es la fracción de un cálculo secuencial y *1-F* la fracción que puede ser paralelizada, entonces la máxima aceleración (o *speedup*) que se puede lograr utilizando *P* procesadores es:

$$\frac{1}{(F+(1-F)/P)}$$

En el límite, con *P* tendiendo a infinito, la máxima aceleración tiende a *1/F*, como podemos observar en la figura 8.5.

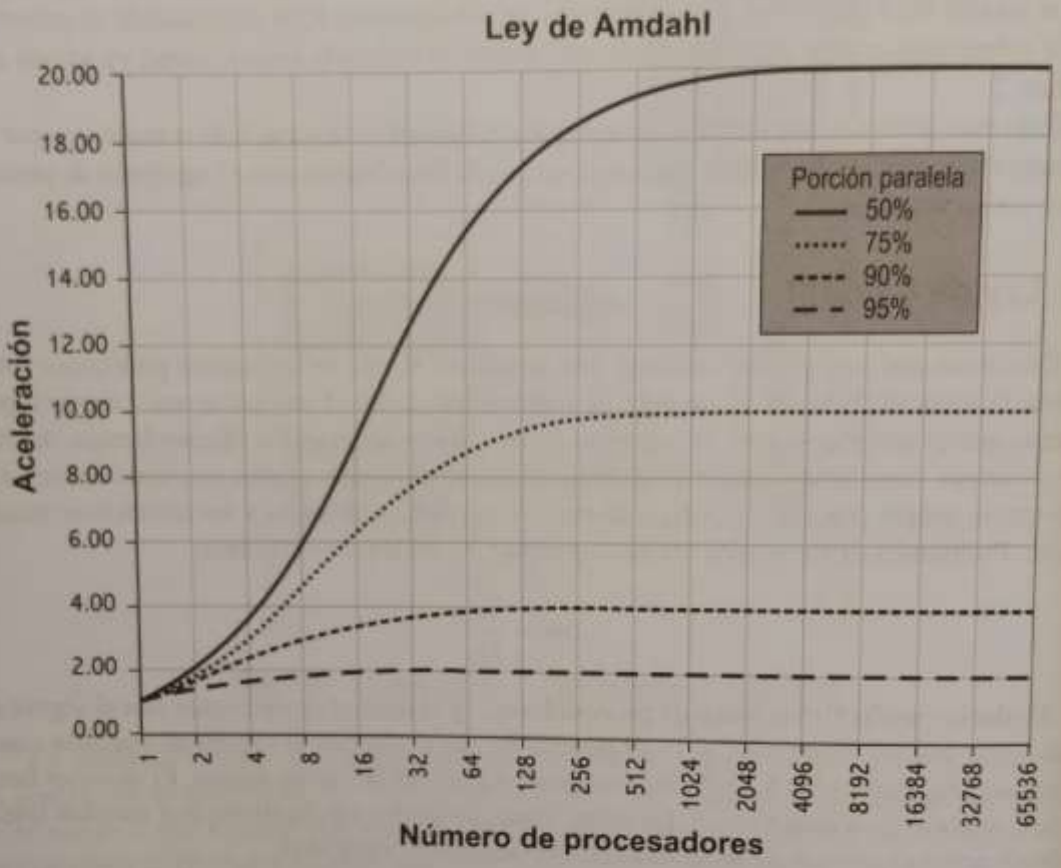


Fig. 8.5 – Ley de Amdahl.

El tratamiento completo de este tema está más allá del alcance de este libro porque implica tratar cómo las aplicaciones paralelas –mediante el uso de bibliotecas específicas– aprovechan las capacidades de un sistema con múltiples procesadores.

8.4 Taxonomía de Flynn

La forma más común que hemos tenido de clasificar a los sistemas de cómputo ha sido la taxonomía propuesta por Flynn (1972), quien eligió dos características consideradas por él como esenciales: el número de flujos de instrucciones y el número de flujos de datos, como vemos en la tabla 8.1.

	Single instruction	Multiple instructions
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Tabla 8.1 – Taxonomía de Flynn.

Una computadora con un flujo de instrucciones y uno de datos se llama SISD (*Single Instruction, Single Data*). Se trata de una computadora netamente serial, explícitamente no paralela. La CPU actúa sobre un único flujo de instrucciones durante un ciclo de reloj y tiene un único flujo de datos utilizado como entrada durante ese ciclo de reloj; por lo tanto, su ejecución es determinística, como vemos en la figura 8.6. Este es el concepto de arquitectura serie de Von Neumann donde, en cualquier momento, sólo se está ejecutando una única instrucción. Todas las máquinas SISD poseen un registro simple –el contador de programa– que asegura la ejecución en serie del programa. Conforme se van leyendo las instrucciones de memoria, el contador de programa se actualiza para que apunte a la siguiente instrucción que se va a procesar en serie. Ningún computador puramente SISD se fabrica hoy en día.

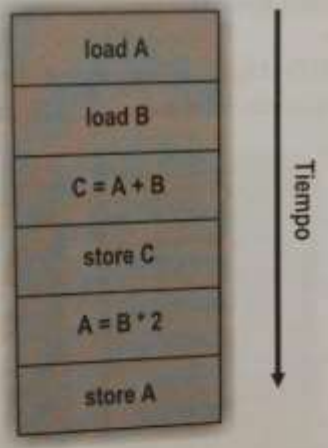


Fig. 8.6 – SISD.

Todas las computadoras personales tradicionales de un solo procesador caían dentro de esta categoría (pero no las actuales), así como ciertos viejos y grandes mainframes.

La siguiente categoría es SIMD (*Single Instruction, Multiple Data*), en la que un único flujo de instrucciones opera sobre múltiples flujos de datos, como vemos en la figura 8.7.

Estas máquinas son útiles para los cómputos que repiten los mismos cálculos en varios conjuntos de datos: por ejemplo, sumando todos los elementos de 64 vectores independientes; es decir que tenemos aquí un paralelismo de datos (o *data parallelism*, en inglés). Un procesador vectorial cae en esta categoría, como también una Unidad de Procesamiento Gráfico (GPU, de *Graphics Processing Unit*, por sus siglas en inglés). Otro ejemplo es el cálculo

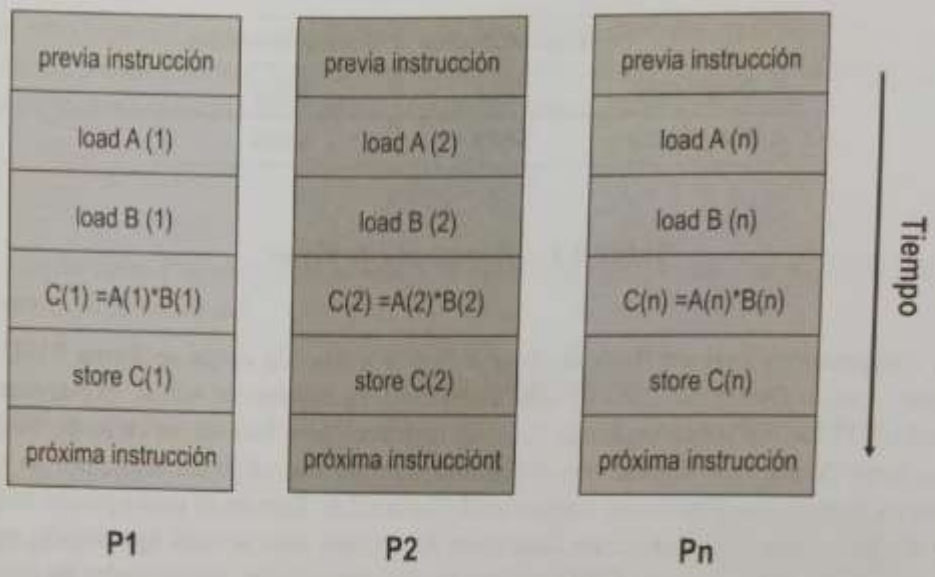


Fig. 8.7 – SIMD.

de la paga para cada trabajador en un empresa, que consiste en repetir la misma operación sencilla para cada trabajador; si se dispone de una arquitectura SIMD, esto se puede calcular en paralelo para cada trabajador.

La siguiente categoría es MISD (*Multiple Instruction, Single Data*), con un flujo de varias instrucciones y un flujo de datos, como se observa en la figura 8.8.

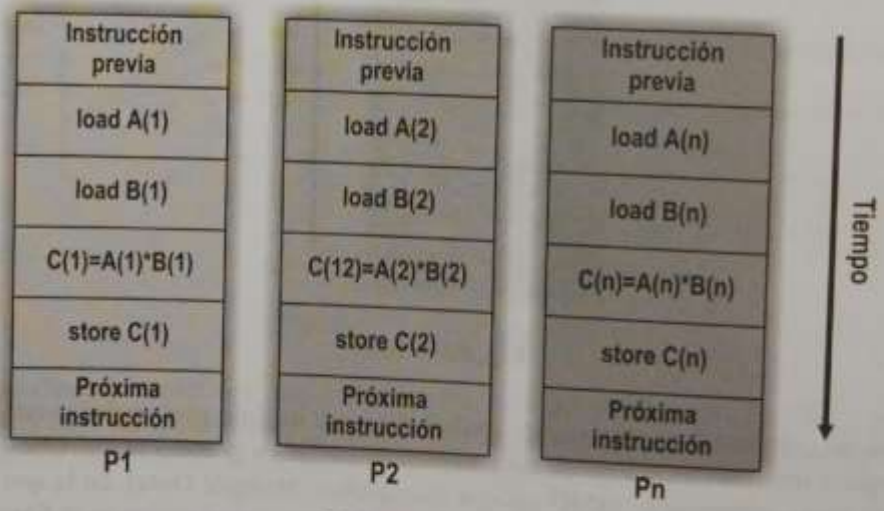


Fig. 8.8 – MISD.

Ninguna de las computadoras comúnmente conocidas se ajusta a este modelo ya que implica que varios procesadores operen sobre los mismos datos – y acuerden en los resultados. Sería aplicable en caso de ser necesario un sistema tolerante a fallos, como por ejemplo el que utiliza la computadora de control de vuelo del transbordador espacial.

Por último, está MIMD (*Multiple Instruction, Multiple Data*), con múltiples flujos de instrucciones y flujo de datos múltiple, que se refiere a un grupo de procesadores independientes, cada uno con su propio contador de programa y datos.

Es decir que, si tenemos procesadores separados operando sobre datos independientes, tendremos un paralelismo de control (o *control parallelism*, en inglés), como se observa en la figura 8.9.

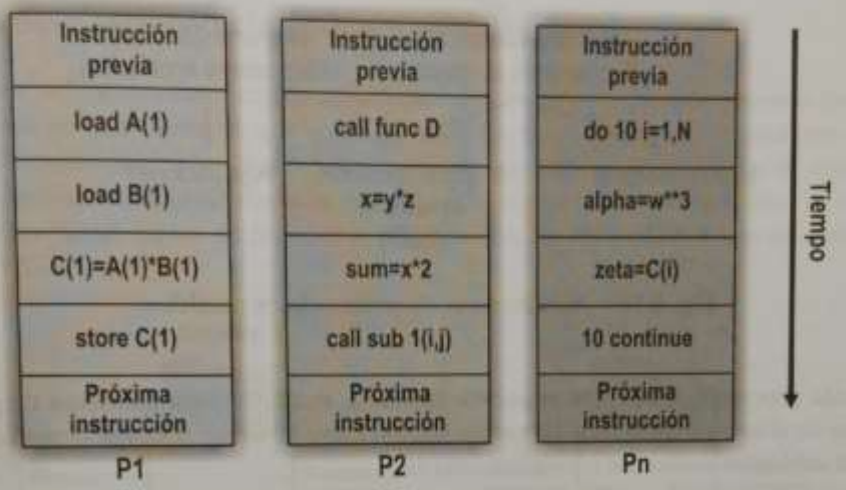


Fig. 8.9 – MIMD.

Las MIMD son las más complejas, pero son también las que potencialmente ofrecen una mayor eficiencia en la ejecución concurrente o paralela.

Todos los sistemas distribuidos son MIMD, por lo que este sistema de clasificación no es muy útil para nuestros fines si no lo explicitamos un poco más. Estos sistemas pueden a su vez ser subdivididos en multiprocesadores, multicomputadoras, multimultiprocesadores y máquinas de flujo de datos.

El multiprocesamiento se originó a mediados de la década de 1950 pero según M. T. Jones (2007b), la primera arquitectura SMP fue desarrollada por Burroughs Corporation en 1962. Se trató del multiprocesador MIMD simétrico D825, que contaba con hasta cuatro CPU que accedían a uno o dieciséis módulos de memoria conectados mediante un conmutador de barras cruzadas (*crossbar switch*, en inglés).

En la organización MIMD, los procesadores son de propósito general porque deben ser capaces de procesar todo tipo de instrucciones para llevar a cabo todas las transformaciones de datos necesarias.

8.5 Arquitecturas de múltiples procesadores

Según Stallings (2008), la clasificación MIMD puede, a su vez, subdividirse de acuerdo con el medio por el cual se comunican, según se aprecia en la figura 8.10.

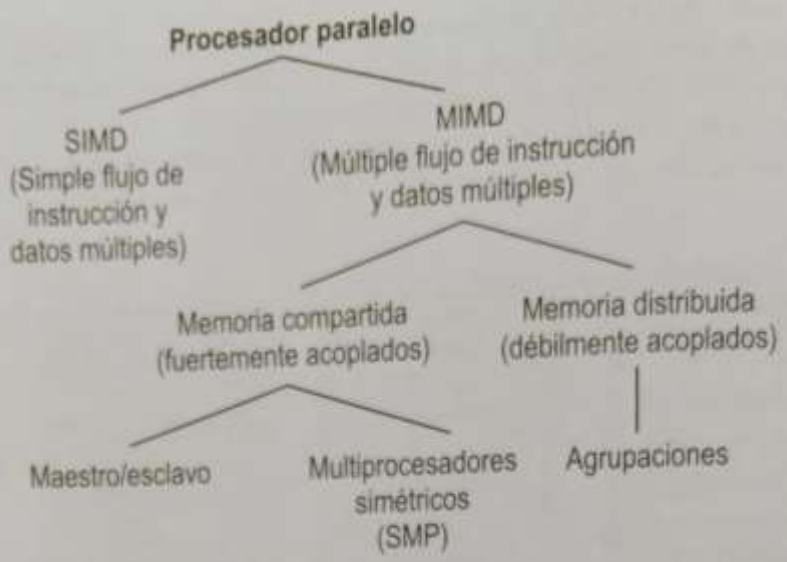


Fig. 8.10 – Arquitecturas de procesadores paralelos.

Si cada procesador tiene una memoria dedicada, entonces cada elemento de procesamiento es en sí una computadora. La comunicación entre las computadoras se realiza ya sea mediante cableados dedicados o mediante una red.

Si la comunicación se realiza mediante una red, se dice que los elementos están «débilmente acoplados» (*loosely coupled*, en inglés). Se los denomina también **multicomputadora** porque sus elementos suelen estar distantes entre sí al menos un par de metros. También se los suele denominar **agrupación** (*cluster*, en inglés), de los cuales el proyecto Linux Beowulf es un buen ejemplo. Podemos apreciar un modelo de arquitectura de agrupación en la figura 8.11.

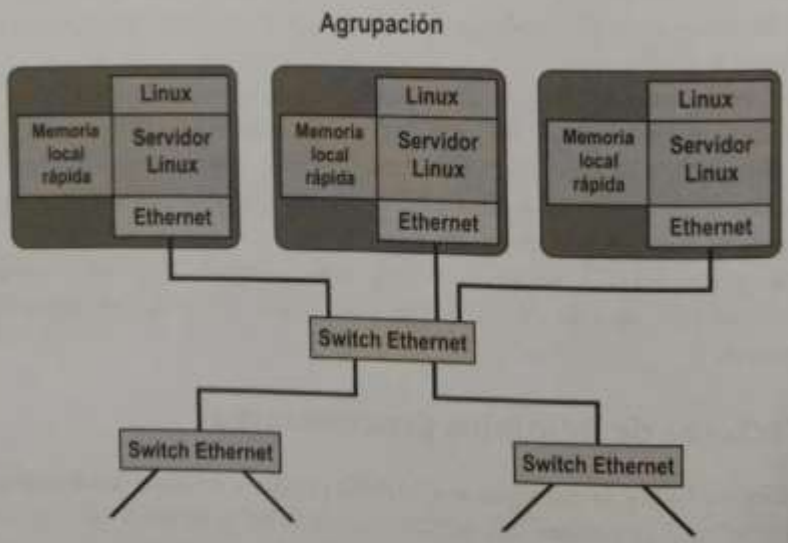


Fig. 8.11 – Arquitectura de una agrupación.

En cambio, si los procesadores comparten una memoria en común y se comunican entre sí a través de esta memoria, tendremos entonces un sistema **multiprocesador** de memoria compartida. Diremos que estos sistemas están «fuertemente acoplados» (*tightly coupled*, en inglés) porque sus elementos suelen estar distantes entre sí –a lo sumo– unos centímetros.

Estas dos arquitecturas no son excluyentes unas con otras; es decir, se suelen construir agrupaciones constituidas por sistemas multiprocesadores de memoria compartida.

8.5.1 Procesamiento simétrico y asimétrico

Existe a su vez otra clasificación general de los multiprocesadores de memoria compartida que se basa en la manera en que son asignados los procesos a los procesadores. Los dos enfoques fundamentales son los del procesamiento asimétrico (AMP, siglas de *Asymmetric Multiprocessing*, en inglés) también llamado maestro/esclavo[3] (*master/slave*, en inglés) y el simétrico (SMP, siglas de *Symmetric Multiprocessing*, en inglés), como se ilustra en la figura 8.12.

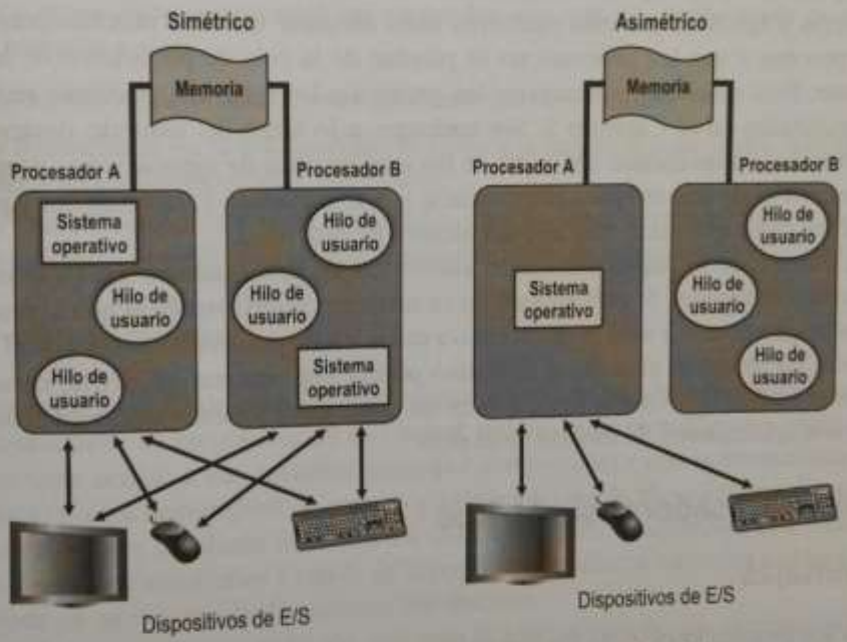


Fig. 8.12 – Multiprocesamiento simétrico y asimétrico.

En la arquitectura maestro/esclavo el núcleo del sistema operativo siempre ejecuta en un procesador mientras que en el otro sólo se ejecutan programas de usuario y tal vez utilitarios del sistema operativo. El maestro o amo es el responsable de la planificación de procesos e hilos en el esclavo, y cada vez que estos procesos e hilos solicitan una llamada a sistema deben enviar un pedido al maestro y esperar que finalice el servicio.

Este enfoque es simple y requiere de pocas modificaciones desde el punto de vista del sistema operativo. No obstante, como desventaja, podemos señalar que un fallo en el maestro

3. Es la denominación más corriente en el mundo hispano, pero se la debería llamar «amo/esclavo»

provocará un colapso en todo el sistema. Además, para el maestro puede representar todo un cuello de botella en el desempeño ya que él sólo se debe encargar de toda la administración y planificación de procesos.

Estas fueron las desventajas enfrentadas por los desarrolladores de los primeros sistemas que tenían más de un procesador, por lo que avanzaron al próximo paso.

En un multiprocesador simétrico (SMP) el núcleo puede ejecutar en cualquier procesador –como decíamos en el apartado 1.2.6– y usualmente cada procesador hace su propia planificación a partir de un grupo de procesos o hilos disponibles. Al núcleo se lo puede construir como varios procesos o hilos, lo que permite que porciones del núcleo puedan ejecutar en paralelo. El término «simétrico» en un sistema SMP indica que cualquier unidad de software debería ser capaz de ejecutar en cualquier procesador: no hay un procesador «maestro».

Como con cualquier cambio que incremente la complejidad del sistema, el uso de múltiples procesadores genera consideraciones de diseño que deben ser abordadas para lograr una operación y desempeño satisfactorio.

El enfoque SMP complica el diseño del sistema operativo; no obstante, muchos sistemas operativos profesionales disponibles en el mercado dan soporte a SMP, tales como Windows, Linux, Solaris y QNX. El sistema operativo debe asegurar que dos procesadores no elijan al mismo proceso y que los procesos no se pierdan de la cola de procesos o de hilos listos para ejecutar. Para resolver y sincronizar los pedidos a los recursos, podemos recurrir a las técnicas analizadas en el Capítulo 5. Sin embargo, a lo largo del capítulo desarrollaremos más aspectos de la arquitectura SMP que de las arquitecturas de agrupaciones porque, como dijimos, necesitan de una red para comunicarse y esto introduce características y problemas propios de las redes que están más allá del alcance de este libro.

En un sistema de multiprocesamiento asimétrico los procesadores son (o al menos son tratados como) diferentes. Si el hardware no es simétrico podría implicar que algunos procesadores no tengan acceso a toda la memoria o a todos los dispositivos externos. Si el hardware es simétrico, no obstante, el sistema operativo podría implementar asimetría, como se dijo, ejecutando el núcleo del sistema operativo en un procesador y tareas adicionales del sistema operativo con aplicaciones de usuario en el otro.

8.6 Multiprocesadores simétricos

8.6.1 Ventajas

Todos los procesadores están fuertemente acoplados dentro de la misma caja con un colector (o *bus*) de alta velocidad o un conmutador. Los procesadores comparten la misma memoria global, discos y dispositivos de entrada y salida. Sólo una copia del sistema operativo corre en todos los procesadores y el sistema operativo debe estar diseñado para explotar esta arquitectura (sistema operativo multihilado).

Entre otras ventajas, encontramos:

- Son una manera rentable de aumentar el rendimiento.
- Ofrecen una imagen única de sistema ya que el sistema operativo está compartido entre todos los procesadores (la administración es fácil).
- Aplican varios procesadores a un único problema (en paralelo).

- El sistema operativo hace el balanceo de cargas.
- El modelo de programación del uniprocesador se puede usar en un sistema multiprocesador de memoria compartida.
- Son escalables para datos compartidos.
- Todos los datos son direccionables por todos los procesadores y se mantienen coherentes por la lógica de *snooping* del hardware: Es un protocolo que está continuamente sondeando o fisgando (*snooping*, en inglés) el colector común para ver qué transacciones podrían introducir incoherencia y actúa en consecuencia.
- No hay necesidad de usar bibliotecas de pase de mensajes para la comunicación entre procesadores porque se hace a través de la memoria compartida global.
- Se pueden solucionar más requisitos de potencia (de cálculo) mediante la adición de más procesadores en el sistema. Sin embargo, se deben establecer expectativas realistas sobre el aumento en el rendimiento al agregar más procesadores a un sistema de este tipo.
- Hoy en día hay disponibles más y más aplicaciones y herramientas. La mayoría de las aplicaciones para uniprocesadores se pueden ejecutar en arquitecturas de este tipo o son portadas a estas.

Asimismo, hay algunas limitaciones para este tipo de sistemas:

- Hay limitaciones en la escalabilidad debidas a la coherencia de caché, mecanismos de trabas y objetos compartidos, entre otros.
- Hay una necesidad de nuevas habilidades para explotar a los multiprocesadores, tales como la programación con hilos y la programación de manejadores de dispositivos.

8.6.2 Organizaciones

Las figuras 1.2 y 8.13 ilustran la organización general de un SMP. Hay varios procesadores y cada uno tiene su propia unidad de control, unidad aritmético-lógica y registros. Cada procesador tiene acceso a una memoria principal compartida (y con la misma latencia en el acceso a esta) y a los dispositivos de entrada y salida a través de alguna forma de mecanismo de interconexión, por lo común mediante un colector (o *bus*) compartido. Los procesadores se pueden comunicar entre ellos a través de la memoria, mediante mensajes e información de estado dejada en un espacio de direcciones compartido.

Si imaginamos que hemos reducido casi toda la arquitectura de una agrupación (como la ilustrada en la figura 8.11) hasta hacerla caber en un *chip*, obtendremos un CMP (de *Chip level MultiProcessing*) con el que podremos realizar computación multinúcleo (*multi-core computing*, en inglés). En la figura 8.14 vemos el diagrama de un procesador multinúcleo.

En cambio, en un sistema CMP tenemos varias CPU conectadas a una memoria compartida (caché de nivel 2) a través de un colector compartido. Cada procesador tiene también su propia memoria, que es rápida, por tratarse de una caché de nivel 1. La naturaleza fuertemente acoplada del CMP permite distancias físicas muy cortas entre los procesadores y la memoria y, por ende, latencias mínimas de acceso a la memoria y mejor desempeño. Este tipo de arquitectura funciona bien en aplicaciones con múltiples hilos en las que estos hilos se puedan distribuir entre todos los procesadores para operar en paralelo. A esta forma de

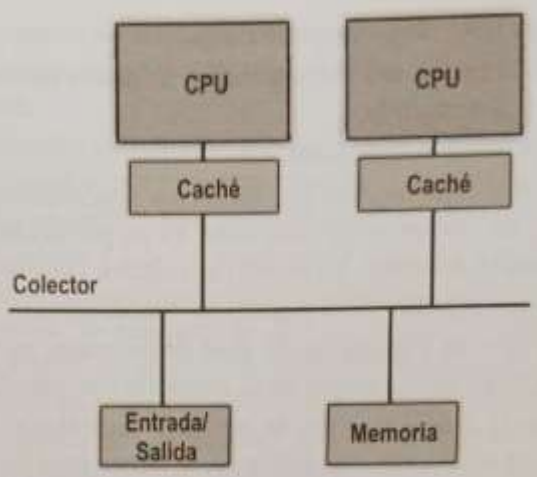


Fig. 8.13 – Jerarquía SMP.

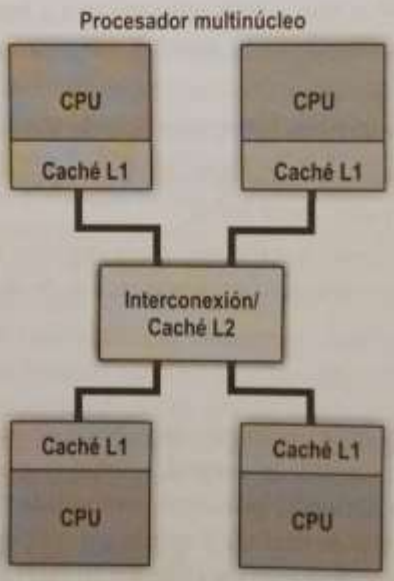


Fig. 8.14 – Procesador multi núcleo.

operar se la conoce como paralelismo a nivel de hilo (TLP, de *Thread-Level Parallelism*, por sus siglas en inglés).

Y, como como decíamos en el apartado 1.2.6, existen los sistemas de múltiples hilos simétricos (SMT, de *Symmetric Multi-Threading*, por sus siglas en inglés); por ejemplo, los procesadores con tecnología *Hyper-Threading* (HT) de Intel. Se caracterizan porque cada *chip* SMT físico puede tener más de un procesador virtual, con la salvedad de que los procesadores virtuales comparten ciertos recursos (algunos tipos de caché). Dado que se comparten ciertos recursos, los procesadores virtuales no deberían ser tratados de la misma forma que los procesadores corrientes.

8.6.3 Afinidad

En un sistema SMP, el planificador del sistema operativo no sólo debe decidir cuándo debe ejecutar un hilo o proceso, sino también dónde debe ejecutarlo, en cuál de los procesadores del sistema. La afinidad del procesador (*processor affinity*, en inglés) es el término utilizado para describir las reglas para asociar ciertos hilos a ciertos procesadores y se refiere a:

> La probabilidad de activación de un hilo al procesador en el cual estaba ejecutando previamente. Esta debería variar directamente con el tamaño del conjunto de trabajo de la caché del hilo e inversamente con la duración del tiempo desde que fue activado anteriormente.

Es importante no sólo distribuir equitativamente la carga entre los procesadores sino también mantener cierta afinidad para usar eficientemente las caché. Observando la figura 8.14 podemos notar que cuando una tarea está asociada con una determinada CPU, moverla hacia otra provocará fallos de caché (*cache miss*), vaciando la caché previa y llenando la actual hasta tener todo el conjunto de trabajo. Esto incrementa la latencia de acceso a la memoria hasta que los datos estén en la caché de la otra CPU; según advierte M. T. Jones (2007b). El mayor grado de afinidad se logrará cuando se ligue por completo un hilo a una CPU, lo que significa que sólo se lo asignará a ese procesador sin importar que haya otros disponibles. Esto puede ser útil tratándose de hilos orientados a la CPU, pero es contraproducente cuando el hilo solicita una entrada o salida, porque si al finalizar esta el procesador en el que ejecutaba se encuentra ahora ocupado, el hilo deberá esperar hasta que se desocupe, sin poder ejecutar en otro procesador que pueda estar disponible en ese instante.

8.6.4 Granularidad

La palabra «granularidad» no existe en español pero la palabra *granularity* en inglés, referida a un sistema, describe el nivel de detalle o el número de componentes distintos que este contiene: mientras más componentes tenga o mayor granularidad tenga, diremos que su grano será más fino, y en consecuencia le dará al sistema mayor flexibilidad.

Por «componente» entenderemos a un pequeño programa u objeto binario que realiza una función específica y está diseñado de forma tal que pueda operar fácilmente con otros componentes y aplicaciones.

En computación paralela, granularidad es una medida cualitativa que se refiere a la cantidad de cálculo en relación a la comunicación.

Según Stallings (2008), una buena forma de caracterizar a los multiprocesadores y ponerlos en contexto con otras arquitecturas es considerar la granularidad de la sincronización o frecuencia de sincronización entre procesos en un sistema. Podemos distinguir cinco categorías de paralelismo que difieren en el grado de granularidad, las cuales se resumen a continuación y están adaptadas de las propuestas por Gehringer, Siewiorek y Segall (1986) y Woodbury y col. (1989), ya que los tamaños pueden ser discutibles (véase Tabla 8.2).

Tamaño de grano	Descripción	Intervalo de sincronización
Fino	Paralelismo inherente en un único flujo de instrucciones	<20
Medio	Procesamiento paralelo o multitarea dentro de una aplicación individual	20-200
Grueso	Multiprocesamiento de procesos concurrentes en un entorno multiprogramado	200-2000
Muy grueso	Proceso distribuido por los nodos de una red para formar un solo entorno de computación	2000-1M
Independiente	Varios procesos no relacionados	(N/A)

Tabla 8.2 – Procesos y granularidad de la sincronización.

8.6.5 Paralelismo

8.6.5.1 Independiente

En el paralelismo independiente no hay sincronización explícita entre procesos. Cada uno representa una aplicación o tarea separada e independiente. Un uso típico de este tipo de paralelismo se da en un sistema de tiempo compartido, en el que cada usuario está ejecutando una determinada aplicación en particular tal como un procesador de textos o una planilla de cálculos. El multiprocesador provee el mismo servicio que un uniprocesador multiprogramado. Debido a que hay disponible más de un procesador, el tiempo de respuesta promedio a los usuarios será menor.

Es posible alcanzar una ganancia similar en el rendimiento proveyendo a cada usuario una computadora personal. Si debieran ser compartidos los archivos o información entonces los sistemas individuales deberían conectarse a un sistema distribuido apoyado por una red. Por otra parte, un único sistema multiprocesador compartido en muchas casos es más rentable que un sistema distribuido, lo que permite economía de escala en discos y otros periféricos.

Cercano a este concepto está el de las aplicaciones «vergonzosamente paralelas» (o *embarrassingly parallel*, en inglés) que resuelven varias tareas similares, pero independientes, de forma simultánea y en las que hay poca o ninguna necesidad de coordinación entre ellas.

8.6.5.2 De grano grueso y muy grueso

Con este tipo de paralelismo hay sincronización entre procesos pero poca. El intervalo de sincronización entre instrucciones según los autores mencionados, estaría entre 200 y 2000. Este tipo de situación es manejada fácilmente como un conjunto de procesos concurrentes que ejecutan sobre un uniprocesador multiprogramado y puede ser implementado sobre un multiprocesador con poco o ningún cambio al software. En general, cualquier conjunto de procesos concurrentes que necesiten comunicarse o sincronizarse pueden beneficiarse del uso de una arquitectura multiprocesador.

Un sistema distribuido puede brindar un buen apoyo en el caso de interacciones muy poco frecuentes entre procesos, digamos con intervalos de sincronización entre 2000 instrucciones o más. No obstante, si la interacción es en cierta manera más frecuente, entonces la sobrecarga de las comunicaciones a través de la red puede invalidar algunas de las mejoras potenciales. En ese caso, la organización de multiprocesadores provee el respaldo más efectivo.

8.6.5.3 De grano medio

Vimos que una aplicación individual puede ser implementada efectivamente como un conjunto de hilos dentro de un único proceso. En este caso, el programador debe especificar en forma explícita el paralelismo potencial de una aplicación. Típicamente, se necesitará tener más bien un alto grado de coordinación e interacción entre los hilos de una aplicación que conduzca a un nivel de grano medio de sincronización. El intervalo de sincronización entre instrucciones puede estar entre 20 y 200.

Visto que tanto el paralelismo independiente como el de grano grueso y muy grueso pueden funcionar tanto en un uniprocesador multiprogramado como en un multiprocesador con bajo o nulo impacto en la planificación, necesitamos reexaminar la planificación cuando se trata de hilos, debido a que los diferentes hilos de una aplicación interactúan tan frecuentemente que las decisiones de planificación acerca de un hilo pueden afectar el rendimiento de toda la aplicación.

8.6.5.4 De grano fino

El paralelismo de grano fino representa un uso del paralelismo mucho más complejo que el encontrado en el uso de los hilos. A pesar de que se ha trabajado mucho en las aplicaciones altamente paralelas, esto es hasta el momento para un área muy especializada y con muchos enfoques diferentes. Este tipo de aplicaciones se sincronizarían cada veinte instrucciones o menos.

8.6.6 Serialización de datos

Cualquier elemento almacenado que puede ser leído o escrito por más de un hilo puede cambiar mientras el programa está ejecutando.

Esto es generalmente cierto tanto respecto a entornos con multiprogramación como a entornos multiproceso, pero la llegada de multiprocesadores agravan el alcance e importancia de esta consideración de dos maneras:

- Los multiprocesadores y el soporte de hilos hacen atractivo y más fácil escribir aplicaciones que comparten datos entre hilos.
- El núcleo ya no puede resolver el problema de la serialización simplemente mediante la desactivación de las interrupciones.

Para evitar problemas serios, los programas que comparten datos deben organizarse para acceder a esos datos serialmente, no en paralelo. Antes de que un programa actualice un dato compartido se debe asegurar de que ningún otro programa (incluyendo otra copia de sí

mismo que esté ejecutando en otro hilo) lo cambiará. Las lecturas usualmente pueden hacerse en paralelo.

El cerrojo es el principal mecanismo que se usa para evitar que los programas se interfieran unos con otros. Es una abstracción que representa el permiso para acceder uno o más ítems de datos. Las solicitudes para trabar y destrabar son atómicas; esto es, están implementadas en una forma tal que ni las interrupciones ni los accesos del multiprocesador afectan el resultado. Todos los programas que acceden a un ítem de dato compartido deben obtener el cerrojo que corresponde a ese ítem de datos antes de manipularlo. Si otro programa ya tiene el cerrojo (u otro hilo que esté ejecutando el mismo programa), el programa solicitante debe diferir su acceso hasta que esté disponible el cerrojo.

Además del tiempo transcurrido esperando al cerrojo, la serialización suma al número de veces que un hilo se vuelve no activable. Mientras que el hilo no es activable, otros hilos pueden probablemente estar provocando que sean reemplazadas las líneas de la caché del hilo no activable, lo que se traduce en un aumento de los costos de latencia a memoria cuando el hilo finalmente consigue el cerrojo y es activado.

El núcleo del sistema operativo contiene muchos ítems de datos compartidos, de manera que debe realizar serialización internamente. Pueden ocurrir retardos de serialización aun en un programa que no comparte datos con otros programas porque los servicios del núcleo usados por el programa tienen que serializar datos compartidos del núcleo.

8.7 Diseño

Un sistema SMP debe encargarse no sólo de la administración de los procesadores, sino también de los otros recursos de computación de manera tal que el usuario del mismo pueda percibirlo de la misma manera que a un sistema monoprocesador. En este sentido, un programador debería poder desarrollar sus aplicaciones sin preocuparse de que sus hilos ejecuten en uno o varios procesadores. Por esto, un sistema operativo multiprocesador debe proveer no sólo todas las funcionalidades de un sistema multiprogramado, sino también aspectos adicionales para poder dar cabida a varios procesadores. Según Stallings (2008), hay varias consideraciones acerca del diseño de un sistema operativo multiprocesador que debemos tener en cuenta:

- Procesos o hilos simultáneos y concurrentes: Las rutinas del núcleo necesitan ser reentrantes para permitir que varios procesadores ejecuten el mismo código del núcleo de manera simultánea. Para prevenir bloqueos mutuos o estancamientos (o *deadlocks*, en inglés) u operaciones inválidas se deben manipular adecuadamente las estructuras de datos del núcleo, si tenemos a varios procesadores ejecutando la misma o diferentes partes del núcleo.
- Planificación: Dado que puede ser hecha por cualquier procesador, se deben evitar los conflictos. Si se utilizan varios hilos en el núcleo, entonces existe la oportunidad de planificar varios hilos del mismo proceso simultáneamente en distintos procesadores.
- Sincronización: Se debe tener cuidado de proveer una efectiva sincronización porque tenemos varios procesos activos que están teniendo acceso potencial a espacios de direcciones compartidas o a recursos de entrada y salida compartidos. La sincronización es una solución que exige exclusión mutua y ordenamiento de eventos. Las trabas (o

locks, en inglés) son un mecanismo de sincronización usado comúnmente en sistemas operativos multiprocesadores.

- Gestión de la memoria: La gestión de la memoria en un multiprocesador debe tratar con todos los problemas encontrados en las computadoras con un procesador. Ademas de esto, el sistema operativo necesita sacar provecho del paralelismo del hardware disponible tal como las memorias *multiported* para lograr el mejor rendimiento. Los mecanismos de paginación en procesadores diferentes deben estar coordinados para imponer consistencia cuando varios procesadores comparten una página o segmento y para decidir el reemplazo de una página. Si consideramos la memoria como un recurso compartido, los sistemas de multiprocesamiento pueden hacer que el problema sea peor. Ahora el sistema debe bloquear varios procesadores a la vez, porque solo un procesador puede acceder a la memoria a la vez.
- Confiabilidad y tolerancia a fallos: El sistema operativo debería proveer degradación gradual frente al fallo de un procesador. El planificador y otras porciones del sistema operativo deben reconocer la pérdida de un procesador y reestructurar como corresponde las tablas de gestión.

8.7.1 Procesos o hilos simultáneos y concurrentes

El diseño del núcleo del sistema operativo debe permitir que las rutinas del núcleo sean reentrantes para que varios procesadores puedan ejecutar el mismo código de núcleo de manera simultánea. Esta característica les permitirá a las aplicaciones que hayan sido desarrolladas implementando varios hilos de ejecución que puedan ejecutarse sobre más de un procesador. Las aplicaciones que no hayan sido desarrolladas con vistas a ejecución en un SMP no se verán tan favorecidas en su desempeño. Dicho de otra forma, el SMP es más útil para las aplicaciones que hayan sido desarrolladas (o modificadas) para el procesamiento con múltiples hilos.

Para prevenir bloqueos mutuos o estancamientos u operaciones inválidas, se deben manipular adecuadamente las estructuras de datos del núcleo si tenemos varios procesadores ejecutando la misma o diferentes partes del núcleo. Los problemas descriptos en las secciones acerca de concurrencia (2.3.3) y bloqueos mutuos (5.7) se evidencian más en un ambiente con múltiples procesadores.

8.7.2 Planificación

El rol de un planificador es intercambiar entre hilos, decidir cuándo conmutar, qué hilo será activado y por cuánto tiempo ejecutará hasta que sea activado el próximo.

Por ende, la planificación en un multiprocesador involucra tres aspectos relacionados entre sí:

- La asignación de hilos a procesadores.
- El uso de la multiprogramación en procesadores individuales.
- La activación misma de un hilo.

En función de estos tres aspectos, es importante tener en cuenta que el enfoque del problema dependerá en general del grado de granularidad de las aplicaciones y del número de procesadores disponibles.

8.7.2.1 Asignación de hilos a procesadores

Si suponemos que tenemos múltiples procesadores simétricos, entonces el enfoque más simple respecto a la planificación es tratar los procesadores como un conjunto de recursos y asignar bajo demanda los hilos a los procesadores. El planteo que se presenta entonces es acerca de si la asignación debería ser estática o dinámica.

Si se asigna de forma permanente un hilo a un procesador, desde su activación hasta su finalización (como vimos en 8.6.3), entonces debe mantenerse una cola dedicada a corto plazo para cada procesador. Una ventaja de este enfoque es que puede haber menos sobrecarga en la función de planificación porque la asignación al procesador se hace una sola vez y para siempre. Además, el uso de procesadores dedicados permite una estrategia conocida como planificación por grupos o pandillas, que se trata más adelante.

Una desventaja de la asignación estática es que un procesador puede estar desocupado, con su cola vacía, mientras que otro procesador tiene trabajos pendientes. Para prevenir esta situación, se puede usar una cola común. Todos los hilos van a una cola global y son ejecutados en cualquier procesador que esté disponible. De este modo, durante la vida de un hilo, este puede ejecutarse en diferentes procesadores en momentos diferentes. Con acoplamiento muy fuerte y una arquitectura de memoria compartida, la información de contexto de todos los hilos se encuentra disponible para todos los procesadores y, por tanto, el coste de la planificación de un hilo será independiente de la identidad del procesador en el que fue planificado.

En una arquitectura simétrica, el sistema operativo puede ejecutarse en cualquier procesador y cada procesador se autoplanifica con el conjunto de hilos disponibles. Esta solución complica el sistema operativo, el cual debe asegurarse de que dos procesadores no seleccionen el mismo hilo y de que estos no se pierdan de la cola. Deben emplearse técnicas adecuadas para resolver y sincronizar las solicitudes concurrentes de recursos.

8.7.2.2 Uso de la multiprogramación en procesadores individuales

Cuando cada hilo se asigna estáticamente a un procesador durante todo su tiempo de vida, surge una nueva cuestión: ¿Puede estar multiprogramado dicho procesador? La primera reacción puede ser preguntarse por qué hace falta responder a la cuestión; puede parecer demasiado dispendioso vincular un procesador con un único hilo cuando dicho hilo puede bloquearse frecuentemente esperando una entrada o salida o por motivos de concurrencia o sincronización.

En los multiprocesadores tradicionales, con sincronización de grano grueso o independiente (ver tabla 8.2), está claro que cada procesador individual podría alternar entre varios procesos para conseguir una alta utilización y, por tanto, un mejor rendimiento. Sin embargo, cuando se trabaja con aplicaciones de grano medio que se ejecutan en muchos procesadores, la situación es menos clara. Cuando hay varios procesadores disponibles, no resulta tan importante que cada procesador esté ocupado al máximo. En su lugar, se debe tratar de obte-

ner el mejor rendimiento, en promedio, para las aplicaciones. Una aplicación que conste de varios hilos puede rendir poco, a menos que todos sus hilos estén disponibles para ejecutar simultáneamente.

8.7.2.3 Expedición de un proceso

El último punto de diseño relacionado con la planificación de multiprocesadores es la selección real del proceso que se va a ejecutar. Se ha visto que en un sistema monoprocesador multiprogramado el uso de prioridades o algoritmos de planificación sofisticados basados en la utilización anterior pueden mejorar la simple estrategia FIFO de decisión. Cuando se considera un multiprocesador, esta complejidad puede ser innecesaria o incluso contraproducente, según Stallings (2008). En el caso de la planificación tradicional de procesos, una técnica sencilla puede ser más efectiva y aportar menos sobrecarga. En los desarrollos recientes de planificación de hilos, entran en juego nuevos elementos que pueden ser más importantes que las prioridades o el historial de ejecución. Los sistemas operativos modernos, como Windows y Linux, han sabido dar respuesta a estos nuevos desafíos, como veremos.

8.7.2.4 Planificación de procesos

En la mayoría de los sistemas multiprocesador tradicionales, los procesos no se asignan a los procesadores de forma dedicada (no se utiliza **afinidad**). En su lugar, hay una cola única para todos los procesadores o, si se utiliza algún tipo de esquema de prioridades, existirán varias colas, según la prioridad, alimentando todas a un reservorio común de procesadores. En cualquier caso, es posible contemplar el sistema como una arquitectura de colas multiservidor.

8.7.2.5 Planificación de multiprocesadores

Planificación de hilos: los hilos han llegado a ser cada vez más habituales en los nuevos sistemas operativos y lenguajes. Como se ha visto, con los hilos el concepto de ejecución se separa del resto de la definición de un proceso. Una aplicación puede implementarse como un conjunto de hilos que cooperan y ejecutan concurrrentemente en el mismo espacio de direcciones.

En un monoprocesador, los hilos pueden usarse como una ayuda a la estructuración de un programa y para superponer la entrada y salida con el procesamiento. Puesto que la penalización por el intercambio de hilos es mínima, comparada con el intercambio de procesos, estos beneficios se llevan a cabo a bajo coste. Sin embargo, toda la potencia de los hilos se hace más evidente en un sistema multiprocesador donde pueden emplearse hilos para obtener un paralelismo real en las aplicaciones. Si los diversos hilos de una aplicación se ejecutan simultáneamente en distintos procesadores, se posibilita un aumento drástico del rendimiento. Sin embargo, puede demostrarse que, en aplicaciones que exijan una interacción considerable entre los hilos (paralelismo de grano medio), pequeñas diferencias en la planificación y gestión de hilos pueden tener un importante significado en el rendimiento, según Anderson, Lazowska y Levy (1989). Actualmente se está investigando en el campo de la planificación

de hilos en multiprocesadores. La discusión ofrecida aquí da una idea de los elementos y métodos clave.

Entre las diversas propuestas de planificación de hilos de multiprocesadores y de asignación de procesadores, destacan los siguientes métodos:

- Balance de carga: Es un concepto que se refiere a la técnica usada para compartir el trabajo que se va a realizar entre varios procesadores. El algoritmo divide de la manera más equitativa posible el trabajo para evitar los cuellos de botella. Se optimiza el uso de recursos, se maximiza el rendimiento y se minimiza el tiempo de respuesta.
- Planificación por grupos: Se planifica un conjunto de hilos afines para su ejecución en un conjunto de procesadores al mismo tiempo, según una relación de uno a uno.
- Asignación dedicada de procesadores: Es el enfoque opuesto a la autoplanificación y ofrece una planificación implícita definida por la asignación de hilos a los procesadores. En el tiempo en que se ejecuta un programa, a cada programa se le asigna un número de procesadores igual al número de hilos que posea. Cuando el programa finaliza, los procesadores retoman a la reserva general para posibles asignaciones a otros programas.
- Planificación dinámica: El número de hilos en un programa puede cambiar en el curso de una ejecución.

Balance de carga: el balance de carga es quizá el método más simple y el que se puede traducir más fácilmente desde un entorno monoprocesador. Tiene las siguientes ventajas:

- La carga se distribuye uniformemente entre los procesadores, asegurando que ningún procesador esté ocioso mientras haya trabajo disponible.
- No es necesario un planificador centralizado. Cuando un procesador está libre, la rutina de planificación del sistema operativo se ejecuta en este procesador para seleccionar un nuevo hilo.
- La cola global puede organizarse y se puede acceder a ella por medio de los esquemas ya tratados, incluyendo los esquemas basados en prioridades y los que tienen en cuenta el historial de ejecución o las solicitudes de proceso por adelantado.

Planificación por grupos: según Stallings (2008), el concepto de planificar un conjunto de procesos simultáneamente en un conjunto de procesadores es anterior al concepto de hilo. A. K. Jones y Schwarz (1980) se refieren a dicho concepto como **planificación por grupos** y citan las siguientes ventajas:

- Si procesos relativamente próximos se ejecutan en paralelo, pueden reducirse los bloqueos por sincronización, pueden hacer falta menos intercambios de procesos y se incrementará el rendimiento.
- La sobrecarga de planificación puede reducirse debido a que una sola decisión afecta a varios procesadores y procesos al mismo tiempo.

En el multiprocesador Cm*, se emplea el término **coplanificación**, según Gehringer, Siewiorek y Segall (1986). La coplanificación (o *coscheduling*, en inglés) se basa en el concepto de planificación de un conjunto de tareas afines, llamado **cuerpo de tareas**. Los elementos individuales de un cuerpo de tareas tienden a ser pequeños y son, por tanto, parecidos a la idea de hilo. El término fue introducido en el trabajo de Ousterhout (1982). Algunas técnicas

de coplanificación exhiben fragmentos de procesos que no se ejecutan concurrentemente con el resto del conjunto coplanificado. Estos algoritmos por lo general le restan importancia a la aparición de estos fragmentos. En cambio, la planificación por pandillas es una variante más estricta de la coplanificación que no permite fragmentos. El algoritmo de Ousterhout tiene en cuenta los patrones de comunicación para garantizar que se ejecuten simultáneamente todos los miembros del grupo. Para ello emplea una matriz conceptual, en la que las filas son los espacios de tiempo, y las columnas son las tablas de procesos de cada procesador. Todos los procesadores deben utilizar un algoritmo de planificación *round robin*, para que durante un período de tiempo fijo los procesadores ejecuten el proceso en el espacio «0», después en el espacio «1», y así sucesivamente, manteniendo sincronizados los intervalos de tiempo.

El término **planificación por pandillas** (o *gang scheduling*, en inglés) se ha aplicado a la planificación simultánea de hilos que forman parte de un único proceso, según Feitelson y Rudolph (1990). Es necesaria para aplicaciones paralelas de grano medio a fino cuyo rendimiento se degrada seriamente cuando alguna parte de la aplicación no está ejecutando mientras otras partes están listas para ejecutar. Es también beneficiosa para cualquier aplicación paralela, incluso aquellas que no son lo bastante sensibles al rendimiento. La necesidad de planificación por pandillas está muy reconocida y existen implementaciones en una gran variedad de sistemas operativos para multiprocesadores.

Una razón obvia por la que la planificación por grupos mejora el rendimiento de una sola aplicación es que se minimizan los intercambios de procesos. Supóngase que un hilo de un proceso está ejecutando y alcanza un punto en el cual debe sincronizarse con otro hilo del mismo proceso. Si ese otro hilo no está ejecutando pero está listo para ejecutar, el primer hilo se queda colgado hasta que se puede realizar un intercambio en otro procesador para traer el hilo que se necesita. En una aplicación con una coordinación estrecha entre sus hilos, tales intercambios reducen drásticamente el rendimiento. La planificación simultánea de los hilos cooperantes puede, además, ahorrar tiempo en el reparto de recursos. Por ejemplo, varios hilos planificados por grupos pueden acceder a un archivo sin el coste adicional del bloqueo durante la operación de búsqueda, lectura o escritura en él.

Asignación dedicada de procesadores: una forma extrema de planificación por grupos, propuesta por Tucker y Gupta (1989), consiste en dedicar un grupo de procesadores a una aplicación mientras dure esta. Es decir, cuando se planifica una aplicación, se asigna cada uno de sus hilos a un procesador que permanece dedicado a ese hilo hasta que la aplicación termine su ejecución.

Este método podría parecer demasiado dispendioso en tiempo de procesador. Si un hilo de una aplicación se bloquea en espera de una entrada o salida o por sincronización con otro hilo, el procesador de dicho hilo quedará desocupado, pues no hay multiprogramación de procesadores. Se pueden plantear varias observaciones en defensa de esta estrategia:

1. En un sistema masivamente paralelo, con decenas o cientos de procesadores, cada uno de los cuales representa una pequeña parte del coste del sistema, la utilización del procesador no es tan importante como medida de la efectividad o el rendimiento.
2. La anulación total del intercambio de procesos durante el tiempo de vida de un programa dará como resultado una aceleración sustancial del programa.

Tanto la asignación dedicada de procesadores como la planificación por grupos abordan el problema de la planificación haciendo frente a la asignación del procesador. Se puede ob-

servar que el problema de asignación del procesador en un multiprocesador se asemeja más al problema de asignación de memoria que al problema de planificación de un monoprocesador. La clave está en cómo se asignan muchos procesadores a un programa en un instante dado, que es análogo a cómo se asignan varios marcos de página a un proceso en un instante dado. Gehringer, Siewiorek y Segall (1986) proponen el concepto de **conjunto de trabajo de actividades**, análogo al conjunto de trabajo en memoria virtual, como el número mínimo de actividades (hilos) que deben ser planificadas simultáneamente en los procesadores para que la aplicación tenga un progreso aceptable. Como en los esquemas de gestión de memoria, el fallo al planificar todos los elementos de un conjunto de trabajo de actividades puede conducir a una **hiperplanificación del procesador** (*processor thrashing*), que se produce cuando la planificación de hilos cuyos servicios son necesarios induce la desplanificación de otros hilos cuyos servicios se necesitarán pronto. De forma similar, la fragmentación del procesador se refiere a la situación en la que sobran algunos procesadores mientras otros están asignados y los procesadores sobrantes son insuficientes en número o están mal organizados para dar soporte a los requisitos de las aplicaciones que esperan. La planificación por grupos y la asignación dedicada de procesadores intentan evitar esos problemas.

Planificación dinámica: en algunas aplicaciones, es posible emplear un lenguaje y unas herramientas del sistema que permitan cambiar dinámicamente el número de hilos de un proceso para permitir al sistema operativo ajustar la carga y mejorar así la utilización.

Zahorjan y McCann (1990) proponen un método en el que tanto el sistema operativo como la aplicación están involucrados en la toma de decisiones de planificación. El sistema operativo es responsable de repartir los procesadores entre los trabajos. Cada trabajo emplea los procesadores de su partición para procesar un subconjunto de sus tareas ejecutables, organizando estas tareas en hilos. A las aplicaciones individuales se les deja (quizá mediante una biblioteca de rutinas de tiempo de ejecución) la decisión sobre el subconjunto por ejecutar, además de la de qué hilo suspender cuando se expulsa un proceso. Este método puede no resultar conveniente para todas las aplicaciones. Sin embargo, algunas aplicaciones podrían limitarse a un único hilo, mientras que otras podrían ser programadas para sacar partido de esta característica del sistema operativo.

Con este método, la responsabilidad de planificación del sistema operativo se limita sobre todo a la asignación del procesador y actúa según la siguiente política. Cuando un trabajo solicita uno o más procesadores (tanto si el trabajo llega por primera vez como si cambian sus necesidades):

1. Si hay procesadores desocupados, se usan para satisfacer la petición.
2. En otro caso, si el trabajo que realiza la petición está recién llegado, se le asigna un procesador individual quitándoselo a algún proceso que tenga más de un procesador asignado.
3. Si no se puede satisfacer alguna parte de la petición, queda pendiente hasta que un procesador pase a estar disponible o hasta que el trabajo anule la petición (por ejemplo, si ya no hacen falta más procesadores).

   Al liberar uno o más procesadores (incluyendo la salida del trabajo):
4. Explorar la cola de peticiones de procesador no satisfechas y asignar un solo procesador a cada trabajo de la lista que no tiene aún procesador (es decir, a todas las nuevas llegadas que estén en espera).

Después, se recorre la lista de nuevo, asignando el resto de procesadores según un FCFS.

Algunos análisis expuestos por Zahorjan y McCann (1990) y Majumdar, D. L. Eager y Bunt (1988) proponen que, para las aplicaciones que pueden sacar partido de la planificación dinámica, este método es superior a la planificación por grupos o la asignación dedicada de procesadores. Sin embargo, el coste de este método puede anular la aparente mejora de rendimiento. Para probar la validez de la planificación dinámica hace falta más experiencia con los sistemas reales.

8.7.3 Sincronización

Los métodos de sincronización son necesarios cuando existe la propiedad de concurrencia, y la concurrencia existe cuando dos o más procesos ejecutan en el mismo período de tiempo y potencialmente interactúan uno con otro.

Como vimos, la concurrencia puede ocurrir sobre uniprocesadores en los que varios hilos comparten la misma CPU y el desalojo crea condiciones de carrera, pero la concurrencia es más frecuente en sistemas SMP en los que los hilos que ejecutan simultáneamente en cada procesador comparten los mismos datos.

Para ejecutar bien en sistemas SMP, los distintos procesadores necesitan coordinar los accesos a estructuras de datos. Las diferentes estructuras de datos se usan de diferentes maneras y, debido a ello, los sistemas operativos deberían tener un amplio rango de mecanismos de sincronización.

8.7.3.1 Estructuras de datos locales a la CPU

No es realmente un mecanismo de sincronización SMP pero merece mencionarse porque realmente ayuda al sistema a trabajar en sistemas SMP y usualmente escala mejor que cualquier estructura de datos compartidos. El truco es simplemente dar a cada CPU su propia estructura de datos y no acceder a la estructura de datos de otras CPU. Cuando se usa este método para estadísticas la función que las muestra simplemente lee la variable en cuestión para cada CPU en el sistema y las calcula.

Este método se usa en el planificador (colas de ejecución por CPU), en la VM (estadísticas) y en otras áreas del núcleo. Note que las estructuras de datos locales a la CPU no funcionan si necesitan compartirse entre todas las CPU.

8.7.3.2 Spinlocks

Este es el mecanismo más simple de sincronización en SMP. Como decíamos en los apartados 4.1.4 y 5.2.4.4, se protege una estructura de datos mediante una traba que sólo una CPU puede tomar por vez. Cuando una CPU solicita una traba que ya se encuentra en poder de otra CPU, entonces simplemente espera (espín proviene del inglés *spin* y significa «giro») en un lazo hasta que la libere –es decir, tampoco habrá una cola de espera. Por supuesto, esto significa que el código que está ejecutando y manteniendo la traba no puede invocar al planificador. Es una manera de asegurar exclusión mutua mediante el uso de una traba de espera activa (*busy wait*).

Entonces, los *spinlocks* giran sobre una traba y se usan a menudo para proteger estructuras de datos que se leen y escriben frecuentemente y sólo necesitan ser trabadas por poco tiempo porque, si bien la espera activa puede parecer ineficiente, en realidad puede ser más rápida que poner el hilo a dormir y luego despertarlo más tarde cuando la traba se haya liberado.

En Linux hay *spinlocks* específicos para el problema de los lectores/escritores y se denominan `rwlocks`, para situaciones en las que es ventajoso tener varios lectores accediendo simultáneamente a la estructura de datos pero sólo a un único escritor sin lectores concurrentes. Sin embargo, debido al corto tiempo por el que se toman los *spinlocks* esto no es muy común.

8.7.3.3 Semáforos - mutexes

Los semáforos son un poco parecidos a los *spinlocks*, excepto que el poseedor de un semáforo es un proceso, no una CPU. Para un proceso es seguro dormir o ser planificado mientras retiene un semáforo. Los semáforos se usan cuando la traba puede quedar retenida por largo tiempo, por ejemplo durante un fallo de página en el que un proceso puede necesitar esperar a que complete una entrada y salida de disco antes de que sea atendido el fallo de página.

También hay semáforos para lectura y escritura para situaciones en las que es beneficioso tener varios lectores simultáneamente.

Recientemente en la serie 2.6 el núcleo mejoró a los mutex, los cuales tienen más o menos la misma semántica que los semáforos. Las implementaciones de semáforos específicas de una arquitectura han sido reemplazadas por otras genéricas. Los mutex son más rápidos que los semáforos pero no tienen la equidad entre las CPU. Un mutex debe ser destrabado por la tarea que lo trabó, y una tarea no puede trabar de nuevo a un mutex ya trabado. Si el núcleo está compilado con la opción DEBUG_MUTEXES, estas reglas se hacen cumplir forzosamente.

En la mayoría de las situaciones se utilizarán los mutex.

8.7.3.4 Traba de lectura rápida

La traba de lectura rápida (del inglés *fast reader lock*) no es una traba en sentido estricto.



Es un mecanismo de sincronización que usa números de secuencia para asegurarse de que los lectores obtengan un estado consistente de una estructura de datos.

Se usa sobre las estructuras de datos que son leídas con una frecuencia tal que una traba para efectuar una operación de lectura y escritura podría provocar la inanición de los escritores.

En este mecanismo, en cambio, los escritores siempre pueden operar escribiendo la estructura de datos. Los lectores simplemente leerán la estructura de datos de nuevo si esta cambió durante la lectura.

Esto se implementa de la siguiente manera: el lector lee el número de secuencia de la estructura de datos, luego lee el dato que necesita y finalmente lee el número de secuencia otra vez. Si el número de secuencia cambió durante la lectura del dato, simplemente reintenta toda la operación. Si el número de secuencia es el mismo, se debe a que la estructura de datos no fue escrita y por lo tanto se interpreta que el dato es válido.

Un escritor incrementa el número de secuencia antes y después de modificar la estructura de datos. De esta manera los lectores pueden ver si la estructura de datos se está cambiando o no.

8.7.3.5 Leer-Copiar-Actualizar

Del inglés *Read-Copy-Update* (o RCU), es un esquema sin traba para sincronizar el acceso a una estructura de datos compartida que es mayormente de sólo lectura. Se utiliza con mayor frecuencia para listas enlazadas, en particular para borrar las entradas de listas enlazadas.

Agregar una entrada a una lista enlazada puede hacerse con bastante facilidad y sin trabas, ya que crear punteros en cabeceras de listas es atómico. Basta con configurar las entradas anterior y siguiente a la derecha y agregar su entrada a la cola de la lista enlazada.

La eliminación es un poco más complicada, ya que otras CPU podrían estar utilizando la entrada de la lista enlazada que desea eliminar. RCU simplemente desvincula la entrada de la lista enlazada y la mantiene en torno (alrededor) hasta que todas las CPU en el sistema se hayan planificado. En ese momento, nadie seguirá usando la entrada y puede ser liberada.

RCU es un mecanismo de sincronización que logra mejoras en la escalabilidad permitiendo que las lecturas se efectúen concurrentemente con las actualizaciones. En contraste con los mecanismos tradicionales que aseguran exclusión mutua entre hilos concurrentes sin tomar en cuenta si serán para efectuar lecturas o actualizaciones, o como ocurre con las trabas lector/escritor que permiten lecturas concurrentes pero no en presencia de actualizaciones. RCU soporta concurrencia entre una única actualización y varias lecturas.

8.7.3.6 Barrera

Una barrera es un método de sincronización. Con el fin de permitirle al programador o al compilador clarificar las semánticas del código, se declaran las **barreras** que les indican a los hilos o procesos que deben detenerse en este punto y que no pueden seguir adelante hasta que los restantes hilos o procesos hayan alcanzado la barrera.

8.7.4 Gestión de la memoria

Para conectar varias unidades de procesamiento o bien las CPU deben soportar estas conexiones de alguna manera o se debe agregar alguna lógica que los ligue. El subsistema también puede estar consciente o no de la presencia de varias CPU. Los procesadores actuales tienen soporte para SMP. Los chips modernos tienen a menudo un colector separado dedicado a la unidad de procesamiento y al controlador de la memoria, por lo cual el número físico del *slot* de la unidad de procesamiento implica la percepción de más de una CPU.

Hay esencialmente dos grupos principales de problemas involucrados en lograr el multiprocesamiento simétrico en hardware. Uno es el problema que concierne a los accesos a memoria de las CPU; el otro es la comunicación de las CPU con los dispositivos de entrada y salida. En el primero, deben manejarse problemas tales como mantener una visión coherente de la memoria y compartir eficientemente el colector. En el segundo grupo, por ejemplo, se encuentran los problemas que conciernen al manejo de las interrupciones.

8.7.4.1 Arquitectura de Acceso Uniforme a Memoria (UMA)

Si todos los procesos acceden a la memoria de manera uniforme, y si el tiempo de acceso o la localización de memoria es independiente del proceso que realiza la petición o del chip de memoria donde se encuentre el dato solicitado, entonces nos encontramos con una arquitectura UMA (del inglés *Uniform Memory Access*, o Acceso Uniforme a Memoria), como se ilustra en la figura 8.15.

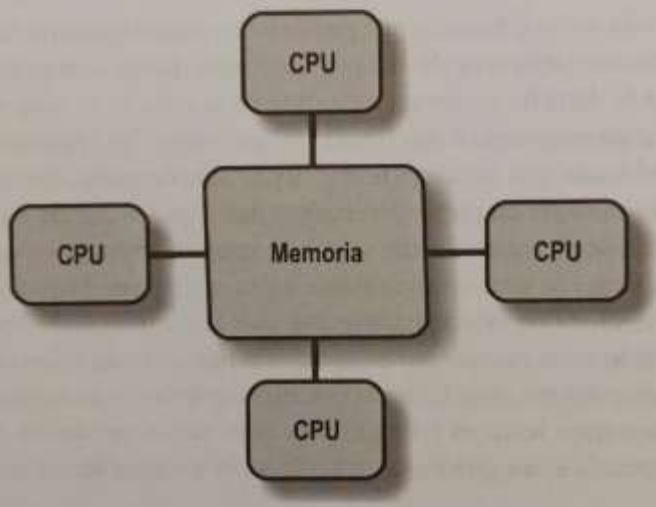


Fig. 8.15 – Memoria compartida UMA.

A veces también se la ha llamado CC-UMA o *Cache Coherent UMA* porque si un procesador actualiza una ubicación en esta memoria compartida, esta actualización se hace visible a todos los otros procesadores. Esta coherencia de caché se lleva a cabo a nivel del hardware.

8.7.4.2 Arquitectura de Acceso No Uniforme a Memoria (NUMA)

Por otra parte, podemos tener otro diseño de memoria en el que esta se accede en posiciones relativas de otro proceso o memoria compartida entre procesos, como se ilustra en la figura 8.16.

Por lo tanto, los tiempos de acceso diferirán (serán no-uniformes). Un procesador podrá acceder a su propia memoria local de forma más rápida que a la memoria no local (memoria local de otro procesador o memoria compartida entre procesadores). En NUMA, cada dirección en el espacio de direcciones global está asignada a un nodo fijo. Cuando los proce-

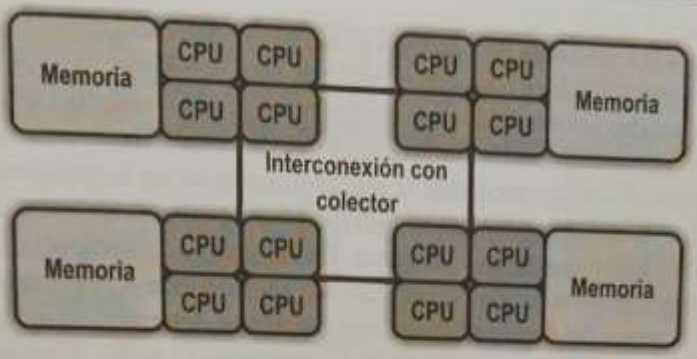


Fig. 8.16 – Memoria compartida NUMA.

sadores acceden a algún dato, se hace una copia en sus cachés locales pero el espacio continúa reservado en el nodo original.

8.7.4.3 Arquitectura de Memoria de Sólo Caché (COMA)

Un multiprocesador que sólo use caché como memoria es considerado de tipo COMA (del inglés *Cache-Only Memory Architecture*, Arquitectura de Memoria de Sólo Caché) en contraste con el uso de las memorias locales como verdaderas memorias principales (por ejemplo, DRAM), como en las arquitecturas NUMA. En COMA no hay nodos que contengan los datos originales y tampoco hay jerarquía de memoria en cada módulo procesador. Todas las cachés forman un mismo espacio global de direcciones. El acceso a las cachés remotas se realiza a través de los directorios distribuidos de las cachés. Dependiendo de la forma de interconexión (redes en los primeros equipos o colectores en los más modernos) empleada, se pueden utilizar jerarquías en los directorios para ayudar en la localización de copias de bloques de caché. El emplazamiento inicial de datos no es crítico puesto que el dato acabará estando en el lugar en que se use más.

Comparado con NUMA, esto reduce el número de copias redundantes y puede permitir un uso más eficiente de los recursos de memoria.

Dahlgren y Torrellas 1999 afirman que COMA incrementa las posibilidades de que los datos estén disponibles localmente al procesador que los usa porque el hardware replica transparentemente los datos y los migra al módulo de memoria del nodo que lo está accediendo. Así, se reduce el impacto de las frecuentes y largas latencias de accesos a la memoria.

Esta arquitectura se utilizó en el *Data Diffusion Machine* (DDM) y se expuso en el trabajo de Hagersten, Haridi y Warren (1990) y posteriormente en Hagersten, Landin y Haridi (1992).

8.7.5 Coherencia y consistencia

Diremos informalmente que una memoria es **coherente** si el valor obtenido en una operación de lectura es siempre el valor que el programador esperaba. El término **consistencia** se usa para referirse a un tipo específico de coherencia. Ambos términos se utilizan frecuentemente (y a veces indistintamente) en el ámbito de los sistemas distribuidos y, si bien no

estamos tratando a los sistemas de memoria compartida distribuida, muchos de sus problemas y soluciones son aplicables a los sistemas SMP.

Algo común a todos los modelos de consistencia es que el compilador tiene que ser cuidadoso en no reorganizar instrucciones cruciales para la consistencia. Muchos compiladores modernos optimizan el código mediante la reorganización de las instrucciones, violando así el orden del código fuente. El orden del código fuente es la visión semántica del programador acerca de cómo se deberían ejecutar las cosas. En un programa secuencial, la ejecución de una función es correcta si la salida de una función corresponde a la función correcta de su entrada (y/o del tiempo). Se dice que un sistema soporta un determinado modelo si las operaciones en memoria siguen normas específicas. La consistencia de los datos se define entre el programador y el sistema, que garantiza que si el programador sigue las reglas, la memoria será **consistente** y el resultado de las operaciones de memoria será predecible.

La coherencia de memoria hace referencia a la necesidad de establecer la lógica necesaria para que los distintos datos replicados a lo largo de la jerarquía de memoria contengan la misma información si se trata de la misma dirección física. Por ejemplo, un dato que se encuentra en el nivel más alto de la jerarquía, esto es, en la caché de L1, y que ha sido modificado, deberá activar los mecanismos necesarios para que esta modificación se lleve a cabo en el resto de niveles, donde este dato también existe.

Esto puede cobrar más relevancia en el caso de los procesadores multinúcleos, donde, en general, se comparte un nivel de caché, L2 o Ln (donde n podrá ser cualquier número natural), pero cada núcleo dispone de su propia L1 donde modifica los datos, pudiendo haber una incoherencia de los datos que lee un núcleo y que han sido modificados recientemente por otro y disponiendo el dato sin modificar en la caché compartida.

En el caso de los procesadores mononúcleo, la lógica de control de coherencia de caché del procesador, es decir, la circuitería interna que en este caso concreto se encarga de gestionar el trasvase de la información de un lugar a otro, se encargará de modificar en los niveles inferiores de la jerarquía aquellos datos que hayan sido modificados en L1, con el objetivo de que cuando este desaparezca de la caché, el dato se encuentre actualizado en los siguientes niveles de caché, memoria principal y disco.

Estos mecanismos de control, por tanto, sólo se activarán como parece lógico pensar, en los casos en los que se produzca una escritura. Faltaría añadir que dentro de la caché y de sus distintas implementaciones, nos podemos encontrar con una lógica de control de mayor o menor complejidad, dependiendo de si es una caché exclusiva o inclusiva. En la inclusiva, nos encontramos una circuitería más simple, mientras que en la exclusiva aumenta la complejidad. Este problema fue presentado ya por A. J. Smith (1982).

La coherencia de caché hace referencia a la integridad o consistencia de los datos almacenados en las cachés locales de los recursos compartidos. La coherencia de la caché es un caso especial de la coherencia de memoria. Cuando los clientes de un sistema, en particular las CPU en un sistema multiprocesador, mantienen cachés de una memoria compartida, los conflictos crecen.

Haciendo referencia a la figura 8.17, si el cliente de arriba tiene una copia de un bloque de memoria de una lectura previa y el cliente de abajo cambia ese bloque, el cliente de arriba podría estar trabajando con datos erróneos sin tener conocimiento de ello. La coherencia de la caché intenta administrar estos conflictos y mantener consistencia entre las cachés y la memoria.

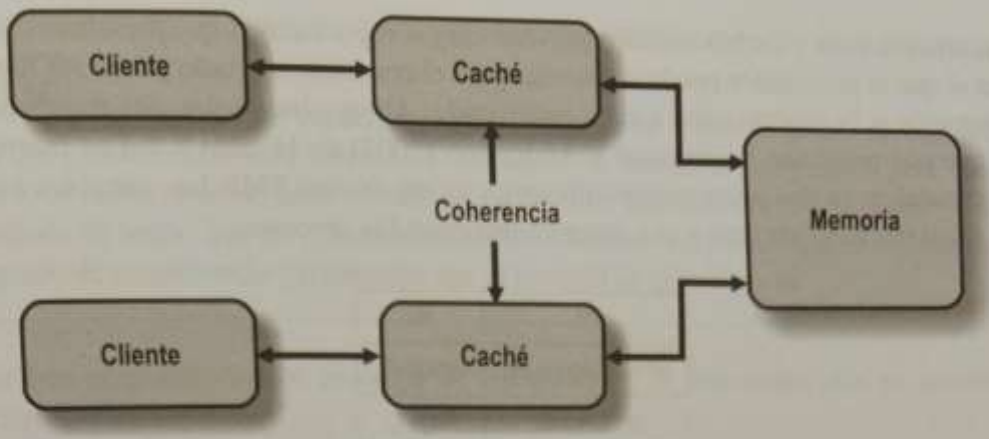


Fig. 8.17 – Coherencia de caché.

La solución al problema de la coherencia de la caché usualmente incluye invalidar todas excepto una de las líneas duplicadas cuando la línea se modifica. Si bien el hardware utiliza lógica de fisgoneo o *snooping* para invalidar, sin intervención del software, cualquier procesador cuya línea de caché haya sido invalidada tendrá un fallo de caché, con su retardo concomitante, la próxima vez que esa línea sea referenciada. La lógica de *snooping* en el procesador difunde un mensaje por el colector cada vez que una palabra en su caché ha sido modificada. La lógica *snooping* también fisgonea en el colector buscando esos mensajes desde otros procesadores.

Cuando un procesador detecta que otro procesador ha cambiado un valor en una dirección existente en su propia caché, la lógica *snooping* invalida esa entrada en su caché. A esto se lo llama **invalidación cruzada**. La invalidación cruzada le recuerda al procesador que el valor en la caché no es válido y que debe buscar el valor correcto en algun lugar más (en la memoria u otra caché). Dado que la invalidación cruzada incrementa los fallos de caché y el protocolo de *snooping* agrega tráfico al colector, resolver el problema de consistencia de la caché reduce el desempeño y la escalabilidad de todos los SMP.

8.7.5.1 Ordenamiento y consistencia de la memoria

Los procesadores modernos a menudo están basados en un diseño superescalar, como afirman Fredriksson y Andersson (2003). Un procesador super escalar consiste de varias unidades funcionales, por ejemplo algunas unidades aritméticas para números enteros, unidades de punto flotante y unidades de carga y almacenamiento. La ventaja que se deriva de esto es que el procesador puede emitir varias instrucciones simultáneamente a diferentes unidades funcionales. Sin embargo, no todas las instrucciones son independientes con respecto a la otra y no es común que el flujo de instrucción en un programa permita una buena utilización de estas unidades funcionales si las instrucciones se debieran completar en el orden del programa. Para permitir que la ejecución proceda más allá de una instrucción de carga o almacenamiento, la mayoría de las cachés son de las llamadas no bloqueantes (*non-blocking caches*, en inglés). Esto esencialmente significa que los comandos de almacenamiento son resguardados en un *buffer* y llevados a cabo con posterioridad para ocultar los retardos del colector.

Algunos procesadores y cachés también proveen cargas especulativas que presentan un resultado con el que el procesador puede continuar, pero el resultado afectado puede ser revocado posteriormente si la especulación estaba equivocada. Ahora observe los dos fragmentos de código que nos proponen Fredriksson y Andersson (2003) en la tabla 8.3. Los fragmentos corren en paralelo en dos procesadores diferentes en un sistema SMP. Las variables `result` y `flag` valen inicialmente cero y son compartidas entre los procesos.

P1	P2
while (flag == 0);	result=5;
print result;	flag=1;

Tabla 8.3 – Ejemplo de variable compartida.

El resultado deseado e intuitivo, según la semántica del programador (el del orden del programa) de estas operaciones, es que `P1` debería imprimir el número 5. Sin embargo, si se le permite al procesador reordenar las instrucciones de ejecución arbitrariamente, la asignación de `flag` en `P2` podría ser visible a `P1` antes que la asignación a `result` en `P2`. También es posible que `P1` ejecute la instrucción `print` antes de salir del bucle `while`. Ambos escenarios darán como resultado que `P1` imprima «incorrectamente» el valor cero.

Para evitar estos problemas de consistencia a menudo se le aplican al procesador un conjunto de reglas de ordenamiento. Estas se conocen como **modelos de consistencia**.

8.7.5.2 Modelos de consistencia

En un sistema multitarea y especialmente en sistemas paralelos, una función puede tener efectos secundarios como el problema de consistencia con las variables compartidas en la tabla 8.3. Suponiendo que los dos fragmentos de código se compilen de forma separada, es probable que el compilador no reconozca los efectos secundarios de la asignación de `flag`. Cuando el compilador no puede identificar estos efectos secundarios, las instrucciones reordenadas arruinan las semánticas combinadas de los fragmentos. Además, los compiladores hacen alias entre las variables y los registros del procesador, las cuales no están de ninguna manera compartidas entre los procesadores. Por ejemplo, considere el efecto en la tabla 8.3 si `P1` forma un alias entre la variable `flag` y un registro. En lenguaje C, esto puede evitarse mediante el uso de la clase `volatile` para variables que deben estar explícitamente en memoria.

Los requerimientos de consistencia varían de una aplicación a otra, según se desprende a partir de los trabajos de Cheriton (1985), de Bennett, Carter y Zwaenepoel (1990) y de Garcia-Molina y Wiederhold (1982).



Un **modelo de consistencia** básicamente se refiere al grado de consistencia que debe mantenerse con respecto a los datos en memoria compartida para que esta funcione correctamente para un cierto conjunto de aplicaciones. Se la define como un conjunto de reglas que las aplicaciones deben obedecer si quieren que el sistema provea el grado de consistencia garantizado por el modelo de consistencia.

Se han propuesto varios modelos de consistencia, de los cuales sólo se describen los principales a continuación.

Aquí debería notarse que la investigación de nuevos modelos de consistencia es actualmente un área activa de investigación. La idea básica es inventar un modelo de consistencia que permita relajar los requerimientos de consistencia a un mayor grado que los modelos de consistencia existentes, de tal manera que pueda funcionar correctamente un conjunto de aplicaciones. Esto ayuda a mejorar el rendimiento de estas aplicaciones porque se puede alcanzar mejor concurrencia. Sin embargo, las aplicaciones que dependen de un modelo de consistencia mas fuerte pueden no tener un desempeño correcto si se ejecutan en un sistema que soporta sólo un modelo de consistencia más débil. Esto es porque si un sistema soporta el modelo de consistencia más fuerte, entonces el modelo de consistencia más débil está automáticamente soportado, pero lo opuesto no es cierto.

8.7.5.3 Modelo de consistencia estricta

El modelo de consistencia estricta es la forma más fuerte de coherencia de memoria y tiene el requerimiento de consistencia más riguroso.



Se dice que un sistema de memoria compartida soporta el modelo de consistencia estricto si el valor retornado por una operación de lectura sobre una dirección de memoria es el mismo que el valor escrito por la operación de escritura más reciente sobre esa misma dirección, sin tener en cuenta las ubicaciones de los procesos que realizan estas operaciones de lectura y escritura.

Es decir, todas las escrituras instantáneamente se hacen visibles a todos los procesos.

La implementación del modelo de consistencia estricto requiere de la existencia de un tiempo global absoluto de manera tal que las operaciones de lectura y escritura puedan ser ordenadas correctamente para dejar en claro el significado de «más reciente». En un sistema SMP el reloj es único y local, como se explica en el Capítulo 5, pero no es posible la sincronización absoluta de los relojes de todos los nodos de un sistema distribuido y, por lo tanto, tampoco es posible la existencia de un tiempo global absoluto en un sistema distribuido. Entonces, como consecuencia, es prácticamente imposible la implementación del modelo de consistencia estricto para un sistema de memoria compartida distribuida.

Es más, este modelo de consistencia no se puede implementar en ningún sistema real. Efectivamente, como la distancia entre los procesadores y la memoria no es nula, la señal que viaja entre el procesador que escribe y la memoria sufre un cierto retraso. Si durante este retraso, en el cual la memoria no tiene aún el nuevo dato escrito, otro procesador decide leer, no va a leer el valor que se acaba de mandar a la memoria, que aún no ha llegado, sino el que hubiera anteriormente. Por lo tanto, la única forma en la que se podría implementar este modelo es suponiendo que los datos viajan de forma instantánea (velocidad infinita).

8.7.5.4 Modelo de consistencia secuencial

El modelo de consistencia secuencial fue propuesto por Lamport (1979).



Se dice que un sistema de memoria compartida soporta el modelo de consistencia secuencial si todos los procesos ven el mismo orden de todas las operaciones de acceso a memoria sobre la memoria compartida.

Un multiprocesador es secuencialmente consistente si el resultado de cualquier ejecución es el mismo que si las operaciones de todos los procesadores se realizaran en algún orden secuencial, y las operaciones de cada procesador individual ocurren en esta secuencia en el orden especificado por el programa.

No importa el orden exacto en el cual se intercalan las operaciones de acceso a memoria. Esto es, si las tres operaciones read (r_1), write (w_1), read (r_2) se realizan sobre una dirección de memoria en ese orden, cualquiera de los ordenamientos (r_1, w_1, r_2), (r_1, r_2, w_1), (w_1, r_1, r_2), (w_1, r_2, r_1), (r_2, r_1, w_1), (r_2, w_1, r_1) de las tres operaciones es aceptable suponiendo que todos los procesos vean el mismo orden. Si uno de los procesos ve uno de los ordenamientos de las tres operaciones y otro proceso ve uno diferente, la memoria no es una memoria secuencialmente consistente. Note aquí que el único ordenamiento aceptable para una memoria estrictamente consistente es (r_1, w_1, r_2).

El requerimiento de consistencia del modelo de consistencia secuencial es más débil que el modelo de consistencia estricto porque el primero no garantiza que una operación de lectura sobre una direción de memoria en particular retorne el mismo valor tal como lo escribió la operación de escritura más reciente sobre esa dirección. Como consecuencia, con una memoria secuencialmente consistente, correr un programa dos veces puede no dar el mismo resultado en la ausencia de operaciones de sincronización explícitas. Este problema no existe en una memoria estrictamente consistente.

Un sistema de memoria compartida distribuida que soporte el modelo de consistencia secuencial puede implementarse asegurando que no se inicie una operación de memoria hasta que todas las previas se hallan completado.

Este modelo no da lugar al reordenamiento en hardware o a la superposición, ya que los procesadores más comunes no tienen una noción de qué operaciones de memoria requieren orden secuencial y cuáles no.

8.7.5.5 Modelo de consistencia causal

El modelo de consistencia causal (o *causal consistency*, en inglés), propuesto por Hutto y Ahamad (1990) y definido en Ahamad y col. (1995), relaja el requerimiento del modelo de consistencia secuencial para una mejor concurrencia. A diferencia del modelo de consistencia secuencial, en el modelo de consistencia causal todos los procesos ven sólo aquellas operaciones de referencia a memoria en el mismo orden (correcto) en que están potencialmente relacionadas en forma causal. Las operaciones de referencia a memoria que no están potencialmente relacionadas en forma causal pueden ser vistas por procesos diferentes en órdenes diferentes. Se dice que una operación de referencia a memoria (*read/write*) está potencialmente relacionada en forma causal a otra operación de referencia de memoria si la primera puede haber sido influenciada de alguna manera por la segunda. Por ejemplo, si un proceso realiza una operación de lectura seguido por una operación de escritura, la operación de escritura está potencialmente relacionada en forma causal a la operación de lectura porque la computación del valor escrito puede haber dependido de alguna forma del valor obtenido por la operación de lectura. Por otra parte, una operación de escritura realizada por un proceso no está causalmente relacionada a una operación de escritura realizada por otro proceso si el primer proceso no ha leído el valor escrito por el segundo proceso o cualquier variable de memoria que fue directa o indirectamente derivada del valor escrito por el segundo proceso.



Se dice que un sistema de memoria compartida soporta el modelo de consistencia causal si todas las operaciones de escritura que están potencialmente relacionadas de forma causal son vistas por todos los procesos en el mismo orden (correcto).

Las operaciones de escritura que no están potencialmente relacionadas en forma causal pueden ser vistas por procesos diferentes en órdenes diferentes. Note que «orden correcto» significa que si una operación de escritura (w_2) está causalmente relacionada a otra operación de escritura (w_1), el orden aceptable es (w_1, w_2) porque el valor escrito por w_2 puede haber sido influenciado de alguna manera por el valor escrito por w_1. Por lo tanto, (w_2, w_1) no es un orden aceptable.

Obviamente, en la implementación de un sistema de memoria compartida que soporte el modelo de consistencia causal hay una necesidad de seguir la pista de cuáles operaciones de referencia a memoria son dependientes de cuáles otras operaciones del mismo tipo. Esto puede hacerse al construir y mantener un grafo de dependencia para las operaciones de acceso a memoria.

8.7.5.6 Modelo de consistencia PRAM

El modelo de consistencia de memoria de acceso al azar encauzado (*pipelined random-access memory PRAM*), propuesto por Lipton y Sandberg (1988), provee una semántica de consistencia más débil que los modelos de consistencia descriptos anteriormente. Sólo asegura que todas las operaciones de escritura llevadas a cabo por un proceso sean vistas por todos los otros procesos en el orden en el cual fueron llevadas a cabo, como si todas las operaciones

de escritura realizadas por un proceso estuvieran en una tubería. Las operaciones de escritura llevadas a cabo por procesos diferentes pueden ser vistas por procesos diferentes en órdenes diferentes. Por ejemplo, si w_{11} y w_{12} son dos operaciones de escritura llevadas a cabo por un proceso P_1 en ese orden, y w_{21} y w_{22} son dos operaciones de escritura realizadas por un proceso P_2 en ese orden, un proceso P_3 puede verlos en el orden $[(w_{11}, w_{12}), (w_{21}, w_{22})]$ y otro proceso P_4 puede verlos en el orden $[(w_{21}, w_{22}), (w_{11}, w_{12})]$.

El modelo de consistencia PRAM es simple y fácil de implementar y también tiene buen desempeño. Puede implementarse mediante la secuenciación de las operaciones de escritura llevadas a cabo en cada nodo independientemente de las operaciones de escritura llevadas a cabo en otros nodos. Conduce a un mejor desempeño que los modelos previos porque un proceso no necesita esperar que complete una operación de escritura realizada por él antes de comenzar la siguiente dado que todas las operaciones de escritura de un proceso están entubadas. Nótese que en la consistencia secuencial todos los procesos acuerdan en el mismo orden de las operaciones de referencia a memoria, pero en la consistencia PRAM no. Por lo tanto, para el ejemplo dado anteriormente, ya sea $[(w_{11}, w_{12}), (w_{21}, w_{22})]$ o $[(w_{21}, w_{22}), (w_{11}, w_{12})]$ son un ordenamiento aceptable para una consistencia secuencial, pero no ambos. Esto es, a diferencia de la consistencia PRAM, ambos procesos P_3 y P_4 deben estar de acuerdo en el mismo orden.

8.7.5.7 Modelo de consistencia de procesador

El modelo de consistencia de procesador propuesto por Goodman (1989) es muy similar al modelo de consistencia PRAM con una restricción adicional de coherencia de memoria. Es decir, una memoria de consistencia de procesador es coherente y adhiere al modelo de consistencia PRAM. La coherencia de memoria significa que, para cualquier ubicación de memoria, todos los procesos están de acuerdo en el mismo orden para todas las operaciones de escritura a esa ubicación. En efecto, la consistencia de procesador asegura que todas las operaciones de escritura realizadas sobre la misma ubicación de memoria (no importa por cuál proceso haya sido realizada) son vistas por todos los procesos en el mismo orden. Este requerimiento es en adicional al requerimiento impuesto por el modelo de consistencia PRAM. Por lo tanto, el ejemplo dado para la consistencia PRAM, si w_{12} y w_{22} son operaciones de escritura para escribir sobre la misma ubicación de memoria x, todos los procesos deben verlas en el mismo orden, w_{12} antes que w_{22} o w_{22} antes que w_{12}. Esto significa que para la consistencia de procesador ambos procesos, P_3 y P_4 deben ver las operaciones de escritura en el mismo orden, el cual puede ser ya sea $[(w_{11}, w_{12}), (w_{21}, w_{22})]$ o $[(w_{21}, w_{22}), (w_{11}, w_{12})]$. Note que, para este ejemplo, la consistencia de procesador y la consistencia secuencial conducen al mismo resultado final, pero esto puede no ser cierto para otros casos.

8.7.5.8 Modelo de orden de almacenamiento total

Otro modelo que permite que las operaciones de lectura pasen por alto a las operaciones de escritura no finalizadas previamente, es decir, que todas las escrituras sean visibles en el orden del programa, es el modelo de orden de almacenamiento total (o *Total Store Order*, en inglés, TSO). La diferencia principal entre este y el modelo de consistencia del procesador es que a este último le falta el atributo de atomicidad en la escritura. «Atomicidad en la escritura»

significa que todos los procesadores tienen exactamente la misma vista que el orden total de las escrituras. El modelo de consistencia del procesador sólo asegura que todas las escrituras provenientes del mismo procesador se ven en el mismo orden que todos los procesadores.

8.7.5.9 Modelo de consistencia débil

El modelo de consistencia débil propuesto por Dubois, Scheurich y Briggs (1988) está diseñado para tomar ventaja de las siguientes dos características, comunes a muchas aplicaciones:

1. No es necesario mostrar el cambio hecho en memoria por cada operación de escritura a los otros procesos. Los resultados de varias operaciones de escritura pueden ser combinados y enviados a otros procesos sólo cuando ellos los necesitan. Por ejemplo, cuando un proceso ejecuta en una sección crítica, se supone que otros procesos no vean los cambios hechos por el proceso a la memoria hasta que el proceso salga de la sección crítica. En este caso, todos los cambios hechos a la memoria por el proceso mientras está en su sección crítica necesitan hacerse visibles a otros procesos sólo en el momento cuando el proceso salga de su sección crítica.
2. Los accesos aislados a variables compartidas son raros. Esto es, en muchas aplicaciones, un proceso hace varios accesos a un conjunto de variables compartidas y luego ningún acceso a estas variables en este conjunto por un largo tiempo.

Ambas características implican que se puede alcanzar un mejor desempeño si la consistencia se impone sobre un grupo de operaciones de referencia a memoria en lugar de hacerlo sobre operaciones de referencia a memoria individuales. Esta es exactamente la idea básica detrás del modelo de consistencia débil.

El problema principal en la implementación de esta idea es la determinación de cómo el sistema puede saber que es tiempo de mostrar los cambios realizados por un proceso a otros procesos, ya que este tiempo es diferente para diferentes aplicaciones. Como no hay manera para el sistema de saber esto por sus medios, se les pide a los programadores que le digan esto al sistema en sus aplicaciones. Para esto, el sistema de memoria compartida distribuida que soporta el modelo de consistencia débil usa una variable especial llamada **variable de sincronización**. Las operaciones sobre ella se usan para sincronizar la memoria. Esto es, cuando una variable de sincronización es accedida por un proceso, toda la memoria (compartida) es sincronizada haciendo todos los cambios a la memoria hechos por todos los procesos visibles a todos los otros procesos. Note que la sincronización de la memoria en un sistema de memoria compartida distribuida involucrará la propagación de actualizaciones de memoria hechas a un nodo a todos los otros nodos que tengan una copia de las mismas direcciones de memoria.

Para soportar la consistencia débil, se deben alcanzar los siguientes requerimientos:

1. Todos los accesos a las varibles de sincronización deben obedecer las semánticas de la consistencia secuencial.
2. Todas las operaciones previas de escritura deben completarse en todas partes antes que sea permitido un acceso a una variable de sincronización.

3. Todos los accesos previos a variables de sincronización deben completarse antes que sea permitido el acceso a una variable de no sincronización.

Note que el modelo de consistencia débil provee mejor desempeño al costo de poner mayores molestias a los programadores.

8.7.5.10 Modelo de consistencia de liberación

Vimos que en el modelo de consistencia débil la memoria (compartida) completa se sincroniza cuando un proceso accede una variable de sincronización, y la sincronización de memoria involucra básicamente las operaciones siguientes:

1. Todos los cambios hechos a la memoria por el proceso se propagan a otros nodos.
2. Todos los cambios hechos a la memoria por otros procesos se propagan desde otros nodos al nodo del proceso.

Una observación más cercana muestra que esto no es realmente necesario porque se necesita realizar la primera operación sólo cuando el proceso sale de una sección crítica y la segunda, sólo cuando el proceso ingresa a la sección crítica. Dado que se usa una única variable de sincronización en el modelo de consistencia débil, el sistema no puede saber que un proceso que está accediendo una variable de sincronización está ingresando a una sección crítica o saliendo de ella. Por lo tanto, tanto la primera como la segunda operación son realizadas sobre cada acceso que un proceso haga a la variable de sincronización. Para un mejor desempeño, el modelo de consistencia de liberación (o *release consistency*, en inglés), propuesto por Gharachorloo y col. (1990), provee un mecanismo para decirle claramente al sistema que un proceso está ingresando a una sección crítica o saliendo de ella de tal manera que el sistema pueda decidir y realizar sólo la primera o segunda operación cuando un proceso está accediendo una variable de sincronización. Esto se logra usando dos variables de sincronización llamadas `acquire` and `release` (*adquirir* y *liberar*, respectivamente) en lugar de una única variable de sincronización. `Acquire` es usada por un proceso para decirle al sistema que está por ingresar a una sección crítica, de manera tal que el sistema realiza sólo la segunda operación cuando esta variable es accedida. Por otra parte, `release` es usada por un proceso para decirle al sistema que acaba de salir de una sección crítica, de manera tal que el sistema realice sólo la primera operación cuando esta variable es accedida. Los programadores son responsables de poner las variables `acquire` y `release` en lugares adecuados en sus programas.

También, se puede realizar la consistencia de liberación al usar el mecanismo de sincronización basado en barreras (o vallas) en lugar de secciones críticas. Una **barrera** define el final de una fase de ejecución de un grupo de procesos que ejecutan concurrentemente. Todos los procesos en el grupo deben completar su ejecución hasta una barrera antes que a cualquier proceso le sea permitido proceder con su ejecución siguiendo la barrera. Esto es, cuando un proceso de un grupo encuentra una barrera durante su ejecución, se bloquea hasta que todos los otros procesos en el grupo completen sus ejecuciones hasta la barrera. Cuando el último proceso completa su ejecución hasta la barrera, todas las variables compartidas se sincronizan y luego todos los procesos continúan con sus ejecuciones. Por lo tanto, `acquire` es la partida

desde una barrera, y `release` es la terminación de la ejecución del último proceso hasta la barrera.

8.7.5.11 Modelo relajado de consistencia de liberación

Una variación del modelo de consistencia de liberación es el modelo relajado de consistencia de liberación (o *lazy release consistency*, en inglés) propuesto por Keleher, A. L. Cox y Zwaenepoel (1992), y es más eficiente que la convencional consistencia de liberación (a la que, para diferenciarla, se la califica como ansiosa o *eaguer*, en inglés). En el enfoque convencional, cuando un proceso hace un acceso `release`, los contenidos de todas las ubicaciones de memoria modificadas en el nodo del proceso se envían a todos los otros nodos que tienen una copia de las mismas ubicaciones de memoria en sus cachés locales. Sin embargo, en el enfoque relajado, las modificaciones se envían a los otros nodos sólo bajo demanda. Esto es, cuando un proceso hace un acceso `acquire`, todas las modificaciones de otros nodos son adquiridas por el nodo del proceso. Por lo tanto, las modificaciones que iban a ser enviadas a este nodo al momento del acceso `release` serán recibidas por el nodo ahora (exactamente cuando se necesitan ahí). La consistencia de liberación relajada tiene mejor desempeño porque en este método no se genera tráfico del todo hasta que un acceso `acquire` se haga por parte de un proceso en algún otro nodo.

TreadMarks es un sistema de memoria compartida distribuida que implementa este protocolo, y está desarrollado en el trabajo de Keleher, A. L. Cox, Dwarkadas y col. (1994).

8.7.5.12 Ordenamiento explícito en memoria

Además de estas normas, podemos utilizar algún tipo de **barrera** , como vimos en 8.7.3.6, que les dicen a los procesadores que ninguna instrucción de carga y/o almacenamiento puede pasar por alto esta instrucción.

P1	P2
while (flag==0);	result=5
LFENCE	SFENCE
print result	flag=1;

Tabla 8.4 – Instrucciones de ordenamiento.

En la tabla 8.4 las operaciones de barrera `LFENCE` y `SFENCE` han sido inferidas para asegurar la salida correcta. La instrucción `LFENCE` en `P1` fuerza al procesador a alcanzar esa instrucción antes de que se realicen más cargas (esto asegura que el lazo haya finalizado antes de la lectura de la variable `result`). `P2` es forzado por la instrucción `SFENCE` para completar todos los almacenamientos antes de esa instrucción y antes de que pueda permitir nuevos almacenamientos (esto asegura que la asignación a `flag` sea hecha luego de la asignación de `result`). En los modelos de consistencia del procesador y orden de almacenamiento total ninguna de las instrucciones de barrera serían necesarias.

El ordenamiento en memoria es un grupo de propiedades de los microprocesadores modernos que caracterizan sus posibilidades en el reordenamiento de operaciones en memoria. Es un tipo de **ejecución fuera de orden** (o *out-of-order execution*, en inglés). El reordenamiento de la memoria se puede usar para utilizar por completo diferentes cachés y bancos de memoria.

En la mayoría de los uniprocesadores modernos las operaciones de memoria no se ejecutan en el orden especificado por el código del programa. Pero en programas con un único hilo desde el punto de vista del programador, todas las operaciones parecen haber sido ejecutadas en el orden especificado, con todas sus inconsistencias ocultadas por el hardware.

La barreras de memoria de un compilador le previenen reordenar instrucciones pero no previenen que la CPU lo haga. Por ejemplo, la instrucción de ensamblador dentro del código en lenguaje C:

```
asm volatile("" ::: "memory");
```

le prohíbe al compilador GCC que reordene los comandos de lectura y escritura alrededor de él.

8.7.5.13 Las barreras de memoria del hardware

Muchas arquitecturas con soporte para SMP tienen instrucciones especiales de hardware para limpiar (*flush*) las lecturas y las escrituras:

```
lfence (asm), void_mm_lfence(void)
sfence (asm), void_mm_sfence(void)
mfence (adm), void_mm_mfence(void)
```

Además, algunos compiladores dan soporte para indicarle al hardware acerca de barreras de memoria en hardware.

8.8 Multiprocesamiento en Windows

Como sabemos, Windows implementa un sistema de planificación con desalojo basado en prioridades. El hilo ejecutable en estado listo con la mayor prioridad es el elegido para ejecutar. En un sistema con un único procesador la planificación es relativamente simple: siempre ejecuta el hilo de mayor prioridad que necesita ejecutar. En un sistema con varios procesadores es más complejo porque el hilo elegido para ejecutar podría quedar limitado por afinidad al procesador sobre el cual se le está permitido ejecutar. Los hilos pueden ejecutar sobre cualquier procesador disponible pero se puede alterar la afinidad mediante llamadas a sistema.

Windows intenta planificar hilos en el procesador óptimo para él tomando en cuenta el procesador preferido y el previo del hilo así como la configuración del sistema multiprocesador. M. E. Russinovich y D. A. Solomon (2004) afirman que, si bien Windows intenta planificar los hilos ejecutables de mayor prioridad sobre todas las CPU disponibles, sólo garantiza que esté ejecutando el (único) hilo de mayor prioridad en alguna parte.

Antes de describir a los algoritmos específicos usados para elegir qué hilos ejecutar dónde y cuándo, examinemos la información adicional que Windows mantiene para registrar el

estado del hilo y el procesador en sistemas con múltiples procesadores y los dos tipos nuevos de sistemas multiprocesador soportados por Windows (hiper hilado y NUMA).

8.8.1 La base de datos del activador

Como se explicó en la sección 4.6.2, la base de datos del activador se refiere a la información mantenida por el núcleo para llevar a cabo la planificación de hilos. Como se muestra en la figura 4.14, en sistemas Windows 2000 y Windows XP para sistemas con múltiples procesadores, la cola de listos y el resumen de listos tienen la misma estructura que la que tienen en sistemas con un único procesador. Además de la cola de listos y el resumen de listos, Windows mantiene dos máscaras de bits que registran el estado de los procesadores en el sistema:

- La máscara del procesador activo (llamada *KeActiveProcessors*, originalmente en inglés) que tiene un conjunto de bits para cada procesador utilizable en el sistema.
- La del resúmen de los inactivos (llamada *KiIdleSummary*, originalmente en inglés) en la que cada bit representa a un procesador ocioso.

En un sistema multiprocesador existe un número de parámetros de configuración –almacenados en la base de datos del activador– para cada hilo. Una máscara de bits define la afinidad del hilo al procesador, es decir, los procesadores sobre los cuales el hilo debe ejecutar; además, cada hilo tiene dos procesadores especiales asociados que están definidos y administrados por el activador: el «procesador ideal», que indica el procesador natal (*home processor*) del hilo en el cual será planificado cada vez que sea posible, y el «último procesador», que indica el procesador en el cual fue ejecutado el hilo por última vez.

Según Vajda (2011), la selección del procesador ideal se hace durante la creación del proceso e hilo, de acuerdo con el siguiente algoritmo:

- En un sistema NUMA: Se selecciona un nodo NUMA ideal para el proceso cuando este es puesto en ejecución. De manera que dos procesos puestos en ejecución uno luego del otro serán asignados a nodos diferentes.
- Cada hilo tendrá un procesador ideal asignado a partir del nodo NUMA ideal, como primera opción. Windows intentará esparcir los hilos pertenecientes al mismo proceso de manera uniforme en todos los procesadores disponibles (el programa tiene la opción de indicar específicamente un procesador ideal deseado).

Las decisiones de planificación en un sistema SMP se reducen en el núcleo de Windows a dos opciones:

1. Para un hilo en condiciones de ejecutar, elegir un procesador para hacerlo.
2. Para un determinado procesador, elegir el próximo hilo para ejecutar.

Cada vez que un hilo está listo para ejecutar, el núcleo Windows que está ejecutando en uno de los núcleos (generalmente donde ocurrió el cambio de estado) intentará elegir un procesador ocioso en el cual ejecutar el hilo. Si hay muchos procesadores ociosos elegibles (por ejemplo dentro del nodo NUMA ideal asociado con el proceso del hilo), se le da preferencia al procesador ideal del hilo, luego al procesador previo del hilo y luego al procesador donde

se está tomando la decisión de planificación. El hilo se inserta en la cola correspondiente y se notifica al procesador para que se pueda iniciar la ejecución.

Si no hay un procesador ocioso, el hilo será agregado a la correspondiente cola del activador en su procesador ideal y a ese procesador se lo notifica de que es necesario un desalojo, en caso de que esté ejecutando un hilo de menor prioridad. Debido a esta política, el planificador de Windows –como decíamos– no garantiza que sean ejecutados todos los hilos que tengan la mayor prioridad si hay un número suficiente de procesadores, sino que garantiza que sea ejecutado al menos uno de los hilos de mayor prioridad. Windows nunca desalojará y moverá un hilo con el fin de permitir que ejecute otro hilo que sólo pudiera hacerlo en ese procesador específico. El hilo será puesto en la cola y tendrá que esperar hasta que el hilo que está ejecutando actualmente deje de ejecutar (por el motivo que fuere).

Un procesador normalmente ejecutará hilos disponibles en sus colas de activación locales, ya sea que hayan sido puestos ahí por el núcleo que está ejecutando en ese procesador o por núcleos en otros procesadores, como se describió anteriormente. Si no hay hilos disponibles el procesador se marca como inactivo y planificará al hilo inactivo del sistema que ya mencionamos en el apartado 4.2. Según M. E. Russinovich y D. A. Solomon (2004) este hilo de Windows no tiene en realidad un nivel de prioridad porque solo ejecuta cuando no hay hilos verdaderos para hacerlo.

8.8.2 Sincronización en el núcleo

En un sistema monoprocesador la base de datos del activador se traba mediante los niveles de solicitudes de interrupción (IRQL, de *interrupt request levels*, por sus siglas en inglés) elevándolo a su máximo valor (el de nivel de activación DPC o *DPC dispatch level*, en inglés) pero esto no es suficiente en un sistema multiprocesador, afirman M. E. Russinovich y D. A. Solomon (2004), porque elevar IRQL en un procesador no previene que ocurra una interrupción en otro procesador, en cuyo caso, todos intentarían operar sobre la base de datos del activador. Entonces se utilizan dos *spinlocks* en el núcleo para sincronizar el acceso: uno para el activador (llamado *KiDispatcherLock*) y otro para el cambio de contexto (*KiContextSwapLock*).

El sistema multiprocesador de Windows tiene una cola de listos para activación por cada procesador, lo que favorece a la escalabilidad del sistema. De esta manera cada CPU puede revisar su propia cola de listos para elegir al próximo hilo sin necesidad de trabar colas globales de todo el sistema.

8.8.3 Gestión de la memoria

El sistema de gestión de la memoria de Windows está estructurado en dos capas principales: el gestor de memoria virtual, responsable de la asignación de grano grueso y gestión de las páginas de memoria virtual, y el gestor del montículo (*heap manager*, en inglés), responsable de la asignación de grano fino de los *buffers* de memoria.

En el nivel de memoria virtual, Windows soporta dos tipos de página: las pequeñas (entre 4 y 8 kb, dependiendo del hardware) y las grandes (entre 4 y 16 Mb, también dependiendo del hardware), pero la granularidad de la asignación es siempre de 64 kb como mínimo. El gestor de memoria soporta prioridades en las páginas (para poder llevar adelante la política

de qué páginas salen intercambiadas si se necesita más memoria física) y también la traba explícita de páginas a la memoria física, con el fin de mejorar el rendimiento de aplicaciones críticas. A nivel de núcleo se logra un efecto similar al tener dos tipos de reservorios de memoria (o *memory pools*, en inglés): los no paginados (o *non-paged pool*), con todas las páginas residentes en memoria todo el tiempo, y los paginados, que serán gestionados como cualquier otro reservorio de páginas virtuales.

Windows también provee una variante del «asignador de baldosas» (o *slab allocator*) visto en la sección 3.3.2.3.1 al que llama «listas de búsqueda de reojo» (LAL, de *look-aside lists*, por sus siglas en inglés). Cada lista contiene objetos de memoria del mismo tamaño que pueden asignarse rápidamente toda vez que se necesiten y la entrega de la memoria se logra mediante la asignación de nuevos conjuntos de páginas cuando se necesiten, de manera similar al enfoque basado en *slabs* de SunOS.

La diferencia básica entre los reservorios y las LAL es que las asignaciones de los reservorios pueden variar de tamaño mientras que una lista contiene sólo bloques de tamaño fijo. Si bien los reservorios son más flexibles en términos de qué pueden suministrar, las LAL son más rápidas porque no usan ningún *spinlock* y porque el sistema no tiene que buscar memoria libre en la que quepa una asignación de tamaño variable.

Vajda (2011) destaca que el administrador de montículo (*heap manager*) de Windows actúa como una capa sobre el administrador de memoria virtual y provee soporte para asignación de memoria a un nivel de granularidad de byte; obviamente hay un administrador de montículo por cada uno de ellos y puede haber varios creados por cada proceso. La implementación predeterminada del administrador de montículo de Windows soporta múltiples hilos, aunque utiliza un esquema de trabas global que limita su escalabilidad en caso de que haya un gran número de hilos distribuidos entre varios procesadores. Para mitigar este problema hay una fachada (*front end*, en inglés) especial de administrador de montículo junto con el administrador predeterminado llamada el «montículo de baja fragmentación» (LFH, de *Low Fragmentation Heap*, por sus siglas en inglés). El LFH organiza a los *buffers* en cubos (*buckets*) de diferentes tamaños y asigna memoria a partir del cubo con mejor ajuste. Para mejorar la escalabilidad, el LFH tiene un método especial de ubicar estructuras internas, dividiendo a los cubos en «zócalos» (o *slots* en inglés) – hay el doble de zócalos que de procesadores. Los hilos son asignados a un zócalo mediante un componente interno llamado el administrador de afinidad y puede moverse a otro zócalo (usando un conjunto diferente de estructuras de datos) en el caso de que se detecte una contienda por los recursos.

Al permitir distintos tipos de de asignadores, las aplicaciones con diferentes necesidades de memoria pueden elegir el asignador apropiado. Por ejemplo, las aplicaciones que esperan pequeñas ráfagas de asignaciones pueden preferir usar el LFH para evitar la fragmentación.

Para dar soporte a los sistemas NUMA, el administrador de memoria de Windows crea durante el arranque un grafo de costos que describe el costo de acceder la memoria asignada en diferentes nodos desde todos los otros. Cada vez que una página necesita ser asignada o intercambiada a la memoria física, el administrador de memoria se asegurará de que se elija el nodo con el menor costo de acceso, basado en la información almacenada en el grafo de costos. Como principio básico, la memoria necesitada por un hilo es priorizada siempre desde el procesador ideal, tanto en el caso de la asignación como en el de fallo, aun si el hilo está en ese punto ejecutando sobre otro procesador, el mecanismo de planificación descripto se asegura de que el hilo va a terminar en su procesador ideal y así se beneficiará de tener su

memoria paginada en la RAM de ese procesador. Sólo si el nodo ideal carece de recursos será elegido otro nodo, siempre basándose en el grafo de costos.

La asignación de montículos y de LAL, incluyendo los reservorios no paginados del núcleo, también se difunde a través de los nodos NUMA para mejorar el tiempo de acceso de los núcleos que ejecutan en procesadores diferentes. La asignación a partir de estos reservorios también asegurará de que sea elegida la dirección virtual mapeada más cercana al núcleo correspondiente.

8.9 Multiprocesamiento en Linux

El núcleo Linux ha tenido soporte para SMP desde la versión 2.0, pero ha estado siempre sometido a modificaciones en su diseño. Tal vez el principal desafío que tuvo que superar fue la **gran traba en el núcleo** (BKL, de *Big Kernel Lock*, por sus siglas en inglés) para poder ofrecer un control en la concurrencia cada vez que un hilo ingresaba a espacio de núcleo. Los hilos en espacio de usuario podían ejecutar concurrentemente en cualquier procesador disponible pero no más de uno podía hacerlo en espacio de núcleo. Cualquier otro hilo que intentara hacerlo debía esperar, es decir que se eliminaba la concurrencia en espacio de núcleo. Esta situación permaneció así hasta la versión 2.6.39, en la que fue dejado totalmente de lado este método de serialización por otro con una granularidad mucho más fina, en particular en los *spinlocks*, *mutex* y RCU.

La planificación en Linux está estructurada en dos niveles: el planificador central y las clases del planificador. La división de tareas entre las dos capas está bastante bien definida: mientras que la capa de clases del planificador toma las decisiones de planificación de acuerdo con distintas políticas, decidiendo –por lo tanto– qué tarea ejecutar a continuación, el planificador central se encarga de la gestión de tareas generales y las actividades de intercambio de tareas, independientemente de cómo (de acuerdo con qué política) haya sido seleccionada la próxima tarea que va a ser ejecutada.

La planificación en arquitecturas SMP en Linux es una extensión de la planificación con un único procesador. Cada CPU tiene su propio planificador pero va acompañado de un balanceo periódico entre CPU. En cada invocación (*tick*) periódica del planificador, y en cada CPU, se comprueba si es necesario otro balanceo. En la práctica, esto significa que si ha transcurrido suficiente tiempo desde el último balanceo, se inicia un nuevo procedimiento para volver a balancear.

Este nuevo balanceo se hace por clase de planificación y siempre dentro del dominio de planificación. Este es un conjunto de CPU que definen un dominio dentro del cual se puede llevar a cabo una replanificación (por ejemplo, compartir la misma tarjeta, zócalo de procesador o dominio NUMA). En una arquitectura SMP perfectamente plana habrá, naturalmente, sólo un dominio de planificación.

El rebalanceo se basa en un mecanismo de robo de la tarea. Cuando se ha decidido un rebalanceo, la CPU ladrona identificará a la CPU con la cola de ejecución más ocupada y si la carga en esa CPU es mayor intentará mover tareas desde esa CPU hacia sí misma (sólo pueden moverse tareas que no están ejecutando en ese momento). En el caso de que no se pueda hacer el movimiento de tareas por alguna razón, la CPU con la cola más ocupada será notificada para que realice ella misma la descarga de alguna de sus tareas. Esto será llevado a cabo por un hilo especial llamado **hilo de migración** (*migration thread*) asociado con cada

CPU y el procedimiento se llama balanceo activo; las tareas serán movidas por la fuerza si es necesario.

La naturaleza distribuida de este método de rebalanceo es su fortaleza y su debilidad. Como no es una actividad sincronizada puede trabajar de modo autónomo, a través de varios dominios de planificación. Por la misma razón, no obstante, puede ser la fuente de competencia entre CPU porque varias pueden intentar mover tareas desde la misma CPU ocupada. La cola de ejecución de cualquier CPU puede ser inspeccionada en cualquier momento por cualquier otra CPU y los intentos por modificarla pueden ocurrir en cualquier momento y de forma concurrente con decisiones de planificación locales. Si bien cuidadosos esquemas de trabe pueden hacer que el mecanismo trabaje sin rispideces para un número razonable de CPU, puede volverse impráctico para sistemas SMP a gran escala, donde a menudo está o bien deshabilitado o confinado a dominios de planificación de tamaño razonable.

8.9.1 Gestión de la memoria

La gestión de la memoria en Linux está basada en el sistema de compañeras para la gestión de la memoria de propósito general y el asignador de baldosas para la asignación de memoria específica del núcleo. La memoria física está dividida, usualmente en una tasa de tres a uno entre las aplicaciones y el núcleo.

Desde el punto de vista del soporte del procesador para varios procesadores y multinúcleos, Linux ofrece soporte para sistemas NUMA y migración de páginas de memoria. Para sistemas multiprocesadores en general y para arquitecturas NUMA en particular, la memoria se organiza en torno a los conceptos de **nodo** y **conjuntos de CPU** (*CPU sets*). Cada nodo posee parte de la memoria física y, en general, corresponde a una CPU (o varios núcleos de procesador que comparten parte de la memoria). Un conjunto de CPU agrupa a varias CPU (nodos) en un grupo que tiene acceso uniforme a parte de la memoria.

Las aplicaciones pueden asignarse a ciertos conjuntos de CPU y el sistema operativo soporta migración automática de páginas de memoria en caso de que la aplicación sea reasignada. El mismo efecto puede lograrse definiendo políticas de memoria que fuerzan la asignación de memoria para una aplicación específica de un conjunto específico de nodos. El gestor de políticas de memoria puede ser configurado para llevar a cabo la migración automática de páginas redefiniendo el conjunto de nodos en los cuales pueden ser asignadas las páginas para esa aplicación específica, acompañado por un pedido explícito de cumplir con la nueva política (lo cual, en la práctica, significa una migración de página).

Además de estos dos métodos (asignación a conjuntos de CPU y política de gestión de memoria), el núcleo Linux no soporta otras políticas de migración de página, ya que el principio fundamental que guía el diseño de la gestión de memoria ha sido siempre el de dejar el control de esta a la aplicación, por falta de un buen método para predecir la localización del mejor ajuste de página de memoria.