

# SISTEMAS OPERATIVOS

## CONCEPTOS FUNDAMENTALES

---

ANTONIA ESTERO BOTARO  
JUAN JOSÉ DOMÍNGUEZ JIMÉNEZ





---

# Sistemas Operativos

## Conceptos fundamentales

---

Antonia Estero Botaro

Juan José Domínguez Jiménez

Dpto. Lenguajes y Sistemas Informáticos

*Universidad de Cádiz*



Universidad  
de Cádiz

Servicio de Publicaciones

Estero Botaro, Antonia

Sistemas operativos : conceptos fundamentales / Antonia  
Estero Botaro, Juan José Domínguez Jiménez. -- Cádiz : Servicio  
de Publicaciones de la Universidad, 2001. --

ISBN 978-84-7786-716-6

I. Sistemas operativos -Tratados, manuales, etc. I. Domínguez  
Jiménez, Juan José. II. Universidad de Cádiz. Servicio de  
Publicaciones, ed. II. Título.

681.3.066

1<sup>a</sup> edición: noviembre 2005

2<sup>a</sup> reimpresión: marzo 2009

3<sup>a</sup> reimpresión: noviembre 2009

Edita: Servicio de Publicaciones de la Universidad de Cádiz.

© Antonia Estero Botaro

Juan José Domínguez Jiménez

Diseño de Cubierta: CREASUR

I.S.B.N.: 978-84-7786-716-6

Depósito Legal: SE-1844-2009

Imprime: Publidisa

*A Patricia, Gerardo y mis padres*

Antonia

*A mis padres*

Juan José



# Índice General

---

<b>1 INTRODUCCIÓN</b>	<b>1</b>
<b>1 Introducción a los Sistemas Operativos</b>	<b>3</b>
1.1 ¿Qué es un sistema operativo? . . . . .	3
1.2 Evolución histórica de los sistemas operativos . . . . .	6
1.2.1 Los primeros sistemas . . . . .	6
1.2.2 Sistemas por lotes . . . . .	7
1.2.3 Sistemas por lotes multiprogramados . . . . .	15
1.2.4 Sistemas de tiempo compartido . . . . .	17
1.3 Sistemas para computadores personales . . . . .	18
1.4 Sistemas de tiempo real . . . . .	18
1.5 Sistemas con más de un procesador . . . . .	19
1.5.1 Sistemas paralelos . . . . .	19
1.5.2 Sistemas en red y distribuidos . . . . .	20
1.6 Mecanismos hardware de protección . . . . .	21
1.7 Resumen . . . . .	22
1.8 Ejercicios . . . . .	24
<b>2 Estructura y funciones de los sistemas operativos</b>	<b>27</b>

2.1	Funciones y componentes de los sistemas operativos . . . . .	27
2.2	Solicitud de servicios . . . . .	30
2.2.1	Llamadas al sistema . . . . .	30
2.2.2	Paso de mensajes . . . . .	31
2.3	Estructura de un sistema operativo . . . . .	32
2.3.1	Estructura simple o monolítica . . . . .	34
2.3.2	Estructura modular . . . . .	37
2.3.3	Máquinas virtuales . . . . .	40
2.3.4	Micronúcleo . . . . .	41
2.4	El sistema operativo LINUX . . . . .	42
2.4.1	Componentes del núcleo . . . . .	43
2.5	Resumen . . . . .	46
2.6	Ejercicios . . . . .	46
<b>2</b>	<b>PROCESOS</b>	<b>47</b>
<b>3</b>	<b>Descripción y control de procesos</b>	<b>49</b>
3.1	¿Qué es un proceso? . . . . .	49
3.2	Vida de un proceso . . . . .	51
3.2.1	Creación de un proceso . . . . .	51
3.2.2	Terminación de un proceso . . . . .	52
3.2.3	Estados de un proceso . . . . .	53
3.3	Imagen de un proceso . . . . .	59
3.3.1	Bloque de control del proceso . . . . .	61
3.4	Control de los procesos . . . . .	63
3.5	Gestión de procesos . . . . .	66
3.5.1	Creación de procesos . . . . .	66
3.5.2	Cambio de proceso . . . . .	67

<b>Índice General</b>	<b>iii</b>
3.6 Hilos de ejecución . . . . .	70
3.7 Procesos en LINUX . . . . .	73
3.7.1 Imagen de un proceso . . . . .	73
3.7.2 El bloque de control de procesos . . . . .	74
3.7.3 La tabla de procesos . . . . .	75
3.7.4 Estados de un proceso . . . . .	75
3.7.5 Llamadas al sistema . . . . .	77
3.7.6 Hilos . . . . .	78
3.8 Resumen . . . . .	83
3.9 Ejercicios . . . . .	83
<b>4 Planificación</b>	<b>85</b>
4.1 Introducción . . . . .	85
4.2 Niveles de planificación . . . . .	86
4.2.1 Planificación a largo plazo . . . . .	87
4.2.2 Planificación a medio plazo . . . . .	88
4.2.3 Planificación a corto plazo . . . . .	88
4.3 Algoritmos de planificación . . . . .	89
4.3.1 Primero en llegar, primero en ser servido . . . . .	89
4.3.2 El proceso más corto primero . . . . .	91
4.3.3 Tiempo restante más corto . . . . .	92
4.3.4 Tasa de respuesta más alta . . . . .	93
4.3.5 Asignación por turnos . . . . .	94
4.3.6 Prioridades . . . . .	96
4.3.7 Planificación en varios niveles . . . . .	98
4.3.8 Planificación en varios niveles con realimentación . . . . .	99
4.4 Evaluación de algoritmos de planificación . . . . .	102
4.4.1 Criterios del planificador a corto plazo . . . . .	102

4.4.2	Evaluación analítica . . . . .	104
4.4.3	Simulación . . . . .	105
4.4.4	Implementación . . . . .	105
4.5	Planificación de hilos . . . . .	105
4.6	Planificación en LINUX . . . . .	106
4.6.1	Planificación en sistemas multiprocesadores . . . . .	108
4.7	Resumen . . . . .	109
4.8	Ejercicios . . . . .	109
<b>3</b>	<b>CONCURRENCIA</b>	<b>113</b>
<b>5</b>	<b>Sincronización y comunicación</b>	<b>115</b>
5.1	Introducción . . . . .	115
5.2	Recursos . . . . .	117
5.3	Interacción entre procesos . . . . .	119
5.3.1	Competencia . . . . .	119
5.3.2	Compartición . . . . .	120
5.3.3	Comunicación . . . . .	121
5.4	Exclusión mutua . . . . .	122
5.4.1	Soluciones software . . . . .	122
5.4.2	Soluciones con ayuda del hardware . . . . .	133
5.5	Semáforos . . . . .	137
5.5.1	Exclusión mutua . . . . .	140
5.5.2	El problema del productor/consumidor . . . . .	142
5.6	Monitores . . . . .	144
5.6.1	El problema del productor/consumidor . . . . .	147
5.7	Señales . . . . .	149
5.8	Paso de mensajes . . . . .	149

5.8.1	Características del direccionamiento . . . . .	150
5.8.2	Sincronización de las primitivas . . . . .	150
5.8.3	Exclusión mutua . . . . .	152
5.9	Mecanismos de concurrencia en LINUX . . . . .	153
5.9.1	Semáforos . . . . .	153
5.9.2	Señales . . . . .	154
5.9.3	Interconexiones en un sentido . . . . .	155
5.9.4	Interconexiones FIFO . . . . .	156
5.9.5	Colas de mensajes . . . . .	156
5.10	Resumen . . . . .	157
5.11	Ejercicios . . . . .	158
<b>6</b>	<b>Interbloqueos</b>	<b>161</b>
6.1	Introducción . . . . .	161
6.2	Condiciones necesarias . . . . .	162
6.3	Modelado del interbloqueo . . . . .	163
6.4	Estrategias para tratar los interbloqueos . . . . .	165
6.4.1	Prevención . . . . .	165
6.4.2	Predicción . . . . .	169
6.4.3	Detección y recuperación . . . . .	174
6.5	Resumen . . . . .	179
6.6	Ejercicios . . . . .	180
<b>4</b>	<b>MEMORIA</b>	<b>183</b>
<b>7</b>	<b>Administración de la memoria</b>	<b>185</b>
7.1	Jerarquía del almacenamiento . . . . .	185
7.2	Traducción de direcciones . . . . .	187

7.3	Funciones del administrador de la memoria . . . . .	190
7.4	Esquemas de asignación de la memoria . . . . .	191
7.5	Sistemas de monoprogramación . . . . .	192
7.6	Multiprogramación con particiones fijas . . . . .	193
7.6.1	Selección del tamaño de las particiones . . . . .	193
7.6.2	Algoritmos de colocación . . . . .	194
7.6.3	Elementos de control . . . . .	196
7.6.4	Protección . . . . .	196
7.6.5	Inconvenientes . . . . .	197
7.7	Multiprogramación con particiones variables . . . . .	197
7.7.1	Compactación . . . . .	200
7.7.2	Algoritmos de colocación . . . . .	201
7.7.3	Elementos de control . . . . .	203
7.8	El sistema compañero . . . . .	206
7.9	Paginación . . . . .	208
7.9.1	Tabla de marcos . . . . .	210
7.9.2	Tabla de páginas . . . . .	211
7.9.3	Traducción de direcciones . . . . .	211
7.9.4	Protección . . . . .	212
7.9.5	Páginas compartidas . . . . .	215
7.9.6	Fragmentación . . . . .	216
7.9.7	Tamaño de las páginas . . . . .	216
7.10	Segmentación . . . . .	217
7.10.1	Tabla de segmentos . . . . .	217
7.10.2	Traducción de direcciones . . . . .	219
7.10.3	Compartición y protección . . . . .	219
7.10.4	Fragmentación y algoritmos de colocación . . . . .	222

7.11 Segmentación paginada . . . . .	222
7.12 Resumen . . . . .	224
7.13 Ejercicios . . . . .	224
<b>8 Memoria virtual</b>	<b>229</b>
8.1 Introducción . . . . .	229
8.2 El principio de localidad . . . . .	230
8.3 Principios de operación . . . . .	231
8.3.1 El área de intercambio . . . . .	234
8.4 Estructuras hardware y de control . . . . .	235
8.4.1 Tabla de marcos . . . . .	235
8.4.2 Tabla de páginas . . . . .	236
8.5 Funciones del gestor de memoria virtual . . . . .	243
8.5.1 Política de lectura . . . . .	244
8.5.2 Política de colocación . . . . .	245
8.5.3 Política de sustitución . . . . .	245
8.5.4 Gestión del conjunto residente . . . . .	256
8.5.5 Política de limpieza . . . . .	266
8.5.6 Control de la carga y la hiperpaginación . . . . .	267
8.6 Gestión de memoria en LINUX . . . . .	269
8.6.1 Regiones de memoria virtual . . . . .	270
8.6.2 Traducción de direcciones . . . . .	270
8.6.3 Estructura de la tabla de páginas . . . . .	271
8.6.4 Estructura de la tabla de marcos . . . . .	273
8.6.5 Gestión del espacio libre . . . . .	274
8.6.6 Fallos de página . . . . .	275
8.6.7 El demonio de paginación . . . . .	276
8.6.8 Gestión del área de intercambio . . . . .	277

8.6.9	Llamadas al sistema . . . . .	277
8.7	Resumen . . . . .	278
8.8	Ejercicios . . . . .	279
<b>5</b>	<b>ENTRADA/SALIDA</b>	<b>283</b>
<b>9</b>	<b>Gestión de dispositivos</b>	<b>285</b>
9.1	Introducción . . . . .	285
9.2	Dispositivos de E/S . . . . .	286
9.2.1	Controladoras de dispositivos . . . . .	287
9.3	Organización del sistema de E/S . . . . .	289
9.4	Modos de realizar las operaciones de E/S . . . . .	293
9.4.1	E/S controlada por programa . . . . .	293
9.4.2	E/S controlada por interrupciones . . . . .	295
9.4.3	Acceso directo a memoria . . . . .	296
9.5	Optimización de las operaciones de E/S . . . . .	299
9.5.1	Técnicas de <i>buffering</i> . . . . .	299
9.5.2	Caché de disco . . . . .	301
9.5.3	Planificación de discos . . . . .	302
9.6	E/S en LINUX . . . . .	310
9.6.1	Técnicas de optimización . . . . .	310
9.6.2	Llamadas al sistema . . . . .	311
9.7	Resumen . . . . .	312
9.8	Ejercicios . . . . .	313
<b>10</b>	<b>Sistemas de ficheros</b>	<b>315</b>
10.1	Introducción . . . . .	315
10.2	Funciones del sistema de ficheros . . . . .	316

---

10.3 Interfaz del sistema de ficheros . . . . .	318
10.3.1 Ficheros . . . . .	319
10.3.2 Directorios . . . . .	321
10.4 Diseño del sistema de ficheros . . . . .	324
10.4.1 Agrupamiento de registros . . . . .	325
10.4.2 Métodos de asignación de espacio a los ficheros . . . . .	326
10.4.3 Gestión del espacio libre . . . . .	333
10.4.4 Implementación de directorios . . . . .	335
10.5 Fiabilidad del sistema de ficheros . . . . .	336
10.5.1 Copias de seguridad . . . . .	337
10.5.2 Consistencia del sistema de ficheros . . . . .	337
10.6 Rendimiento del sistema de ficheros . . . . .	337
10.7 Sistema de ficheros en LINUX . . . . .	338
10.7.1 Estructura lógica . . . . .	338
10.7.2 Estructura física del fichero . . . . .	341
10.7.3 Estructura física de un sistema de ficheros . . . . .	343
10.7.4 Directorios . . . . .	347
10.7.5 Enlaces . . . . .	348
10.7.6 Búsqueda de un fichero . . . . .	350
10.7.7 Optimización del sistema de ficheros . . . . .	352
10.7.8 Consistencia del sistema de ficheros . . . . .	353
10.7.9 Llamadas al sistema . . . . .	355
10.8 Resumen . . . . .	355
10.9 Ejercicios . . . . .	356
<b>APÉNDICES</b>	<b>361</b>
<b>A Conceptos básicos sobre el hardware</b>	<b>363</b>

---

A.1 Elementos básicos del computador . . . . .	363
A.2 Registros del procesador . . . . .	365
A.2.1 Registros visibles al usuario . . . . .	365
A.2.2 Registros de control y estado . . . . .	366
A.3 Ejecución de una instrucción . . . . .	367
A.4 Interrupciones . . . . .	371
A.4.1 Procesamiento de la interrupción . . . . .	372
A.5 Buses . . . . .	374
<b>B Prefijos para los múltiplos binarios</b>	<b>377</b>
<b>Bibliografía</b>	<b>379</b>
<b>Índice</b>	<b>385</b>

# Índice de Figuras

---

1.1	Estructura de un sistema de computación . . . . .	4
1.2	Estructura de la memoria con monitor residente . . . . .	9
1.3	Conjunto de tarjetas de un trabajo . . . . .	10
1.4	Modos de operación . . . . .	12
1.5	<i>Buffering</i> . . . . .	13
1.6	<i>Spooling</i> . . . . .	14
1.7	Estado de la memoria en un sistema de multiprogramación . . . .	15
1.8	Ejecución de trabajos en un sistema multiprogramado . . . .	16
1.9	Evolución de los sistemas operativos . . . . .	23
2.1	Funciones del sistema operativo . . . . .	29
2.2	Funcionamiento de una llamada al sistema . . . . .	31
2.3	Funcionamiento del paso de mensajes . . . . .	32
2.4	Clasificación de los sistemas operativos según su estructura . . . .	33
2.5	Estructura del sistema UNIX . . . . .	35
2.6	Estructura de MS-DOS . . . . .	36
2.7	Capa de un sistema operativo . . . . .	38
2.8	Estructura en capas del sistema THE . . . . .	38
2.9	Estructura en capas de OS/2 . . . . .	39

2.10 Sistema operativo tradicional frente a máquina virtual . . . . .	41
2.11 Estructura de micronúcleo . . . . .	42
2.12 Diagrama del núcleo de un sistema LINUX . . . . .	44
3.1 Conversión de un programa fuente en proceso . . . . .	50
3.2 Modelo de procesos de cinco estados . . . . .	55
3.3 Modelo de colas para el diagrama de la figura 3.2 . . . . .	57
3.4 Diagrama de transición de estados con estados suspendidos . . . . .	59
3.5 Imagen de un proceso . . . . .	60
3.6 Implementación de la tabla de procesos . . . . .	64
3.7 Cambios que se producen al atender una interrupción . . . . .	69
3.8 Proceso pesado frente a proceso multihilo . . . . .	72
3.9 Diagrama de estados de un proceso en LINUX . . . . .	76
3.10 Sincronización entre proceso padre e hijo . . . . .	78
3.11 Relación entre el proceso y el hilo . . . . .	79
3.12 Ejemplo de creación de hilos (paso 1) . . . . .	80
3.13 Ejemplo de creación de hilos (paso 2) . . . . .	80
3.14 Ejemplo de creación de hilos (paso 3) . . . . .	81
3.15 Ejemplo de creación de hilos (paso 4) . . . . .	81
3.16 Ejemplo de creación de hilos (paso 5) . . . . .	82
4.1 Intercalamiento en la ejecución de varios procesos . . . . .	86
4.2 Planificación y transiciones de estado de los procesos . . . . .	87
4.3 Algoritmo FIFO . . . . .	90
4.4 Algoritmo SPN . . . . .	91
4.5 Algoritmo SRT . . . . .	93
4.6 Algoritmo HRRN . . . . .	94
4.7 Algoritmo RR con cuanto de 2 unidades . . . . .	95

4.8 Algoritmo RR con cuanto de 4 unidades . . . . .	95
4.9 Algoritmo por prioridades no apropiativo . . . . .	97
4.10 Algoritmo por prioridades apropiativo . . . . .	97
4.11 Planificación en varios niveles . . . . .	98
4.12 Planificación en varios niveles con realimentación . . . . .	99
4.13 Planificación en 3 niveles con realimentación para $q=1$ . . . . .	100
4.14 Planificación en 3 niveles con realimentación para $q=2^n$ . . . . .	101
4.15 Instante $t = 8$ para el ejemplo de planificación en 3 niveles con realimentación para $q=2^n$ . . . . .	101
5.1 Trazas de ejecución de un conjunto de procesos . . . . .	116
5.2 Estructura de un monitor . . . . .	146
5.3 Comunicación directa e indirecta entre procesos . . . . .	151
5.4 Representación gráfica de una interconexión . . . . .	155
6.1 Notación para el modelado de interbloqueos . . . . .	163
6.2 Grafo de asignación de recursos con interbloqueo . . . . .	164
6.3 Grafo de asignación de recursos con ciclo pero sin interbloqueo . . . . .	164
6.4 Relaciones entre estado seguro, inseguro e interbloqueo . . . . .	171
6.5 Ejemplo de reducción de un grafo . . . . .	176
7.1 Jerarquía del almacenamiento . . . . .	186
7.2 Tipos de direccionamientos . . . . .	188
7.3 Soporte hardware para la traducción de direcciones relativas . . . . .	190
7.4 Organización de la memoria en monoprogramación . . . . .	193
7.5 Protección de la memoria en sistemas de monoprogramación . . . . .	194
7.6 Diseño de particiones fijas en una memoria de 4 MiB . . . . .	195
7.7 Alternativas de planificación en multiprogramación con particiones fijas . . . . .	196

---

7.8 Tabla de particiones para un sistema de particiones fijas . . . . .	197
7.9 Esquemas de protección en sistemas con particiones fijas . . . . .	198
7.10 Asignación de la memoria con particiones variables . . . . .	199
7.11 Compactación en un sistema con particiones variables . . . . .	200
7.12 Ejemplo de uso de diversos algoritmos de colocación . . . . .	202
7.13 Mapa de bits de una zona de memoria con 4 procesos . . . . .	204
7.14 Lista enlazada de una zona de memoria con 4 procesos . . . . .	204
7.15 Combinaciones posibles cuando un proceso sale de la memoria . .	205
7.16 Sistema compañero . . . . .	207
7.17 Memoria lógica y física en el esquema de paginación . . . . .	209
7.18 Carga de un proceso en un sistema de paginación . . . . .	210
7.19 Traducción de direcciones en un sistema de paginación . . . . .	213
7.20 Traducción de direcciones en paginación con TLB . . . . .	214
7.21 Esquema del sistema segmentado . . . . .	218
7.22 Traducción de direcciones en un sistema de segmentación . . . . .	220
7.23 Traducción de direcciones en un sistema segmentado paginado . .	223
8.1 Pasos realizados durante un fallo de página . . . . .	232
8.2 Traducción en un sistema de memoria virtual con TLB . . . . .	237
8.3 Tabla de páginas de un nivel y de dos niveles . . . . .	239
8.4 Tabla de páginas de dos niveles . . . . .	240
8.5 Tabla de páginas invertida . . . . .	242
8.6 Relación entre el nº de marcos y la tasa de fallos de página . . .	246
8.7 Relación entre el tamaño de la página y la tasa de fallos de página	247
8.8 Funcionamiento del almacenamiento intermedio de páginas . . . .	255
8.9 Conjuntos de trabajo de un proceso para distintos $\Delta$ . . . . .	259
8.10 Evolución del conjunto de trabajo con el tiempo . . . . .	260
8.11 Comportamiento de la frecuencia de fallos de página . . . . .	265

8.12 Frecuencia de fallos de página . . . . .	266
8.13 Hiperpaginación . . . . .	267
8.14 Traducción de direcciones en LINUX . . . . .	271
8.15 Entrada de la tabla de páginas en Intel x86 . . . . .	272
8.16 Entrada de la tabla de páginas en Alpha AXP . . . . .	273
8.17 Estructura de la lista de marcos libres . . . . .	274
9.1 Estructura de una controladora de E/S . . . . .	288
9.2 Direccionamiento de los registros de las controladoras . . . . .	290
9.3 Estructura del sistema de E/S . . . . .	291
9.4 Manejadores de dispositivos reconfigurables . . . . .	292
9.5 E/S controlada por programa . . . . .	294
9.6 E/S controlada por interrupciones . . . . .	297
9.7 Operación de entrada con <i>buffer</i> . . . . .	299
9.8 Doble <i>buffer</i> . . . . .	300
9.9 <i>Buffer</i> circular . . . . .	301
9.10 Componentes de un disco . . . . .	303
9.11 Algoritmo FIFO . . . . .	305
9.12 Algoritmo SSTF . . . . .	306
9.13 Algoritmo SCAN . . . . .	307
9.14 Algoritmo LOOK . . . . .	308
9.15 Algoritmo C-SCAN . . . . .	309
9.16 Algoritmo C-LOOK . . . . .	310
10.1 Relación entre el sistema de ficheros y el de E/S . . . . .	317
10.2 Directorio con estructura de árbol . . . . .	322
10.3 Directorio con estructura de grafo acíclico . . . . .	323
10.4 Directorio con estructura de grafo general . . . . .	324

10.5 Efecto del tamaño de bloque . . . . .	325
10.6 Métodos de agrupamiento de registros . . . . .	327
10.7 Asignación contigua . . . . .	328
10.8 Asignación enlazada . . . . .	329
10.9 Tabla de asignación de ficheros . . . . .	331
10.10 Asignación indexada . . . . .	332
10.11 Mapa de bits para la gestión del espacio libre . . . . .	334
10.12 Lista de espacio libre enlazada . . . . .	334
10.13 Entrada de un directorio en MS-DOS . . . . .	336
10.14 Optimización del rendimiento en asignación indexada . . . . .	339
10.15 Jerarquía de directorios y ficheros . . . . .	340
10.16 Sistema de ficheros constituido por tres dispositivos . . . . .	341
10.17 Estructura del nodo índice del sistema de ficheros <b>ext2</b> . . . . .	342
10.18 Estructura física del sistema de ficheros <b>ext2</b> . . . . .	343
10.19 Estructura de un grupo de bloques . . . . .	344
10.20 Entrada de un directorio del sistema de ficheros <b>ext2</b> . . . . .	348
10.21 Enlaces duros y simbólicos en el sistema de ficheros <b>ext2</b> . . . . .	349
10.22 Las etapas en la búsqueda de <code>/home/linux/ext2</code> . . . . .	351
10.23 Preasignación de bloques . . . . .	352
10.24 Estados de consistencia de un sistema de ficheros <b>ext2</b> . . . . .	354
A.1 Visión a alto nivel de un computador . . . . .	364
A.2 Ciclo de instrucción básico . . . . .	368
A.3 Características de una máquina hipotética . . . . .	369
A.4 Ejemplo de ejecución de un programa . . . . .	370
A.5 Ciclo de instrucción con interrupciones . . . . .	372
A.6 Procesamiento de una interrupción . . . . .	374
A.7 Módulos del computador . . . . .	375

# Índice de Tablas

---

3.1	Elementos típicos de un bloque de control de un proceso . . . . .	62
3.2	Sucesos que interrumpen la ejecución de un proceso . . . . .	68
3.3	Llamadas al sistema relacionadas con procesos . . . . .	77
3.4	Algunas funciones de LINUX relacionadas con los hilos . . . . .	78
4.1	Conjunto de procesos . . . . .	89
4.2	Resultados para el algoritmo FIFO . . . . .	90
4.3	Resultados para el algoritmo SPN . . . . .	91
4.4	Resultados para el algoritmo SRT . . . . .	93
4.5	Resultados para el algoritmo HRRN . . . . .	94
4.6	Resultados para el algoritmo RR con un cuanto de 2 unidades . .	96
4.7	Resultados para el algoritmo RR un cuanto de 4 unidades . . .	96
4.8	Resultados para el algoritmo de prioridades no apropiativo . . .	97
4.9	Resultados para el algoritmo de prioridades apropiativo . . . .	97
4.10	Resultados para 3 niveles con realimentación ( $q=1$ ) . . . . .	100
4.11	Resultados para 3 niveles con realimentación ( $q=2^n$ ) . . . . .	101
4.12	Criterios de planificación orientados al usuario . . . . .	103
4.13	Criterios de planificación orientados al sistema . . . . .	103
4.14	Comparación de resultados de distintos algoritmos de planificación	104

5.1 Llamadas al sistema relacionadas con semáforos . . . . .	153
5.2 Llamadas al sistema relacionadas con señales . . . . .	154
5.3 Llamadas al sistema relacionadas con interconexiones . . . . .	156
5.4 Llamadas al sistema relacionadas con mensajes . . . . .	157
8.1 Funciones del gestor de memoria virtual . . . . .	244
8.2 Comportamiento del algoritmo óptimo . . . . .	248
8.3 Comportamiento del algoritmo FIFO . . . . .	248
8.4 Anomalía de Belady . . . . .	249
8.5 Comportamiento del algoritmo LRU . . . . .	250
8.6 Comportamiento del algoritmo del reloj . . . . .	252
8.7 Comportamiento del reloj mejorado . . . . .	253
8.8 Códigos de error de un fallo de página . . . . .	276
8.9 Llamadas al sistema relacionadas con la gestión de memoria en LINUX	278
9.1 Llamadas al sistema relacionadas con los dispositivos . . . . .	311
10.1 Funciones del sistema de ficheros . . . . .	318
10.2 Información del superbloque . . . . .	345
10.3 Influencia del tamaño del bloque en la estructura física del sistema ext2 . . . . .	346
10.4 Información del descriptor de grupo . . . . .	347
10.5 Llamadas al sistema relacionadas con el sistema de ficheros . . . . .	355
B.1 Prefijo para múltiplos binarios . . . . .	377

# Índice de Programas

---

2.1	Llamada al sistema . . . . .	30
5.1	Utilización de un recurso . . . . .	117
5.2	Primer intento . . . . .	124
5.3	Segundo intento . . . . .	125
5.4	Tercer intento . . . . .	127
5.5	Cuarto intento . . . . .	128
5.6	Algoritmo de Dekker . . . . .	130
5.7	Algoritmo de Peterson . . . . .	132
5.8	Algoritmo de Lamport . . . . .	133
5.9	Deshabilitación de interrupciones . . . . .	134
5.10	Instrucción comprobar_y_establecer . . . . .	135
5.11	Exclusión mutua con comprobar_y_establecer . . . . .	135
5.12	Instrucción intercambiar . . . . .	136
5.13	Exclusión mutua con intercambiar . . . . .	136
5.14	Semáforo de Dijkstra . . . . .	138
5.15	Semáforo . . . . .	139
5.16	Semáforos contadores con semáforos binarios . . . . .	140
5.17	Semáforos binarios con semáforos contadores . . . . .	141

---

5.18 Exclusión mutua con semáforos . . . . .	142
5.19 Productor/Consumidor con semáforos . . . . .	143
5.20 Productor/Consumidor con monitor . . . . .	148
5.21 Exclusión mutua con mensajes . . . . .	152
7.1 Traducción de direcciones en un sistema paginado . . . . .	215
7.2 Traducción de direcciones en un sistema segmentado . . . . .	221
8.1 Conjunto de trabajo . . . . .	262
8.2 Frecuencia de fallos de página . . . . .	264

# Índice de Algoritmos

---

4.1	Planificador de procesos . . . . .	107
5.1	Esquema para resolver la exclusión mutua . . . . .	123
6.1	Negación de la no apropiación . . . . .	168
6.2	Algoritmo del banquero . . . . .	172
6.3	Algoritmo de seguridad . . . . .	173
6.4	Algoritmo de detección . . . . .	177
8.1	Algoritmo del reloj . . . . .	251



---

**Parte 1**

# **INTRODUCCIÓN**

---



# Capítulo 1

## Introducción a los Sistemas Operativos

---

Un elemento esencial en todo sistema de computación es el sistema operativo. En este capítulo veremos cuál es su función y objetivo principal: abstraer el hardware para ofrecer un conjunto de servicios a los usuarios. Con el fin de comprender mejor los distintos elementos que componen un sistema operativo contemporáneo y sus objetivos, veremos como a lo largo de la historia se han ido introduciendo innovaciones para aumentar el rendimiento del sistema.

### 1.1 ¿Qué es un sistema operativo?

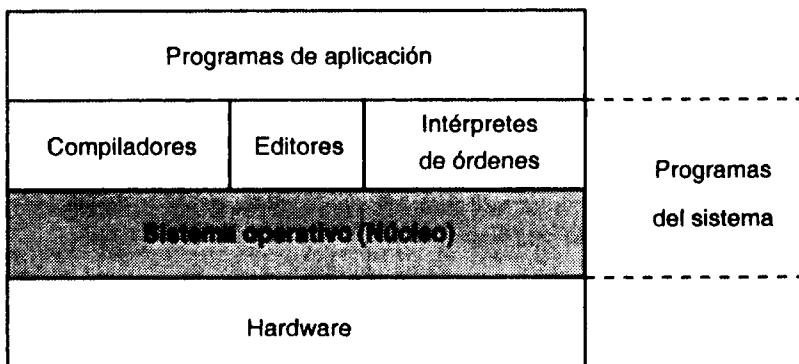
Todo sistema de computación consta de una parte software y otra hardware. El software del computador puede dividirse de modo general en dos clases:

- los **programas del sistema**, que controlan el funcionamiento del computador.
- los **programas de aplicación**, que resuelven los problemas de los usuarios y son específicos para un tipo de problema.

Respecto al hardware, un computador es un sistema complejo que consta de uno o más procesadores, memoria central, relojes, terminales, discos y otros dis-

positivos de E/S (ver apéndice A). Si los programadores tuvieran que tener en cuenta cómo funciona cada uno de estos elementos a la hora de hacer un programa, la elaboración de uno de ellos sería una tarea bastante difícil.

Para evitar que los programadores tengan que enfrentarse a la complejidad del hardware, se introduce una capa software por encima del primero, con objeto de manejar todas las partes del sistema y presentar al usuario una interfaz que sea más fácil de entender y programar. Esta capa es el **sistema operativo** (figura 1.1).



**Figura 1.1:** Estructura de un sistema de computación

El sistema operativo es la parte más importante de lo que hemos llamado **programas del sistema**, ya que controla todos los recursos del computador y ofrece la base sobre la cual pueden escribirse los programas de aplicación.

Sobre la capa software que constituye el sistema operativo descansa el resto del software del sistema. Aquí se encuentran el intérprete de órdenes, los compiladores, editores de texto, etc. Estos programas se suelen distribuir junto con el sistema operativo para otorgarle una mayor funcionalidad y comodidad al entorno, por lo que los usuarios pueden creer que forman parte de él. Esto no es así, la diferencia fundamental que existe entre ambos es que el sistema operativo se ejecuta en **modo núcleo**, mientras que los compiladores y editores funcionan en **modo usuario**. El modo núcleo es un modo de ejecución privilegiado que le permite tener acceso a todo el hardware, a la vez que lo protege de la manipulación indebida por parte del usuario. Si a uno no le agrada un compilador determinado tiene la libertad de escribir uno propio si así lo desea, pero no la tiene para escribir su propio manejador de interrupciones del disco, porque es parte del sistema operativo y está protegido por el hardware contra intentos de modificación por parte del usuario.

Se denomina **núcleo** del sistema operativo a la parte que se comunica directamente con el hardware, éste será el verdadero corazón y motor del sistema.

La pregunta ¿qué es un sistema operativo? es difícil de contestar; debido principalmente a que éstos realizan dos funciones que no están relacionadas entre sí. Para intentar responderla veamos qué es lo que hace un sistema operativo.

Los programas que se ejecutan en el sistema necesitan utilizar recursos. Desde el punto de vista del programa lo ideal sería disponer de todos los recursos del sistema, esto implicaría que no pudiese coexistir junto con otros. Como se verá más adelante en este capítulo, para mejorar el rendimiento del sistema es conveniente que haya varios programas activos simultáneamente. Para conseguir esto es necesario que se repartan los recursos entre todos los programas que compiten por ellos. Esta función la realiza el sistema operativo. Así, podemos decir que éste se comporta como **administrador de los recursos** del sistema, distribuyéndolos entre los programas que se ejecutan en él. Es decir, el sistema operativo concede o deniega las peticiones de recursos que hacen los programas en ejecución.

La gestión de los recursos que realiza el sistema operativo consigue que los programas en ejecución tengan la ilusión de que están solos en el sistema, disponiendo de todos los recursos de éste. En realidad, lo que existe es un reparto transparente de los recursos entre todos los programas que compiten por ellos.

Como podemos ver en la figura 1.1, el sistema operativo se sitúa entre el hardware y el resto del software del sistema, para recibir las peticiones de los distintos programas y controlar de esta forma la asignación de los recursos. Esto añade una funcionalidad más al sistema operativo, que es la de proporcionar una capa de **abstracción del hardware**. La arquitectura de muchos computadores en el nivel del lenguaje máquina es primitiva y difícil de programar, especialmente lo referido a las operaciones de E/S, debido a la gran variedad de dispositivos existentes. Sería muy complicado para un programador introducirse en los detalles reales de cómo se realiza una operación de lectura o escritura en un disco, por ejemplo. Además, el simple hecho de cambiar de dispositivo implicaría modificar el código del programa. El sistema operativo se encarga de abstraer estos detalles. En el caso de los discos, el sistema operativo presenta la abstracción de que éstos contienen información almacenada como un conjunto de ficheros con un nombre. Para poder acceder a la información sólo se necesita saber el nombre del fichero, y no las operaciones reales que deben realizarse con el disco.

De esta forma, la función del sistema operativo es la de presentar al usuario una máquina ampliada que sea más fácil de programar que el hardware implícito. Es decir, actúa como una interfaz para las distintas peticiones del software.

## 1.2 Evolución histórica de los sistemas operativos

Para llegar a entender mejor qué son y qué hacen los sistemas operativos modernos, es conveniente considerar la evolución que han sufrido a lo largo de su historia. El estudio de esta evolución nos va a conducir a través de los problemas que se planteaban y las soluciones presentadas, lo que nos permitirá entender mejor cómo y por qué se han desarrollado de la forma que lo han hecho.

Los sistemas operativos y la arquitectura de los computadores se han influido mucho mutuamente. Por un lado, los sistemas operativos se han desarrollado para facilitar el uso del hardware y, por otro, a medida que se iban diseñando nuevos sistemas operativos y se planteaban nuevos problemas, se hacía evidente que ciertos cambios en el diseño del hardware podían solucionar los problemas existentes o bien hacer más simple el sistema operativo.

### 1.2.1 Los primeros sistemas

Los primeros computadores eran máquinas enormes, ocupaban cuartos enteros y contenían decenas de miles de tubos de vacío, pero eran mucho más lentos que el más barato de los computadores domésticos que existe hoy.

En estos primeros días un grupo de personas diseñaba, construía, programaba y mantenía cada máquina. La programación se realizaba en lenguaje máquina, ya que no existía todavía ningún lenguaje de programación de alto nivel. Cuando se quería ejecutar un programa había que cargarlo manualmente en memoria; esto se hacía al principio desde un panel con clavijas y posteriormente mediante tarjetas o cintas perforadas. Después había que pulsar los botones adecuados para cargar la dirección de comienzo y empezar la ejecución del programa. A medida que el programa se ejecutaba, el programador podía seguir su ejecución mediante las luces que se encendían en la consola. Si se descubría algún error, el programador podía parar la ejecución del programa, examinar el contenido de la memoria y de los registros del procesador, y depurar el programa directamente desde la consola. La salida del programa se imprimía o se obtenía en tarjetas o cintas perforadas. En este tipo de sistemas la memoria estaba ocupada únicamente por el programa que se estaba ejecutando; eran sistemas **monoprogramados**.

Un aspecto a destacar de esta forma de trabajo era su naturaleza **interactiva**. El programador era también el operador del sistema. Cada programador se reservaba el uso exclusivo de la máquina durante un tiempo determinado; se trataba de un **sistema monousuario**. Este tipo de acceso al computador puede provocar ciertos problemas. Supongamos que reservamos una hora de uso del computador para ejecutar un programa que hemos desarrollado. Si nos encontra-

mos con un error difícil de detectar y no somos capaces de finalizar la ejecución durante dicho tiempo, como probablemente alguien habrá reservado el uso de la máquina a continuación, tendremos que parar, recoger toda la información que podamos y volver más tarde para continuar. Por otra parte, si las cosas nos van bien y terminamos antes de lo previsto, durante el resto del tiempo reservado la máquina permanecerá ociosa.

Con el tiempo se desarrollaron hardware y software adicional. Aparecieron dispositivos como las lectoras de tarjetas y las unidades de cinta magnética. Se desarrollaron los ensambladores, cargadores y enlazadores, para facilitar las tareas de programación, así como las bibliotecas de funciones comunes, que pueden ser utilizadas por cualquier programa sin tener que escribirlas de nuevo.

Las rutinas que realizan las operaciones de E/S son especialmente importantes. Cada dispositivo de E/S tiene sus propias características, por lo que hay que escribir una rutina específica para él. Estas rutinas, llamadas **manejadores de dispositivos** (*device drivers*), se colocan en bibliotecas de uso común para no tener que ser escritas para cada programa.

Más tarde aparecieron los compiladores de Fortran, Cobol y otros lenguajes. Esto facilitó mucho la tarea del programador, sin embargo hizo más compleja la tarea del computador. Por ejemplo, para preparar un programa escrito en Fortran para su ejecución, el programador tiene que cargar primero el compilador de Fortran en el computador (normalmente el compilador estaba almacenado en cinta magnética). El programa puede ser leído por una lectora de tarjetas y pasado a cinta magnética. El compilador de Fortran produce una salida en lenguaje ensamblador que necesita ser ensamblada con el ensamblador, lo que requiere montar otra cinta con el ensamblador. La salida del ensamblador necesita ser enlazada con las rutinas del sistema adecuadas. Finalmente, la forma binaria del programa está lista para ser ejecutada. Puede ser cargada en memoria y depurada desde la consola como antes. A este conjunto de labores que hay que realizar para ejecutar un programa se le conoce con el nombre de **trabajo**.

### 1.2.2 Sistemas por lotes

El tiempo adicional que se gastaba en la preparación de los trabajos constituía un problema, ya que durante el tiempo en que se están montando las cintas o el programador está operando en la consola, la CPU permanece ociosa. En estos primeros tiempos había pocos computadores y eran muy caros, por lo que su tiempo era muy valioso y sus propietarios querían que fuesen usados al máximo.

La solución dada a este problema fue doble. Por un lado, aparece el operador

profesional y el programador ya no interviene en la operación de la máquina. Así, tan pronto como un trabajo finaliza, el operador arranca el siguiente; por tanto, no hay ya tiempo ocioso debido a que se reserve un cierto tiempo de uso del computador. Al tener el operador más experiencia en el montaje de cintas que el programador, el tiempo de preparación también se reduce. El usuario le proporcionará todas las tarjetas perforadas o cintas que se van a necesitar, así como una corta descripción de cómo debe ser ejecutado el trabajo. Por supuesto, el operador no se encarga de depurar un programa incorrecto, puesto que él no tiene por qué entender el programa. Cuando un programa falla, obtiene un volcado de la memoria que entrega al programador. Así, el operador podrá continuar con el siguiente trabajo. Sin embargo, al programador se le hace más difícil depurar su programa.

También se consigue un ahorro en el tiempo que se dedica a la preparación de los trabajos reuniendo aquellos que tengan necesidades similares en un lote y procesándolos como un grupo. Esto es lo que se conoce como **procesamiento por lotes** (*batch*). Por ejemplo, supongamos que el operador recibe un trabajo en Fortran, un trabajo en Cobol y otro en Fortran. Si los ejecuta en ese orden tendrá que preparar el computador para ejecutar el trabajo en Fortran, luego para el trabajo en Cobol y después otra vez para el otro trabajo en Fortran. Sin embargo, si ejecuta los dos trabajos en Fortran como un lote sólo tendría que preparar el computador una vez para los dos, con lo cual se ahorra tiempo.

Los cambios introducidos, haciendo que la persona que opera con la máquina sea distinta del usuario y el procesamiento por lotes de trabajos similares, modifican la forma de comunicación entre el programador y la máquina perdiendo su naturaleza interactiva. Estos cambios mejoraron un poco la utilización de los computadores, pero todavía seguía habiendo problemas. Por ejemplo, cuando un trabajo se para, el operador tiene que darse cuenta observando la consola, determinar por qué se ha parado (si es normal o no), hacer un volcado de la memoria si es necesario, y preparar el lector de tarjetas o de cintas con el trabajo siguiente, volviendo a arrancar el computador. Durante esta transición de un trabajo a otro la CPU permanece sin hacer nada.

Para suprimir este tiempo ocioso, se introdujo un secuenciador automático de trabajos, y con él, se creó el primer sistema operativo. Lo que se quería era un procedimiento para transferir automáticamente el control de un trabajo al próximo. Con este propósito se crea un pequeño programa llamado **monitor residente**, que siempre está (residente) en memoria. Como puede verse en la figura 1.2, ahora la memoria contiene al monitor residente y al programa que el usuario manda a ejecutar.

El monitor residente tiene el control del computador cuando éste se enciende,

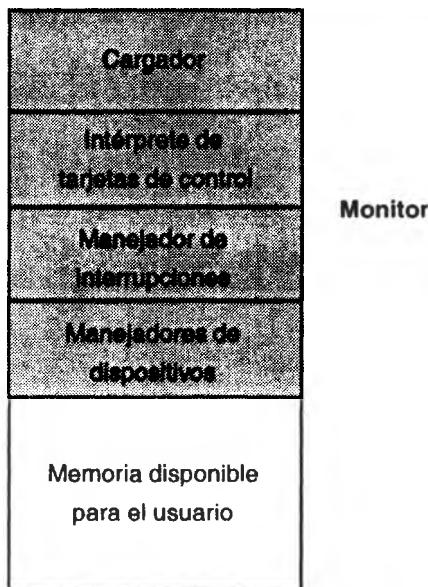
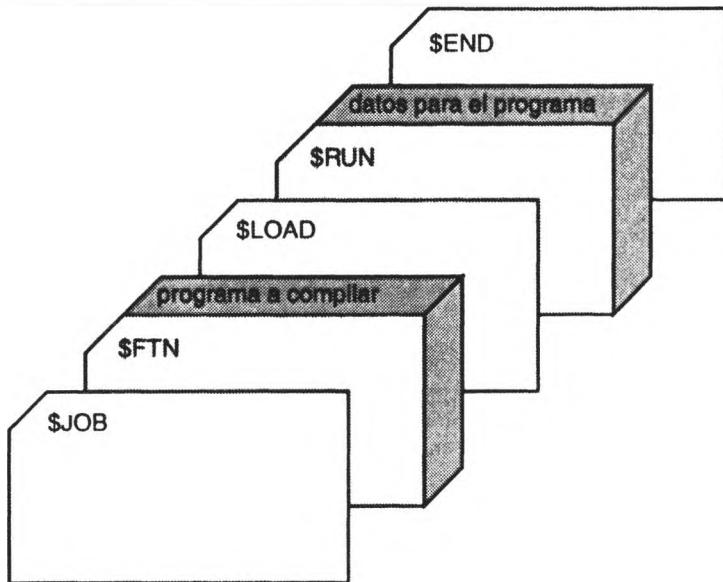


Figura 1.2: Estructura de la memoria con monitor residente

y este control puede transferirse a un programa cuando sea necesario. Cuando termine su ejecución devuelve el control al monitor residente. Por tanto, éste se encarga de transferir el control de un programa a otro dentro del mismo trabajo, y de un trabajo a otro.

¿Pero cómo sabe el monitor residente qué programa ejecutar? Previamente, el operador le dará al monitor residente una breve descripción de qué programas se van a ejecutar y con qué datos, para ello introduce unas **tarjetas de control**. La idea es simple, además de las tarjetas que contienen el programa y las de los datos, se introducen unas tarjetas especiales que contienen instrucciones para el monitor residente indicándole qué programa va a ejecutar. Para distinguir estas tarjetas de control de las de datos o programa, se las identifica con un carácter especial, tal como \$, //, etc. En la figura 1.3 se muestra el conjunto de tarjetas necesarias para ejecutar un trabajo.

En un monitor residente se pueden distinguir varios componentes (figura 1.2). Uno es el intérprete de tarjetas de control que es el encargado de traducir cada orden en una ejecutable en código máquina. Éste llamará cada cierto tiempo al cargador para cargar los programas en memoria. Tanto el intérprete de tarjetas de control como el cargador necesitarán realizar operaciones de E/S, por tanto el monitor residente tiene un conjunto de manejadores de dispositivos para trabajar



**Figura 1.3:** Conjunto de tarjetas de un trabajo

con los dispositivos de E/S del sistema. También se necesita un módulo para el tratamiento de las interrupciones producidas por los dispositivos de E/S, el **manejador de interrupciones**.

El monitor residente proporcionaba una secuenciación automática de los trabajos mediante las indicaciones de las tarjetas de control. Cuando una tarjeta de control indica que se debe ejecutar un programa, el monitor lo carga en memoria y le transfiere el control. Cuando el programa termina, devuelve el control al monitor, que lee la próxima tarjeta de control, carga el programa apropiado y sigue. Esto se repite hasta que todas las tarjetas de control de ese trabajo han sido interpretadas. Entonces el monitor continúa automáticamente con el próximo trabajo.

Un **sistema operativo por lotes** normalmente lee un flujo de trabajos (de una lectora de tarjetas, por ejemplo), cada uno con sus propias tarjetas de control que predefinen lo que hace el trabajo. Cuando el trabajo termina, normalmente su salida se imprime. La característica principal de un sistema por lotes es la falta de interacción entre el usuario y el trabajo mientras éste se está ejecutando. El trabajo se prepara y se envía; algún tiempo después (minutos, horas o días), aparece la salida. La demora entre el momento de envío del trabajo y su terminación, que se conoce como **tiempo de retorno**, es debida a la cantidad de computación

que éste necesita y al tiempo que se tarda en empezar a procesarlo.

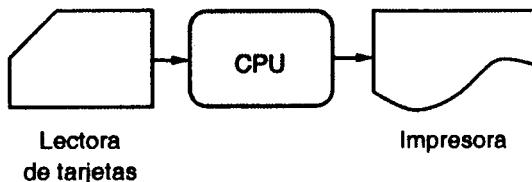
La introducción del monitor residente hace necesaria la introducción de diversos mecanismos hardware de protección, para evitar que un programa de usuario acceda a la zona de memoria del monitor, o que monopolice el uso del procesador, por ejemplo. En el apartado 1.6 se tratarán los mecanismos necesarios.

El cambio a sistemas por lotes con secuenciación automática de trabajos se hizo para mejorar el rendimiento de la CPU. Pero aún así hay momentos en que ésta permanece ociosa. El problema es debido a que la velocidad de los dispositivos mecánicos de E/S es intrínsecamente más lenta que la de los dispositivos electrónicos. Una CPU lenta puede ejecutar millones de instrucciones por segundo, mientras que una lectora de tarjetas rápida puede leer unas 1200 tarjetas por minuto. Con el tiempo, los progresos de la tecnología producen dispositivos de E/S más rápidos. Pero las velocidades de la CPU también aumentan, incluso de forma más rápida; por tanto, el problema no sólo no queda resuelto sino que incluso empeora.

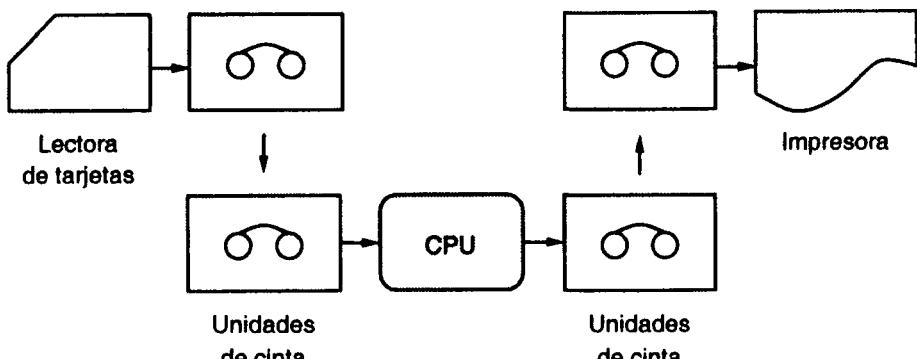
### 1.2.2.1 Procesamiento fuera de línea

Una solución a este problema fue reemplazar las lectoras de tarjetas (dispositivos de entrada) y las impresoras de línea (dispositivos de salida) por unidades de cinta magnética. Ahora, en vez de leer directamente las tarjetas, éstas se copian primero en cinta magnética. Cuando un programa necesita tomar su entrada de una tarjeta, el registro equivalente se lee de la cinta. De forma similar, la salida de los programas se escribe en cinta magnética y el contenido de ésta se imprime más tarde. Las lectoras de tarjetas y las impresoras operaban **frente a frente**, es decir, sin utilizar el computador principal (figura 1.4).

La ventaja principal de la operación fuera de línea es que permite utilizar múltiples sistemas lectora-a-cinta y cinta-a-impresora para una CPU. De esta forma, si la CPU puede procesar la entrada dos veces más rápido que una lectora de tarjetas, entonces dos lectoras trabajando simultáneamente pueden producir bastantes cintas para mantener a la CPU ocupada. Por tanto, el computador principal ya no está coartado por la velocidad de las lectoras de tarjetas e impresoras de línea, sino por la velocidad de las unidades de cinta magnética, que son mucho más rápidas. Sin embargo, ahora hay una demora aún mayor a la hora de ejecutar un trabajo particular. Primero debe ser pasado a cinta y hay que esperar que haya bastantes trabajos hasta llenarla. Una vez llena tiene que ser rebobinada, desmontada, llevada a la CPU y montada en una unidad de cinta libre.



(a) Operación en línea



(b) Operación fuera de línea

Figura 1.4: Modos de operación

### 1.2.2.2 *Buffering*

El procesamiento fuera de línea permite un solapamiento de las operaciones de la CPU y de E/S mediante la ejecución de estas acciones en dos máquinas independientes. Si nos fijamos en la ejecución de un trabajo observaremos que consiste en el uso alterno del procesador y la espera de operaciones de E/S; se suele decir que se alternan ráfagas de CPU y de E/S. El *buffering* es un método que permite simultanear las operaciones de la CPU y de E/S para un mismo trabajo en una misma máquina.

Consiste en dotar a los dispositivos de E/S de *buffers* donde almacenar la información temporalmente. De esta forma, mientras la CPU está procesando un dato que se acaba de leer, el dispositivo puede estar realizando otra operación de entrada, situando el nuevo dato en el *buffer*. Análogamente, cuando se trata de una operación de salida, la CPU puede estar introduciendo los datos en el *buffer*, mientras el dispositivo realiza una operación de salida. En la figura 1.5 podemos ver un diagrama temporal donde se compara un sistema sin *buffer* y otro con un *buffer*. Se puede apreciar cómo el sistema que posee el *buffer* puede solapar las

operaciones de cálculo con el primer dato mientras realiza la entrada del segundo. Esto permite aumentar el rendimiento del sistema, ya que el sistema sin *buffer* sólo puede empezar a leer el segundo dato cuando termina de procesar el primero.

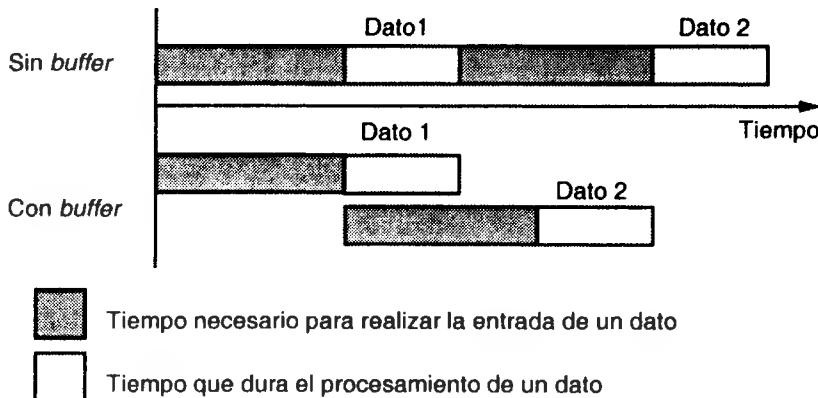


Figura 1.5: *Buffering*

Los *buffers* permiten reducir las diferencias entre la velocidad a la que se procesan los datos y a la que se realizan las operaciones de E/S, consiguiendo aumentar el rendimiento del sistema al mantener menos tiempo ociosa a la CPU. Pero los *buffers* no son realmente efectivos si los trabajos no presentan un intercalamiento de operaciones de E/S y de cálculo. Así, los trabajos limitados por la CPU que necesitan ráfagas largas de CPU y realizan pocas operaciones de E/S mantendrán los *buffers* de entrada siempre llenos y los de salida vacíos. Por otro lado, los limitados por la E/S, que realizan muchas operaciones de E/S y necesitan ráfagas de CPU cortas, mantienen vacíos los *buffers* de entrada y llenos los de salida, por lo que la CPU se ve limitada por la velocidad de los dispositivos.

### 1.2.2.3 *Spooling*

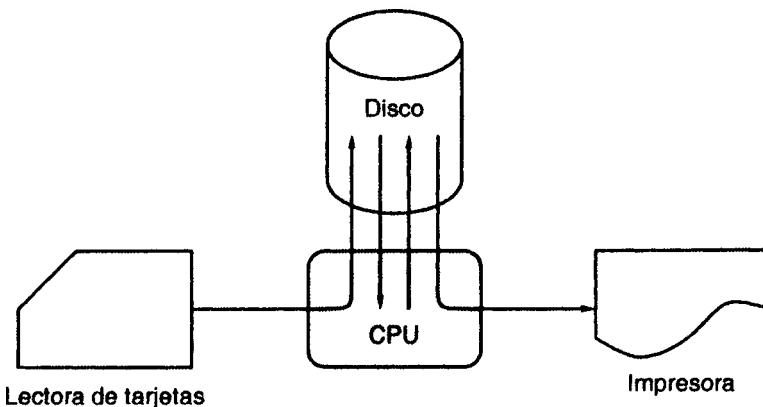
La aparición de los discos introduce una nueva forma de procesamiento denominada *spooling*<sup>1</sup> que permite solapar las operaciones de E/S y de la CPU utilizando una sola máquina. La diferencia fundamental entre los discos y las cintas (dispositivos de almacenamiento utilizado hasta el momento) es su modo de acceso. Los discos son dispositivos de acceso aleatorio, por lo que se puede pasar rápidamente de una zona a otra, mientras que las cintas sólo permiten un acceso secuencial.

En un sistema de *spooling* (figura 1.6), las tarjetas son leídas directamente del

<sup>1</sup> El nombre es un acrónimo de *simultaneous peripheral operation on-line*.

lector de tarjetas al disco. Cuando se ejecuta un trabajo, el sistema operativo satisface sus peticiones de entrada del lector de tarjetas leyendo del disco. De forma similar, cuando el trabajo hace una petición para escribir en la impresora una línea, la información se copia en un *buffer* del sistema situado en el disco. Cuando el trabajo se completa, la salida se envía a la impresora.

El *spooling*, en esencia, usa el disco como un gran *buffer*, para adelantar la lectura tanto como sea posible en dispositivos de entrada y para almacenar la información de salida hasta que los dispositivos sean capaces de aceptarla.

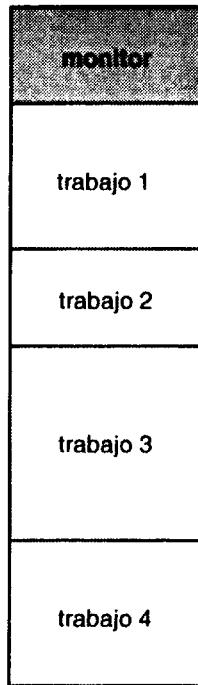


**Figura 1.6: Spooling**

El *spooling* solapa la E/S de un trabajo con la computación de otros. Incluso en un sistema simple el *spooler* puede estar leyendo la entrada de un trabajo mientras está imprimiendo la salida de otro diferente.

Esta técnica tiene un beneficio directo sobre el rendimiento del sistema. Por el costo de algo de espacio en disco, la CPU puede solapar la computación de un trabajo con la E/S de otros. Así, el *spooling* puede mantener trabajando tanto a la CPU como a los dispositivos de E/S a velocidades mucho más altas.

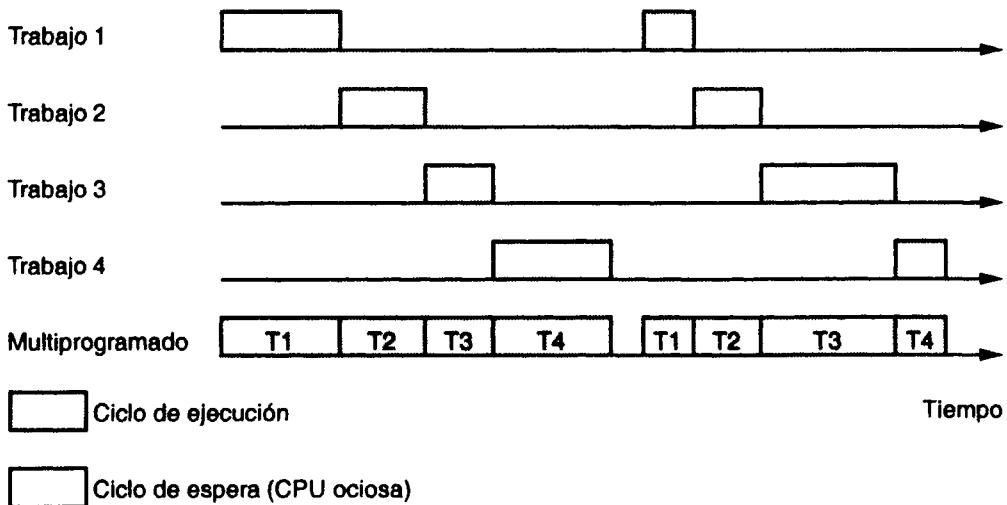
El *spooling* permite disponer de un **pool de trabajos**, es decir, un conjunto de trabajos en disco dispuestos para ejecutarse. Esto va a permitir al sistema operativo seleccionar el próximo trabajo a ejecutar cuando el que está ejecutando actualmente tiene que realizar una operación de E/S. Cuando los trabajos están en tarjetas o cinta magnética, no es posible saltar y ejecutarlos en un orden diferente, sino que tienen que ser ejecutados secuencialmente. Sin embargo, cuando varios trabajos están en un dispositivo de acceso directo, tal como un disco, es posible hacer una **planificación** de éstos.



**Figura 1.7:** Estado de la memoria en un sistema de multiprogramación

### 1.2.3 Sistemas por lotes multiprogramados

La operación fuera de línea, el *buffering* y el *spooling* tienen sus limitaciones para solapar la E/S. Un solo trabajo no puede, en general, mantener ocupados todo el tiempo la CPU y los dispositivos de E/S. Una forma de conseguirlo es que el sistema operativo mantenga varios trabajos en memoria a un tiempo como se muestra en la figura 1.7, esto es lo que se conoce como **multiprogramación**. Este conjunto de trabajos será normalmente un subconjunto de los que se mantienen en el disco (ya que la capacidad de la memoria es usualmente mucho menor que la del disco). El sistema operativo escoge uno de los trabajos que está en memoria y empieza a ejecutarlo. Si en algún momento el trabajo tiene que esperar algo, como por ejemplo que se monte una cinta, que se dé una orden, o que se complete una operación de E/S, el sistema operativo podrá elegir otro trabajo para ejecutarlo. En un sistema donde no existe la multiprogramación, la CPU estaría ociosa durante este tiempo. Con la multiprogramación aumenta la utilización de la CPU organizando los trabajos de forma que ésta siempre tenga algo que ejecutar. Esto es lo que se representa en la figura 1.8.



**Figura 1.8:** Ejecución de trabajos en un sistema multiprogramado

Un concepto básico en estos sistemas es el **grado de multiprogramación**, que se define como el número de trabajos cargados simultáneamente en memoria.

Los **sistemas operativos por lotes multiprogramados** son claramente más sofisticados que los anteriores, pues son los primeros que toman decisiones por los usuarios. Todos los trabajos que entran en el sistema se mantienen en el *pool* de trabajos a la espera de obtener espacio en la memoria principal para comenzar su ejecución. Si varios trabajos están listos para ser llevados a memoria, y no hay suficiente sitio para todos ellos, entonces el sistema debe elegir. Esta decisión es lo que se conoce como **planificación de trabajos**. Cuando el sistema operativo selecciona un trabajo del *pool*, lo carga en memoria para su ejecución. Cuando tenemos varios trabajos en memoria dispuestos para ejecutarse, también el sistema debe elegir entre ellos para decidir cuál es el que se ejecutará primero; esto es lo que se conoce como **planificación de la CPU**. Todos estos aspectos serán estudiados con mayor amplitud en el capítulo 4.

El tener varios trabajos cargados en memoria al mismo tiempo requiere un manejo de la memoria más complejo que en los sistemas anteriores, al tener que decidir en qué zonas van a ubicarse los distintos trabajos. En el capítulo 7 se analizará este problema con mayor amplitud. Por otro lado, la existencia de varios trabajos activos simultáneamente hace imprescindible proteger los distintos recursos a los que pueden acceder, tales como la memoria, dispositivos de E/S y la CPU, con objeto de evitar interferencias entre ellos. Los mecanismos de protección necesarios se tratarán en más profundidad en el apartado 1.6.

### 1.2.4 Sistemas de tiempo compartido

Los sistemas por lotes con multiprogramación proporcionan un entorno en el que los diferentes recursos del sistema (CPU, memoria, dispositivos de E/S) se utilizan de una forma más efectiva. Sin embargo, desde el punto de vista del programador o del usuario, estos sistemas presentan algunas dificultades. Éstas son debidas a la falta de interacción entre los usuarios y el sistema, por ejemplo, los programas deben ser depurados estáticamente, a partir de los volcados de memoria. El programador no puede modificar un programa a medida que se ejecuta para estudiar su comportamiento.

**Los sistemas de tiempo compartido** son una extensión lógica de los sistemas de multiprogramación. En ellos, múltiples trabajos son ejecutados por la CPU intercambiándose entre ellos, además los cambios ocurren tan frecuentemente que los usuarios pueden interaccionar con cada programa mientras se está ejecutando. Un sistema de tiempo compartido es un sistema de multiprogramación interactivo; esto permite que el usuario dé instrucciones al sistema operativo o a un programa en ejecución directamente, y reciba una respuesta inmediata. De esta forma el usuario puede experimentar fácilmente y ver los resultados inmediatamente.

En estos sistemas, el usuario envía órdenes y espera los resultados, por tanto, el **tiempo de respuesta** debería ser lo más pequeño posible. Esta es una de las condiciones principales que se le debe imponer a un sistema interactivo.

Los primeros computadores eran sistemas interactivos, ya que el sistema completo estaba a disposición del usuario. Esto permitía al programador gran flexibilidad y libertad en la prueba y desarrollo de programas. Pero daba lugar a que la CPU estuviera ociosa durante períodos largos de tiempo. Debido al alto coste de estos primeros computadores, el tiempo de CPU ocioso no era deseable, por lo que se desarrollaron los sistemas por lotes. Éstos mejoraron la utilización del sistema para los propietarios de los computadores.

Los sistemas de tiempo compartido son el resultado de intentar proporcionar un uso interactivo del computador a un costo razonable. Un sistema operativo de tiempo compartido utiliza la planificación de la CPU y la multiprogramación para proporcionar a cada usuario una pequeña porción de un computador de tiempo compartido. Cuando un programa se ejecuta, normalmente lo hace durante períodos cortos de tiempo, teniendo que liberar la CPU cada vez que transcurre el tiempo máximo permitido o bien cuando necesita realizar una operación de E/S. Siempre que se dan estas circunstancias el sistema pasa a ejecutar otro programa diferente.

Un sistema operativo de tiempo compartido permite a muchos usuarios utilizar simultáneamente el computador; se trata, por tanto, de un sistema **multiusua-**

rio, al contrario que los sistemas anteriores que eran todos monousuario. Esto se consigue dando a cada usuario el uso de la CPU durante un intervalo corto de tiempo. Es decir, la CPU va atendiendo a los distintos usuarios del sistema durante períodos cortos de tiempo, dándoles a estos la impresión de que el computador está a su entera disposición.

La idea del tiempo compartido fue demostrada al principio de los años 60, pero como estos sistemas eran más difíciles y caros de construir, no llegaron a ser comunes hasta principios de los años 70.

El tiempo compartido plantea nuevos problemas para el sistema operativo. La presencia de varios usuarios interactivos introduce la necesidad de proteger los ficheros para que sólo los usuarios autorizados puedan tener acceso a éstos.

### 1.3 Sistemas para computadores personales

Con el tiempo los costos del hardware han disminuido mucho, haciendo posible de nuevo tener un computador dedicado a un solo usuario. Éstos se conocen comúnmente como **computadores personales**.

Hasta hace poco, los procesadores de estos computadores habían perdido las características que se necesitaban para proteger al sistema operativo de los programas de los usuarios. Los sistemas operativos que se escribían para ellos eran monousuarios y monoprogramados. Las metas de estos sistemas operativos también son diferentes, en vez de intentar maximizar la utilización de la CPU y de los periféricos, optan por la comodidad del usuario.

A medida que los computadores personales se han ido haciendo más potentes, sus sistemas operativos han adoptado algunas de las características de los sistemas estudiados anteriormente.

### 1.4 Sistemas de tiempo real

Los **sistemas operativos de tiempo real** son sistemas de propósito especial construidos para resolver problemas concretos. Se utilizan cuando existen requisitos estrictos de tiempo en la operación del procesador o en el flujo de datos. En estos sistemas la corrección de su operación depende no sólo de los resultados lógicos de la computación, sino también del tiempo que tarda en producirse estos resultados. Algunos ejemplos de uso de sistemas de tiempo real son el control de procesos de producción en la manufacturación de productos, los sistemas de

inyección de gasolina de algunos automóviles, etc. En todos los casos, unos sensores suministran información sobre el entorno y el sistema tiene que responder de forma adecuada a los cambios que se produzcan. Esta respuesta tiene unos límites bien definidos de tiempo, y si no se produce dentro de éstos el sistema fallará.

Se pueden distinguir dos tipos de sistemas de tiempo real: los duros y los suaves. Para los sistemas de tiempo real duros es absolutamente imprescindible que las respuestas se produzcan dentro de los límites de tiempo especificados. Los sistemas de tiempo real suaves son menos restrictivos, sigue siendo importante que los tiempos de respuesta estén dentro de los límites, pero si alguna vez no lo están, el sistema seguirá funcionando. Estos sistemas tienen una utilidad más limitada que los duros, ya que no pueden ser utilizados para control industrial y robótica. Sin embargo, hay varias áreas donde son útiles, por ejemplo, multimedia, realidad virtual, etc. Estos sistemas disponen de características de los sistemas operativos avanzados que no están disponibles en los sistemas de tiempo real duros.

## 1.5 Sistemas con más de un procesador

Todos los sistemas considerados hasta ahora se ejecutan en computadores con un solo procesador. Posteriormente, ha empezado a construirse sistemas con más de un procesador, por lo que el sistema operativo se ha tenido que adaptar a esta circunstancia. Estos sistemas proporcionan un ahorro de dinero con respecto a los sistemas de un solo procesador, debido a la compartición entre todos ellos de los mismos periféricos, etc. Existen distintos tipos de sistemas con múltiples procesadores: los sistemas paralelos, en red y distribuidos.

### 1.5.1 Sistemas paralelos

Estos sistemas tienen varios procesadores que comparten el bus del sistema, el reloj, la memoria, etc. Hay varias razones que justifican la construcción de este tipo de sistemas. Una ventaja es el aumento de rendimiento. Es de esperar que al aumentar el número de procesadores que trabajan juntos, aumente la cantidad de trabajo que se realiza por unidad de tiempo. Esto es lo que suele ocurrir, aunque el aumento de rendimiento no es exactamente proporcional al número de procesadores, sino algo menor, ya que suele haber cierta sobrecarga en este tipo de sistemas para hacer que todos los procesadores trabajen de forma coordinada.

Otra razón para la construcción de este tipo de sistemas es el aumento de fiabilidad. En ellos, el fallo de un procesador no paraliza el sistema completo, sólo se hace más lento.

La mayor parte de estos sistemas usan hoy día el **multiprocesamiento simétrico**, en el cual cada procesador ejecuta una copia idéntica del sistema operativo. Algunos sistemas usan lo que se conoce como **multiprocesamiento asimétrico**, en el cual a cada procesador se le asigna una tarea específica. Un procesador maestro controla el sistema; los demás procesadores pueden esperar a recibir instrucciones del procesador maestro o tienen tareas predefinidas.

### 1.5.2 Sistemas en red y distribuidos

A mediados de la década de los 80 se empezaron a construir redes de ordenadores personales que ejecutaban sistemas operativos en red y distribuidos.

En los dos casos tenemos un conjunto de ordenadores conectados mediante una red de comunicación que establece la necesidad de un protocolo de comunicaciones entre los distintos equipos. Una diferencia entre ambos es el tipo de sistema operativo que ejecutan. En el caso de los sistemas en red cada máquina tiene su propio sistema operativo, no teniendo que ser iguales los de todas las máquinas; sobre éste se ejecuta un software de red que sirve para comunicar las máquinas. Los sistemas distribuidos se caracterizan porque el sistema operativo que se ejecuta en todas las máquinas es idéntico.<sup>2</sup>

Existe otra diferencia fundamental entre ambos tipos de sistemas, en el caso de un sistema operativo en red los usuarios son conscientes de la existencia de múltiples computadores, pueden acceder a máquinas remotas o copiar ficheros de una máquina a otra, pero saben que están accediendo a un recurso remoto. Los sistemas operativos en red no son fundamentalmente diferentes de los sistemas operativos para equipos con un solo procesador. Tienen características adicionales para permitir la comunicación, pero éstas no modifican la estructura esencial del sistema operativo.

Sin embargo, los sistemas distribuidos aparecen ante los usuarios como si se tratara de un sistema uniprocesador tradicional. La distribución de los recursos es transparente a los usuarios, ya que no son conscientes de qué procesador ejecuta su programa, dónde están localizados los ficheros, etc; todo esto es manejado de forma automática por el sistema operativo. Una de las formas de conseguir esta transparencia es a través del **modelo cliente-servidor**, que permite describir la interacción de los programas en un sistema distribuido. De acuerdo a este modelo, cualquier programa que se ejecute en el sistema puede proveer servicios para otros, o pedirlos. Los programas de usuario que solicitan servicios son los denominados

<sup>2</sup> Actualmente se construyen sistemas distribuidos en los que cada máquina puede ejecutar un sistema operativo diferente. En éstos existe una capa de software denominada *middleware* que es la que proporciona las características distribuidas del sistema.

clientes, y aquellos que proveen los servicios se denominan **servidores**.

## 1.6 Mecanismos hardware de protección

Los primeros sistemas se caracterizaban porque sólo residía en memoria el programa que se estaba ejecutando. Cuando aparece el monitor residente, tenemos simultáneamente en memoria a éste y al programa de usuario; en los sistemas de multiprogramación la memoria es compartida por el sistema operativo y varios programas de usuario. Por tanto, la evolución de los sistemas ha conducido a una mayor compartición de los recursos para obtener un rendimiento mayor de la máquina, pero también ha introducido nuevos problemas. Ahora un error en un programa de usuario puede afectar al sistema operativo o a los demás programas de usuario, de ahí que sea necesario introducir mecanismos de protección que aseguren el funcionamiento correcto del sistema.

El **modo de operación dual** es un mecanismo de protección hardware que consiste en disponer de dos modos de ejecución distintos: el modo usuario y el modo supervisor<sup>3</sup>. Para ello el hardware nos proporciona un **bit de modo** que nos permite distinguir entre ambos. El valor de este bit indicará en qué modo de ejecución se encuentra el sistema. Los programas de usuario se ejecutan en modo usuario, mientras que el sistema operativo se ejecuta en modo supervisor.

Cuando el sistema arranca, el hardware se inicia en modo supervisor. A continuación se carga el sistema operativo, que da el control a los programas de usuario en modo usuario. Cada vez que se produce una interrupción, el hardware pasa de modo usuario a modo supervisor, tomando el control el sistema operativo.

El modo dual nos va a permitir definir ciertas instrucciones que podrían causar daño al sistema como **instrucciones privilegiadas** que sólo se pueden ejecutar en modo supervisor. Si se intenta ejecutar una de estas instrucciones en modo usuario, el hardware la tratará como una instrucción no válida produciendo una excepción<sup>4</sup> y, por tanto, pasará el control al sistema operativo. De esta forma podemos conseguir proteger distintos aspectos del sistema; por ejemplo, podemos evitar que los programas de usuario accedan directamente a los dispositivos de E/S para operar con ellos. Definiendo todas las instrucciones de E/S como privilegiadas se logra que sea el sistema operativo el que las realice en su nombre, lográndose así la protección de la E/S.

Otro elemento a proteger es la memoria. Como hemos comentado anteriormente debemos proteger la zona donde reside el sistema operativo del acceso de

<sup>3</sup>También conocido como modo monitor, modo sistema o modo privilegiado.

<sup>4</sup>Una excepción o trampa es una interrupción producida por el software.

los programas de usuario, y a éstos entre sí. Esta protección de la memoria se puede conseguir de varias formas, nosotros vamos a describir aquí una de las más simples.

Una forma de delimitar el espacio de memoria que le corresponde a cada programa es el uso de dos registros, que se suelen denominar **registro base** y **registro límite**. El registro base mantiene la dirección de memoria más baja permitida para el programa. El registro límite puede contener el tamaño del espacio de memoria que le corresponde al programa de usuario, o bien la dirección más alta a la que puede acceder. Así, cada vez que el programa en ejecución hace referencia a una dirección de memoria, la CPU comprueba si se trata de una dirección legal o no. Si la dirección a la que se intenta acceder no es legal, se produce una excepción y el sistema operativo toma el control.

Para que esta protección sea efectiva, los valores de los registros base y límite sólo pueden ser modificados por el sistema operativo mediante una instrucción privilegiada, evitándose así que puedan ser modificados por un programa de usuario.

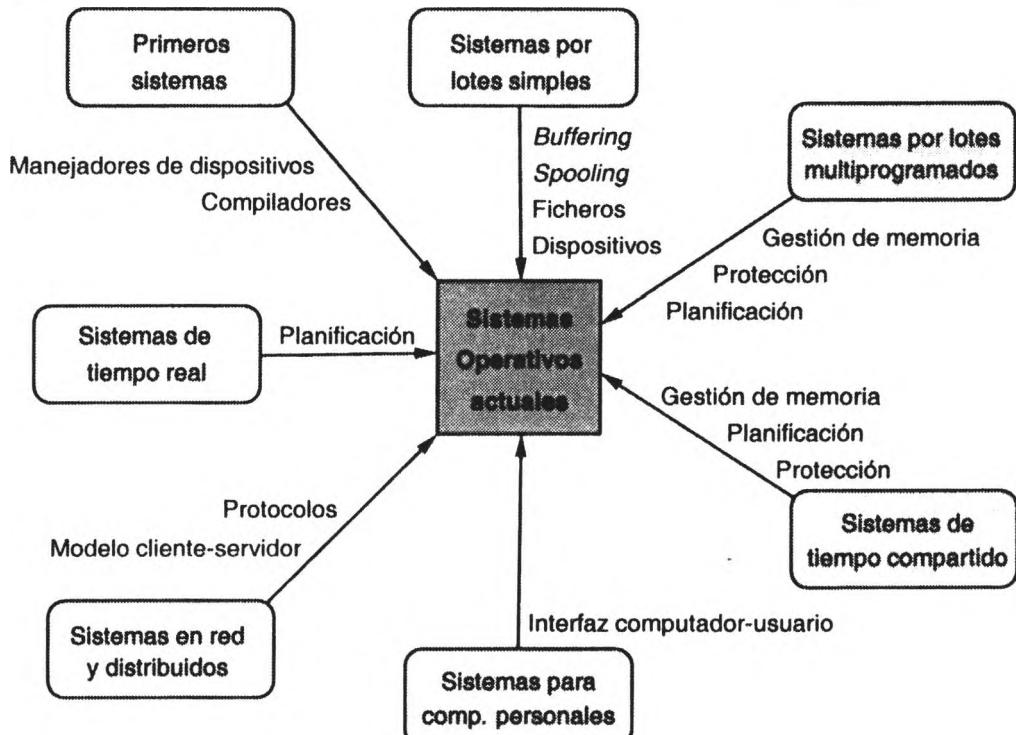
El tercer elemento a proteger es la CPU; en este caso se debe evitar que un programa de usuario no devuelva el control de la CPU al sistema, impidiendo que se ejecuten otros programas. Esto podría ocurrir, por ejemplo, si entrara en un bucle infinito. La protección de la CPU se puede conseguir introduciendo un temporizador que produzca una interrupción después de un período de tiempo determinado. Al producirse la interrupción toma el control el sistema operativo, y de esta forma nos aseguramos de que un programa de usuario nos devuelva el control de la CPU. Por supuesto, al igual que en el caso anterior, para que esta protección sea efectiva debemos disponer del modo dual de operación, que nos va a permitir definir como privilegiadas aquellas operaciones que actúan sobre el temporizador.

## 1.7 Resumen

Dentro de un sistema de computación, uno de los elementos más importantes es el sistema operativo. El objetivo de éste es abstraer los aspectos complejos del sistema ofreciendo una máquina virtual, donde el usuario puede trabajar en un entorno más cómodo, sin necesidad de conocer el funcionamiento del hardware.

Los sistemas operativos actuales han heredado características de todos los que han surgido a lo largo de la historia: por lotes, multiprogramados, de tiempo compartido, etc. Esto se puede ver en la figura 1.9.

Uno de los primeros aspectos que incorporaron los sistemas operativos fueron



**Figura 1.9:** Evolución de los sistemas operativos

los manejadores de dispositivos. Con ellos se consigue uno de los objetivos, abstractar la capa hardware con el fin de hacer el sistema de computación más fácil de manejar.

Los sistemas por lotes simples incorporan dos técnicas, el *buffering* y el *spooling*, cuyo objetivo es solapar las operaciones de la CPU con las de E/S, con el fin de aumentar el rendimiento del sistema y mantener a la CPU inactiva el menor tiempo posible. Además, con la aparición de los discos surgen los ficheros y los denominados dispositivos lógicos, que permiten abstractar aún más el hardware.

Sin embargo, el gran avance se va a producir en los sistemas por lotes multiprogramados, que consiguen mantener en memoria varios trabajos a la vez, con la posibilidad de intercalar sus ejecuciones cuando tengan que esperar. Esto introduce una serie de cambios en el sistema operativo, debido a la necesidad de administrar la memoria para alojar los distintos trabajos, planificar los distintos recursos del sistema (incluida la CPU) entre todos los programas que se están ejecutando.

Con la aparición de los sistemas operativos de tiempo compartido y las terminales aparecen los primeros sistemas multiusuario, en los que varios usuarios pueden trabajar interactivamente con una máquina.

El descenso en el precio del hardware permitió la construcción de ordenadores personales. Los sistemas operativos para estos ordenadores han aprovechado las características de los sistemas construidos para los grandes computadores.

Los sistemas de tiempo real se pueden clasificar en dos tipos: duros y suaves. Los primeros se utilizan como dispositivos de control en aplicaciones dedicadas, y tienen unos requisitos de tiempo bien definidos, que deben cumplirse ya que de otro modo el sistema fallará. Los sistemas de tiempo real suaves son menos restrictivos en cuanto a estas limitaciones de tiempo.

Los sistemas paralelos tienen más de un procesador que comparten el bus, la memoria y diversos dispositivos. Estos sistemas proporcionan un aumento del rendimiento y una mayor fiabilidad.

Los sistemas operativos en red permiten comunicar diversas máquinas pudiendo los usuarios de éstas utilizarlas de forma remota, acceder a sus recursos, etc. Los sistemas operativos distribuidos permiten hacer esto mismo pero de una forma transparente a los usuarios. Una forma de conseguirlo es mediante lo que se denomina el modelo cliente-servidor.

## 1.8 Ejercicios

1. ¿Cuáles son las funciones principales de un sistema operativo?
2. ¿Por qué se considera al sistema operativo como un componente de los programas del sistema? ¿En qué se diferencian de los programas de aplicación?
3. ¿Qué es el monitor residente? ¿Cuál es su función?
4. ¿Cuál es la utilidad del *buffering* y el *spooling*?
5. ¿Qué diferencias existen entre los distintos métodos existentes (operaciones fuera de línea, *buffering* y *spooling*) para el solapamiento de las operaciones de la CPU con las de E/S?
6. Explique la necesidad de proteger la memoria en un sistema por lote multi-programado.
7. ¿Cuál es la utilidad del modo dual?
8. ¿Qué diferencias existen entre un sistema distribuido y un sistema en red?

9. ¿Qué incorpora la aparición de los sistemas de tiempo real al diseño de los sistemas operativos actuales?
10. Razone por qué los avances en los sistemas operativos están ligados a los avances en el hardware (por ejemplo, el *spooling* con la aparición de los discos).



## Estructura y funciones de los sistemas operativos

---

El sistema operativo es un software muy complejo e importante para los sistemas de computación. Veremos como podemos estructurarlo para facilitar su diseño e implementación. Relacionada con esta estructura están las distintas funciones que realiza, con el objetivo de abstraer el hardware y gestionar los recursos del sistema. Por este motivo, el sistema operativo proporciona un conjunto de servicios. En este capítulo veremos cuáles son y las distintas formas de implementarlos.

### 2.1 Funciones y componentes de los sistemas operativos

Para conseguir los objetivos principales de un sistema operativo, gestionar los recursos y abstraer el hardware, éste realiza una serie de funciones que se encargan de proporcionar un ambiente para la ejecución de programas. Éstas pueden variar de un sistema a otro, sin embargo, es posible encontrar o diferenciar las siguientes:

**Gestión de los procesos y los recursos** Un programa cuando se ejecuta se convierte en uno o más **procesos**. El concepto de proceso es clave dentro del sistema operativo, de ahí que ésta sea la función principal del sistema. Los recursos son elementos que necesitan los programas para poder ejecutar-

se. El sistema define el ambiente de ejecución de los procesos, permitiendo que éstos utilicen los distintos recursos y los compartan. Por lo tanto, se encargará de la creación, eliminación, sincronización, comunicación entre procesos, asignación de recursos, etc. El componente del sistema encargado de realizar esta función es el denominado **administrador de procesos**.

**Gestión de la memoria** La memoria principal es un recurso necesario para que un programa pueda ejecutarse. El sistema operativo, mediante el **administrador de la memoria**, se encarga de asignar memoria a los procesos de usuario, y de controlar qué zona ocupan.

**Gestión del sistema de E/S** El **administrador del sistema de E/S** permite al usuario realizar operaciones de E/S sin conocer las particularidades de los diferentes dispositivos. Éste es uno de los objetivos principales del sistema operativo: abstraer el hardware.

**Gestión de los ficheros** Se encarga de almacenar la información en los dispositivos de almacenamiento, abstrayendo las características de éstos en un concepto lógico como el **fichero**. Esta función se realiza a través del **administrador de ficheros** que se encarga de su creación, borrado, lectura, etc.

**Gestión de la red** Esta función dependerá del tipo de sistema operativo. En el caso de los sistemas distribuidos, es el propio sistema el que debe encargarse de la comunicación a través de la red de los programas de usuario que se ejecutan en distintos equipos. En sistemas no distribuidos, esta función la realiza un software independiente del sistema.

**Protección del sistema** Un sistema de computación debe ser seguro, no sólo en cuanto a los usuarios que lo puedan utilizar, sino también a la seguridad en la ejecución de los programas. Así, se deben proteger las zonas de memoria de los distintos procesos de usuario, el acceso a los diferentes recursos, etc. Esta función viene realizada por el denominado **sistema de protección**.

**Interfaz de órdenes** Esta función no tiene por qué formar parte del sistema operativo. Por ejemplo, los sistemas de tiempo real duros suelen carecer de ella. Sin embargo, en sistemas interactivos es una de las más importantes, debido a que es la interfaz entre el usuario y el sistema. En algunos sistemas operativos, como UNIX, esta función no forma parte del núcleo y la realiza un software especial que se ejecuta en el momento en que el usuario entra al sistema. Éstos son los distintos *shell* como el bash, Bourne, C, etc, permitiendo al usuario escoger una u otra interfaz. En otros sistemas, como MS-DOS, esta función está integrada dentro del propio sistema operativo en el denominado **intérprete de órdenes**, sin posibilidad de cambiarla.

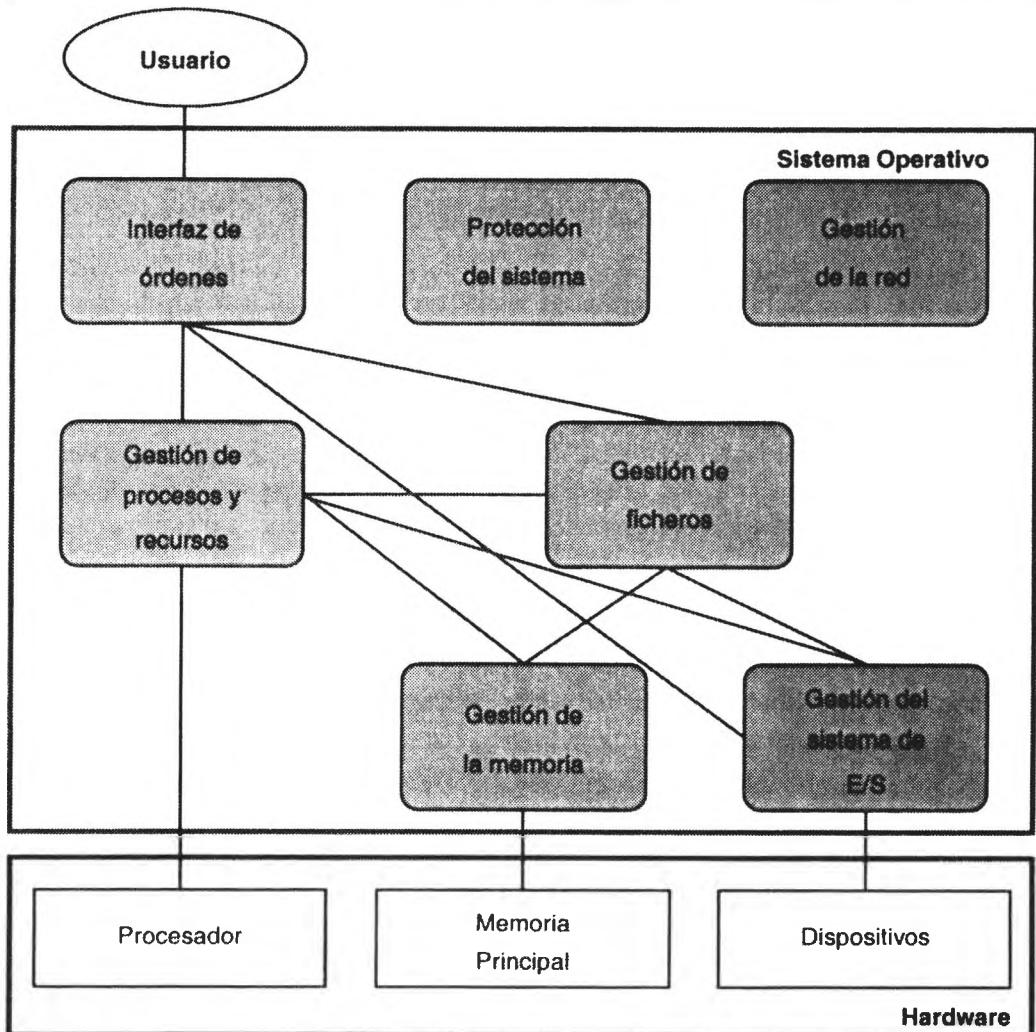


Figura 2.1: Funciones del sistema operativo

La figura 2.1 muestra las relaciones entre las distintas funciones y las correspondientes abstracciones de las que se encargan. Podemos ver como el hardware queda oculto por el sistema operativo, siendo la interfaz de órdenes la relación entre el sistema y el usuario. Por claridad, se han omitido las relaciones de la función de protección del sistema, debido a que afectan a todos los módulos.

## 2.2 Solicitud de servicios

Algunas de las funciones que realiza el sistema operativo reciben el nombre de **servicios** y pueden ser solicitadas por los programadores. Éstos nos permiten acceder al hardware y ciertas estructuras software del sistema de una forma transparente, facilitando de este modo la labor de programación.

Los servicios proporcionados por el sistema operativo pueden solicitarse mediante dos técnicas, las **llamadas al sistema** y el **paso de mensajes**. La utilización de una u otra dependerá de la estructura que presente el sistema como se verá en el apartado 2.3. Ambos mecanismos proporcionan una interfaz entre los procesos y el sistema operativo.

### 2.2.1 Llamadas al sistema

Muchos sistemas operativos proporcionan unas instrucciones especiales que se denominan llamadas al sistema. Algunos sistemas sólo permiten hacer estas peticiones desde el lenguaje ensamblador, mientras que otros, como UNIX, permiten realizarlas desde un lenguaje de alto nivel como C. En el programa 2.1 tenemos un ejemplo de cómo se realizan llamadas al sistema en lenguaje C en un sistema UNIX. En él se utilizan las llamadas **read** y **write**. La llamada **read** lee de la entrada estándar y va almacenando los datos leídos en el vector **buf**. Posteriormente, la llamada **write** los escribe en la salida estándar. Desde el punto de vista del usuario, no hay ninguna diferencia entre hacer una llamada al sistema o a una función de biblioteca.

Programa 2.1

Llamada al sistema

```
#include <unistd.h>
#define TAM 255

main()
{
    char buf[TAM];
    int count;

    while ((count = read(STDIN_FILENO, buf, TAM)) > 0)
        write(STDOUT_FILENO, buf, count);
}
```

En la figura 2.2 podemos ver su funcionamiento. Cuando un proceso hace una llamada al sistema provoca una interrupción al sistema operativo, se cambia de

modo usuario a modo supervisor, el sistema determina el servicio solicitado, y llama a la función que se encarga de realizarlo. Cuando finaliza, cambia a modo usuario y devuelve el control al proceso de usuario que realizó la llamada.

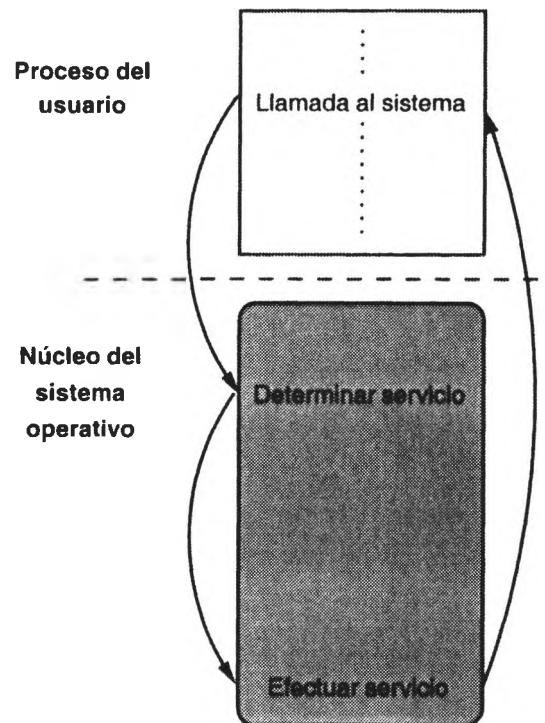
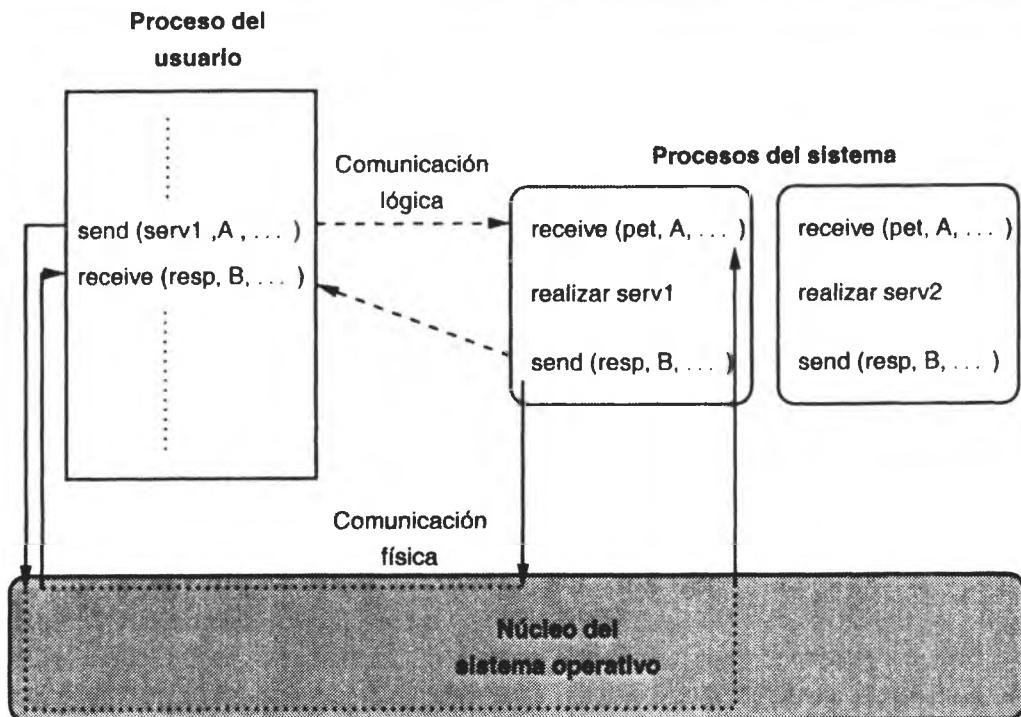


Figura 2.2: Funcionamiento de una llamada al sistema

### 2.2.2 Paso de mensajes

Otro mecanismo para solicitar los servicios del sistema operativo es el paso de mensajes. El procedimiento se puede ver en la figura 2.3. En este caso, el proceso de usuario construye un mensaje que describe el servicio que solicita, éste se envía al sistema operativo mediante la función `send`. Éste se encarga de comprobarlo, cambiar a modo supervisor, entregarlo a un proceso del sistema que realiza el servicio solicitado, y cambiar a modo usuario.

Mientras, el proceso de usuario espera el resultado del servicio solicitado mediante la función `receive`. Cuando el proceso del sistema termina de realizar el servicio, devuelve a través del sistema operativo, mediante la función `send`, la respuesta al proceso de usuario que lo solicitó.



**Figura 2.3:** Funcionamiento del paso de mensajes

Podemos ver que se establece una comunicación lógica entre el proceso del usuario y el proceso del sistema que realiza el servicio. Sin embargo, el mecanismo de comunicación entre ambos va a través del núcleo que se encarga de realizar el intercambio de mensajes.

## 2.3 Estructura de un sistema operativo

Un sistema operativo es un software bastante complejo y grande, cuyo objetivo principal es el de abstraer el hardware haciendo más fácil su utilización. Para ello, el sistema operativo se suele implementar como una colección de módulos que realizan las funciones vistas en el apartado 2.1.

La labor del diseñador del sistema consiste precisamente en determinar cómo se van a diseñar los módulos que implementan las funciones del sistema, con objeto de satisfacer los requisitos mínimos de corrección, rendimiento, mantenimiento, flexibilidad y transportabilidad. No existe un mecanismo que permita diseñar e

implementar de forma óptima un sistema operativo.

El núcleo es la parte del sistema operativo que se ejecuta en modo supervisor, y se puede decir que es el verdadero sistema operativo. Existen dos formas básicas de estructuración de éste. Una consiste en que incluya todas las funciones y servicios del sistema operativo; es lo que se denomina **núcleo complejo**. La segunda consiste en que el núcleo sea lo más pequeño posible, y la mayor parte de los servicios sean proporcionados por los denominados procesos del sistema, en este caso hablamos de un **núcleo mínimo**.

Como puede verse en la clasificación de la figura 2.4, en ambos casos existen distintas posibilidades a la hora de estructurar los componentes que lo forman. Así, para los sistemas que presentan un núcleo complejo podemos distinguir a los que no tienen una estructura interna, como es el caso de la estructura monolítica, y los que sí la tienen, en este segundo caso se trata de un núcleo modular.

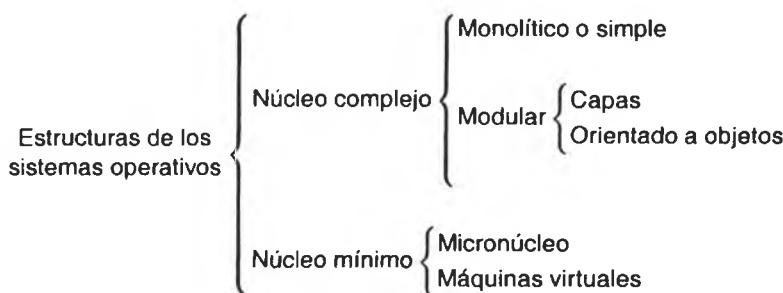


Figura 2.4: Clasificación de los sistemas operativos según su estructura

No existen reglas que determinen cuando es mejor utilizar un núcleo complejo o mínimo. Sin embargo, es posible encontrar ciertas diferencias entre los dos modelos. Así, cuando se diseña un sistema con un núcleo complejo, podemos verlo como un conjunto de funciones, de tal modo, que los procesos de usuario se ejecutan en modo usuario, y cuando solicitan un servicio lo hacen a través de las llamadas al sistema, pasando en ese momento a ejecutar código del sistema operativo en modo núcleo.

Por otro lado, un sistema que presenta un conjunto mínimo de servicios en su núcleo, requiere que el resto de los servicios sean proporcionados mediante procesos del sistema. En este caso, los procesos de usuario solicitan los servicios a los procesos del sistema mediante el paso de mensajes, siendo el núcleo el encargado del intercambio de mensajes.

Las diferencias entre los dos modelos tienen implicaciones en el rendimiento del

sistema. Los sistemas operativos basados en llamadas al sistema son más eficientes que los basados en paso de mensajes, ya que éstos requieren crear un mensaje, copiarlo al espacio de direcciones del proceso del sistema y pasar a ejecutar este proceso. La respuesta implica hacer los mismos pasos a la inversa. Mientras que en el caso de una llamada al sistema, el proceso de usuario al hacer la llamada provoca un cambio de modo pasando a ejecutar código del núcleo; al finalizar la ejecución de éste se produce otro cambio de modo.

Por otro lado, los sistemas basados en paso de mensajes son más flexibles porque permiten que el propio usuario pueda añadir y modificar servicios del sistema, ya que esto no implica modificar el código del núcleo. Además es más fácil diseñarlos para que puedan tomar el control de la máquina en situaciones especiales, tales como la ejecución temporal de tareas. Por este motivo, en los sistemas operativos basados en funciones suele existir un conjunto de procesos del sistema que se encargan de manejar estas situaciones especiales; podemos citar como ejemplo los sistemas **UNIX** que poseen los llamados **demonios** (*daemons*).

### 2.3.1 Estructura simple o monolítica

Los sistemas que presentan esta estructura se componen de un único módulo lógico que realiza todas las funciones. Aunque se aplican algunas técnicas de programación modular, el problema es distinguir los distintos componentes del sistema (administrador de procesos, de la memoria, etc.), ya que todos están relacionados. De esta forma, si queremos aislar el administrador de procesos es difícil debido a que suele estar relacionado con el administrador de memoria; éste a su vez necesita conocer el estado de las operaciones de E/S, etc. La característica principal de estos sistemas operativos es que los servicios se solicitan mediante llamadas al sistema.

Este tipo de sistema operativo está escrito como un conjunto de procedimientos. Cada uno tiene una interfaz bien definida en términos de parámetros y resultados, y puede llamar a cualquier otro que le proporcione algo que necesite. No existe ninguna ocultación de información ya que todos los procedimientos son visibles a los demás. Para construir el programa objeto del sistema operativo, en primer lugar se compilan los procedimientos individuales, enlazándolos posteriormente en un solo fichero objeto.

Estos sistemas operativos tienen la ventaja de estar muy adaptados al hardware, por lo que consiguen un alto rendimiento. Sin embargo, sacrifican otros aspectos del diseño como la facilidad de mantenimiento, portabilidad, etc.

Hoy en día, numerosos sistemas comerciales presentan esta estructura, debido a que empezaron siendo sistemas pequeños, simples y limitados. Sin embargo, gracias a su éxito fueron creciendo y superando los objetivos inicialmente establecidos. Ejemplos de estos sistemas son UNIX y MS-DOS.

### 2.3.1.1 El sistema operativo UNIX

Uno de los ejemplos más representativos de la estructura monolítica es el sistema operativo UNIX. En él es posible distinguir dos partes bien diferenciadas: el núcleo y el resto de programas del sistema. En la figura 2.5 podemos ver su estructura.

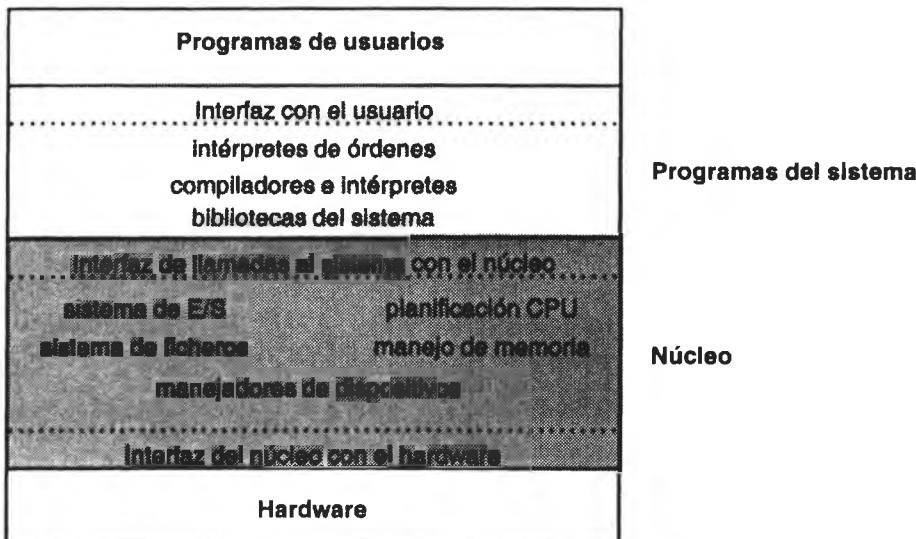


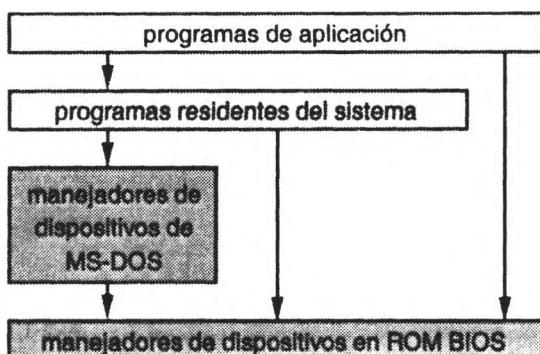
Figura 2.5: Estructura del sistema UNIX

El núcleo es la parte más importante del sistema UNIX. Se sitúa entre el hardware y la interfaz de llamadas al sistema. En él se encuentran las funciones de gestión de memoria, de E/S, planificación de la CPU, así como un conjunto de manejadores de dispositivos, que con el paso del tiempo ha ido aumentando. Por tanto, en un solo nivel existe una gran cantidad de funciones que imposibilita organizarlas. Como consecuencia de la evolución sufrida por UNIX, el núcleo ha pasado de ocupar unos 40 KiB a tener cerca de 1 MiB, aunque en algunas máquinas ocupa varios MiB.

El sistema **UNIX** a pesar de poseer una estructura monolítica, suele tener una serie de procesos del sistema. Éstos son los demonios, que se encargan de tomar el control de la máquina en situaciones especiales y permiten tener más seguridad en el sistema.

### 2.3.1.2 El sistema operativo MS-DOS

Otro claro ejemplo de este tipo de estructura es **MS-DOS**. Este sistema fue diseñado e implementado originalmente por unas pocas personas que no suponían que llegaría a ser tan popular. Fue escrito para proporcionar la máxima funcionalidad en el menor espacio, por lo que no fue dividido en módulos cuidadosamente. La figura 2.6 muestra su estructura.



**Figura 2.6:** Estructura de MS-DOS

Aunque **MS-DOS** tiene cierta estructura, sus interfaces y módulos funcionales no están bien separados. Por ejemplo, los programas de aplicación son capaces de acceder a las rutinas básicas de E/S para escribir directamente en la pantalla y en las unidades de disco. Tal libertad hace al sistema **MS-DOS** vulnerable a los programas erróneos, causando la caída completa del sistema o el borrado del disco cuando un programa de usuario falla. Por supuesto, **MS-DOS** está también limitado por el hardware para el que fue diseñado, dado que el microprocesador Intel 8088 no proporciona modo dual, por lo que los diseñadores de **MS-DOS** tuvieron que dejar el hardware base accesible.

### 2.3.2 Estructura modular

Los sistemas con una estructura monolítica presentan ciertos inconvenientes, como la dificultad de mantenerlos y actualizarlos. Sin embargo, un objetivo de diseño de cualquier software debe ser la facilidad de mantenimiento. Por este motivo, se debe buscar una estructura interna que nos permita cumplir con ese objetivo.

Para facilitar tanto el diseño como la implementación de los sistemas operativos, se puede utilizar una estructura modular. Consiste en dividirlo en unidades lógicas definiendo bien las interfaces entre ellas.

Un sistema que presente estructura modular es más fácil de mantener y actualizar. Sin embargo, la modularización conlleva un coste en pérdida de rendimiento del sistema respecto a la estructura monolítica. Por este motivo, es necesario llegar siempre a un compromiso entre modularización y rendimiento.

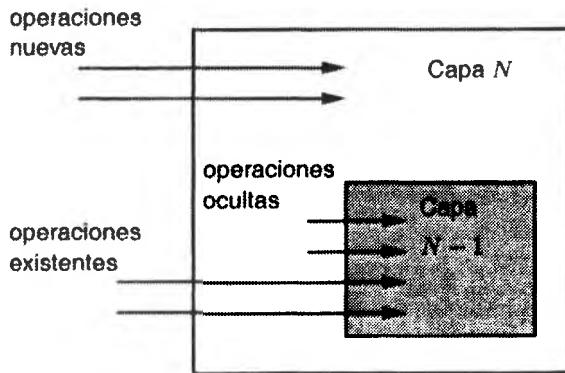
La modularización de un sistema se puede llevar a cabo de muchas formas; aquí consideraremos dos enfoques: la estructura en capas y la orientada a objetos.

#### 2.3.2.1 Estructura en capas

Una forma de estructurar el sistema operativo es mediante la **estratificación**, que consiste en romper el sistema en estratos, niveles o capas, cada uno de ellos construido sobre los anteriores. Una capa de un sistema operativo es una implementación de un objeto abstracto que permite la encapsulación de datos y las operaciones que pueden manipular esos datos. En un sistema de este tipo, la capa más baja (capa 0) es el hardware, y la más alta (capa  $N$ ) es la interfaz con el usuario.

En la figura 2.7 aparece una capa típica de un sistema operativo, que consta de algunas estructuras de datos y un conjunto de funciones que pueden ser llamadas por capas de nivel superior. Esta capa, a su vez, puede utilizar operaciones de las de nivel inferior, sin importarle cómo están implementadas.

La principal ventaja de los sistemas por capas es la encapsulación. Las capas se seleccionan de tal forma que cada una sólo utiliza las funciones (operaciones) y servicios de las de nivel inferior. Esto hace mucho más fácil la depuración y verificación del sistema. El primer nivel puede ser depurado sin tener en cuenta el resto del sistema, puesto que, por definición, sólo usa el hardware para implementar sus funciones. Una vez que el primer nivel ha sido depurado, se pasa a depurar el siguiente independientemente de los demás, y así sucesivamente. Si se encuentra un error durante la depuración de un nivel particular, sabemos que el error debe estar en él, puesto que los inferiores han sido ya depurados. Por tanto,



**Figura 2.7:** Capa de un sistema operativo

el diseño y la implementación de un sistema se simplifican cuando se rompe en niveles.

La mayor dificultad que presenta este método es la definición de los niveles. Es necesario hacer una planificación cuidadosa, ya que cada capa sólo puede usar aquellas funciones que están por debajo de ella. Esta limitación ha hecho que en los últimos años se haya abandonado un poco este tipo de estructura. En la actualidad se tiende a utilizar menos capas con más funcionalidad, proporcionando la mayoría de las ventajas del código modular y evitándose los problemas de definición de capas y su interacción.

El primer sistema operativo que se diseñó siguiendo este método fue el sistema THE en la *Technische Hogeschool Eindhoven*, que estaba formado por cinco capas, como se muestra en la figura 2.8.

<b>Capa 5: Programas de usuario</b>
<b>Capa 4: Buffering para dispositivos de E/S</b>
<b>Capa 3: Manejador del dispositivo de consola</b>
<b>Capa 2: Manejo de la memoria</b>
<b>Capa 1: Planificación de la CPU</b>
<b>Capa 0: Hardware</b>

**Figura 2.8:** Estructura en capas del sistema THE

Otro ejemplo de sistema en capas es OS/2, descendiente directo de MS-DOS, que fue creado para vencer las limitaciones de éste. El sistema OS/2 añade la multitarea, así como otras características nuevas. Debido a esta complejidad añadida y al hardware más potente para el que fue diseñado, OS/2 fue implementado

de una forma más estratificada que MS-DOS. En la figura 2.9 puede verse claramente que no está permitido el acceso directo de los usuarios al hardware, ya que éste proporciona el modo de operación dual, dándole al sistema operativo mayor control sobre el hardware.

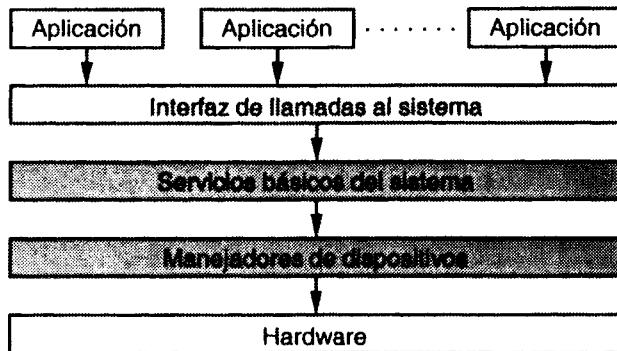


Figura 2.9: Estructura en capas de OS/2

### 2.3.2.2 Sistema operativo orientado a objetos

La aparición de la metodología orientada a objetos ha permitido el desarrollo de este tipo de sistemas operativos. Éstos se caracterizan porque todos sus elementos se representan como objetos, permitiendo tener una serie de clases representativas, tales como dispositivos hardware, procesos, ficheros, etc.

Cada objeto presenta una serie de atributos y un conjunto de operaciones sobre ellos que definen su comportamiento y a la vez que determinan su interfaz con otros objetos. Las posibles relaciones entre los objetos reciben el nombre de **capacidades**, donde se indican los tipos de operaciones que se pueden realizar sobre ellos.

El núcleo del sistema operativo se encarga de mantener los distintos objetos existentes en el sistema, además de controlar las operaciones que actúan sobre ellos. Así, cuando un objeto necesita interactuar con otro, el núcleo comprueba que el objeto dispone de esa capacidad de relación.

Este tipo de sistemas presenta todas las ventajas de una metodología orientada a objetos como son:

- Encapsulación de las estructuras de datos en objetos.

- Ocultamiento de las implementaciones de las operaciones.
- Jerarquía de clases, permitiendo la herencia entre ellas.
- Reutilización de código.
- Acceso uniforme a los objetos, sean locales o remotos.

Un ejemplo de un sistema operativo orientado a objetos es Choices<sup>1</sup>. Surge de una investigación experimental, donde se emplea tanto un diseño como un lenguaje orientado a objetos.

La característica principal de este sistema es definir un marco de trabajo de objetos (*framework*), donde se define su estructura. En este marco se definen las clases básicas del sistema y sus relaciones, creando un entorno generalizado donde actúa el sistema. Posteriormente, con ayuda de la herencia, va creando submarcos, que describen cómo actúan los diferentes elementos. La herencia juega un papel importante en este sistema, porque permite adaptarlo fácilmente a cualquier hardware. Así, cuando se lleva a una nueva plataforma, sólo es necesario redefinir las implementaciones para dicho hardware, manteniendo las clases superiores y sus relaciones intactas.

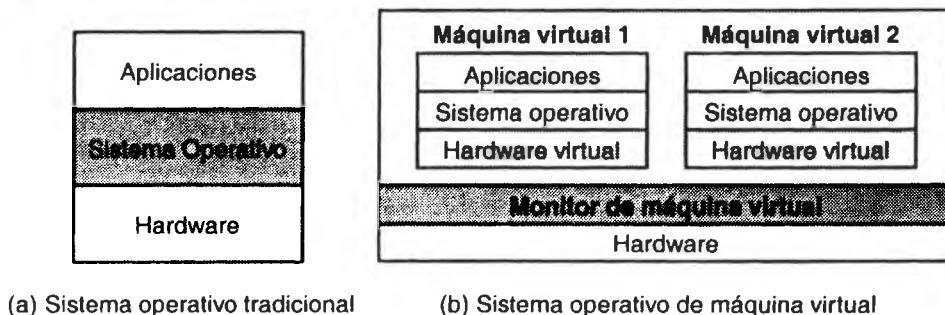
### 2.3.3 Máquinas virtuales

El concepto de máquina virtual surge con el sistema VM/370 de IBM en 1972. La idea principal de la máquina virtual es la de permitir ejecutar varios sistemas operativos simultáneamente sobre el mismo hardware. Para ello, separa las dos funciones básicas que realiza un sistema de tiempo compartido: multiprogramación y abstracción del hardware.

El software del sistema ubicado encima del sistema operativo, así como los programas de usuario usan tanto los servicios del sistema como instrucciones hardware sin llegar a diferenciarlas, dando la ilusión de tener contacto directo con éste. Esta idea es utilizada por los sistemas de máquina virtual. En éstos, el corazón del sistema, conocido como **monitor de máquina virtual**, se ejecuta sobre el hardware y proporciona varias máquinas virtuales al siguiente nivel de software, como se muestra en la figura 2.10. Para ello crea una interfaz de llamadas al sistema que simulan el hardware subyacente. Estas máquinas virtuales no son máquinas extendidas, con ficheros y otras características agradables, sino que son copias exactas del hardware desnudo, incluyendo los modos núcleo/usuario, dispositivos de E/S, interrupciones, y todo aquello que tiene la máquina real.

---

<sup>1</sup>Sistema operativo construido en la Universidad de Illinois



**Figura 2.10:** Sistema operativo tradicional frente a máquina virtual

Debido a que cada máquina virtual es idéntica al hardware verdadero, cada una puede estar ejecutando cualquier sistema operativo que se pudiera ejecutar directamente sobre éste. Así, diferentes máquinas virtuales pueden ejecutar diferentes sistemas operativos.

El concepto de máquina virtual vuelve a estar de moda en los sistemas operativos por distintos motivos. Así, para poder mantener la compatibilidad con antiguos programas de MS-DOS en nuevos equipos, se crea una máquina Intel virtual donde poder ejecutarlos, de forma que sus instrucciones son posteriormente traducidas al procesador nativo. Esto lo emplean fabricantes como Sun.

Recientemente ha surgido VMWare que permite llevar esta tecnología a los ordenadores personales, de forma que un usuario pueda estar ejecutando sistemas como Windows NT y LINUX de forma simultánea.

La idea de máquina virtual ha sido llevada también a los lenguajes de programación. Así, Java se implementa con un compilador que genera un código de salida que se ejecuta en la Máquina Virtual Java (JVM<sup>2</sup>). De este modo, los programas tienen una mayor transportabilidad, ya que para poder ejecutarlo sólo necesitan disponer en el sistema informático de la JVM, independientemente del hardware original sobre el que se ejecute.

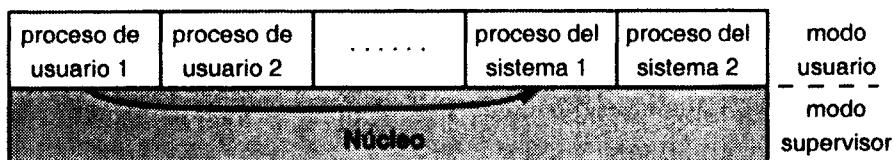
### 2.3.4 Micronúcleo

Una tendencia en los sistemas operativos modernos es llevar más allá la idea de trasladar gran parte del código del sistema operativo a niveles superiores, dejando un núcleo mínimo. El método usual consiste en implementar la mayor parte de las

<sup>2</sup> Java Virtual Machine

funciones del sistema operativo en procesos del sistema. Para pedir un servicio, como por ejemplo la lectura de un bloque de un fichero, un proceso de usuario envía una petición al proceso que realiza el trabajo y éste le envía la respuesta. De esta forma, la petición de los servicios al sistema no se realiza mediante una llamada al sistema, sino mediante el paso de mensajes.

En este modelo, que se muestra en la figura 2.11, todo lo que el núcleo hace es manejar la comunicación entre los procesos de usuario y del sistema. Al dividir el sistema operativo en partes, cada una de las cuales se encarga del manejo de una faceta de éste, como el servicio de ficheros, procesos, terminales, etc., cada parte del sistema se hace más pequeña y fácil de manejar.



**Figura 2.11:** Estructura de micronúcleo

La imagen que aparece en la figura 2.11 de un núcleo que maneja sólo el transporte de mensajes entre procesos de usuario y procesos del sistema no es completamente real. Algunas funciones del sistema operativo (tales como la carga de órdenes en los registros de los dispositivos de E/S) son difíciles, si no imposible, de hacer desde programas en el espacio del usuario. Una forma de tratar este problema es tener algunos procesos críticos del sistema (por ejemplo, los manejadores de los dispositivos de E/S) que se ejecuten en modo supervisor, con acceso completo a todo el hardware, pero que se comunican con otros procesos usando el mecanismo normal de mensajes.

Este tipo de estructura también se suele conocer como cliente-servidor; a los procesos de usuario que piden un servicio se les denomina clientes, mientras que los procesos que realizan el servicio son los servidores.

## 2.4 El sistema operativo LINUX

LINUX es una versión de UNIX para procesadores Intel 386 y 486, que se distribuye de forma gratuita. Fue desarrollado principalmente por Linus Torvalds de la Universidad de Helsinki, y posteriormente se ha ido desarrollando con la ayuda de muchos programadores a través de Internet. El núcleo de LINUX no utiliza código de AT&T ni de ningún otro propietario, y la mayor parte del *software* disponible

para LINUX ha sido desarrollado por el proyecto GNU de la *Free Software Foundation*. Además, programadores de todo el mundo han contribuido al crecimiento del volumen de *software* disponible para LINUX. Actualmente muchas empresas están portando sus productos para LINUX, como ejemplos podemos citar el procesador de textos Word-Perfect, o el sistema de gestión de bases de datos ORACLE.

LINUX fue desarrollado originalmente como un entretenimiento por Linus Torvalds. Estaba inspirado por Minix, un pequeño sistema UNIX realizado por Andrew Tanenbaum, y las primeras discusiones sobre LINUX se produjeron en el grupo de noticias de USENET `comp.os.minix`. Estas discusiones estaban relacionadas con el desarrollo de un sistema UNIX pequeño para aquellos usuarios de Minix que querían más.

El 5 de octubre de 1991, Linus anunció su primera versión «oficial» de LINUX, la versión 0.02. En marzo de 1992 apareció la versión 1.0. Las distintas versiones que surgen de LINUX se caracterizan por representarlas mediante tres números separados por puntos. El primero es el **número principal** (*major version number*), seguido del **número secundario** (*minor version number*). Este segundo número si es impar indica que se trata de una versión beta y si es par corresponde a una versión final. Por último, el tercer número (*patch level*) es el que indica la revisión de la versión actual del núcleo definida por los dos números anteriores.

Hoy día, LINUX es un clónico de UNIX capaz de ejecutar X Window, TCP/IP, Emacs, UUCP, correo electrónico y *software* de news. Casi todos los paquetes de *software* libre han sido transportados a LINUX, y también está apareciendo *software* comercial. El núcleo de LINUX soporta cada vez más *hardware* y ya está disponible para otros procesadores diferentes de los de Intel.

Existen muchas distribuciones de LINUX, cada una con sus peculiaridades y sus propios mecanismos de instalación. El único elemento común a todas las distribuciones es el núcleo del sistema operativo, que se desarrolla de forma coordinada y con actualizaciones sistemáticas. LINUX, al igual que su predecesor, presenta un núcleo monolítico. No obstante, presenta cierta estructura modular.

#### 2.4.1 Componentes del núcleo

En la figura 2.12 se muestra un diagrama de bloques del núcleo, en el que aparecen todos los módulos que lo componen y las relaciones que se establecen entre ellos.

Esta figura muestra tres niveles: usuario, núcleo y hardware. El núcleo proporciona un conjunto de servicios al resto de aplicaciones o procesos de usuario que pueden ser solicitados a través de las llamadas al sistema. Éstas constituyen la interfaz entre los programas de usuario y el núcleo. En LINUX se definen un

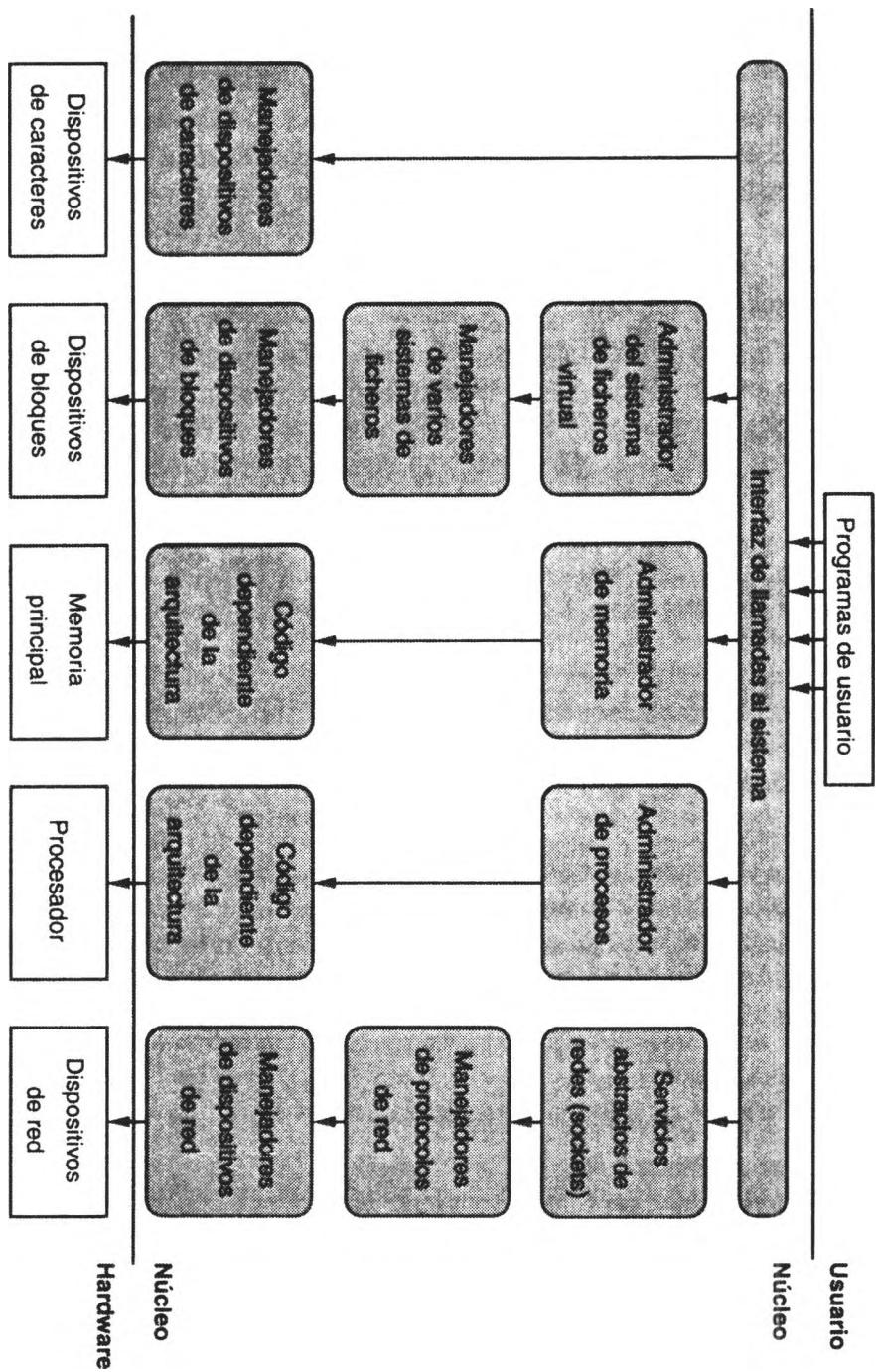


Figura 2.12: Diagrama del núcleo de un sistema Linux

total de 163 llamadas al sistema.

Las partes más importantes del núcleo son el administrador de la memoria y el de procesos. El primero es el encargado de asignar la memoria y áreas de intercambio a los procesos. De este modo, es el responsable de ubicar un proceso en memoria principal o en memoria secundaria, según las necesidades del sistema.

Por otro lado, el administrador de procesos es el encargado de la creación de éstos, permitiendo la multiprogramación, mediante la conmutación del procesador entre los procesos activos.

LINUX proporciona varios sistemas de ficheros, por lo que el núcleo contiene un manejador específico para cada uno de ellos. La similitud entre las operaciones que éstos realizan permiten agruparlas en el denominado **sistema de ficheros virtual** (VFS<sup>3</sup>). Cuando un proceso intenta realizar una operación sobre un fichero, la petición va a través del VFS que la desvía al manejador específico del sistema de ficheros.

En el nivel más bajo, el núcleo contiene un manejador específico para cada tipo de dispositivo soportado. Sin embargo, las similitudes entre ellos hace posible establecer clases generales de manejadores. Esto permite que los miembros de una clase posean la misma interfaz con el núcleo, diferenciándose en la manera de implementarla.

Otros servicios proporcionados por el núcleo tienen propiedades similares. Así, los distintos protocolos de redes han sido abstraídos en una única interfaz programable, la biblioteca *BSD socket*. Aunque LINUX no es un sistema distribuido, las funciones de gestión de la red están incorporadas en el núcleo en lugar de ofrecerse como un software independiente del sistema.

LINUX utiliza una serie de procesos del sistema, denominados demonios. Éstos se encargan de tomar el control de la máquina en situaciones especiales. Entre ellos se encuentran el demonio de impresión, de paginación, etc.

El sistema operativo LINUX permite configurar el núcleo adaptándolo a las características del sistema. Se puede generar un **núcleo dinámico**, en el que ciertas partes del sistema se cargarán en memoria cuando se necesiten y, en caso contrario, se descargarán. Éstos son los llamados **módulos** que permiten tener un núcleo más flexible y de menor tamaño.

---

<sup>3</sup> Virtual File System.

## 2.5 Resumen

Un sistema operativo realiza numerosas funciones que definen un ambiente de ejecución de los programas. Éstas se encargarán de gestionar todos los recursos del sistema, y realizar una abstracción del hardware, permaneciendo oculto a los usuarios.

A la hora de diseñar un sistema operativo, es muy importante determinar su estructura interna, ya que ésta influye en la forma de organizar las distintas funciones del sistema, así como el mecanismo de solicitar los servicios.

Éstos pueden solicitarse mediante dos mecanismos: paso de mensajes y llamadas al sistema. Los primeros son más lentos pero más flexibles a la hora de diseñar nuevos servicios. Las llamadas al sistema son funciones propias del núcleo del sistema operativo, por lo que introducir un nuevo servicio requiere disponer del código fuente y un gran conocimiento de cómo se encuentra diseñado.

## 2.6 Ejercicios

1. ¿Qué funciones suelen proporcionar los sistemas operativos?
2. ¿Qué diferencias hay entre los servicios y las llamadas al sistema?
3. ¿Cuál es la diferencia entre un núcleo complejo y otro mínimo?
4. ¿Qué significa que el sistema operativo **LINUX** presenta un núcleo dinámico?
5. Razone la siguiente afirmación: «En un núcleo con estructura de micronúcleo los servicios se solicitan por paso de mensajes.»
6. ¿Qué ventajas e inconvenientes presenta un núcleo con estructura modular frente a uno con estructura monolítica?
7. ¿Cuáles son las características de una capa de abstracción de un sistema estructurado en capas?
8. ¿Qué es el monitor de máquina virtual
9. ¿Cuál es la ventaja principal que presenta un sistema con estructura de máquina virtual para los usuarios?
10. ¿Qué características presenta el sistema operativo **LINUX**?

---

Parte 2

# PROCESOS

---



# Capítulo 3

## Descripción y control de procesos

---

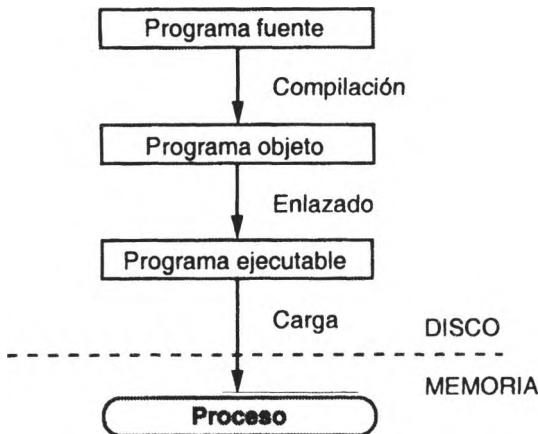
La unidad fundamental de un sistema operativo es el proceso. Cuando un usuario ejecuta un programa se convierte en un proceso. Éste es el que va a ejecutarse en la CPU y va a hacer uso de los distintos recursos del sistema. En este capítulo veremos cómo se representa un proceso en el sistema, qué estructuras se necesitan para gestionarlo, y la evolución que sufre desde que se crea hasta que termina. Finalmente, se estudiará una nueva tendencia en los sistemas operativos: los hilos.

### 3.1 ¿Qué es un proceso?

El término proceso fue utilizado por primera vez por los diseñadores del sistema Multics en los años sesenta. Desde entonces, del término proceso, que a veces se utiliza como sinónimo de tarea<sup>1</sup>, se han dado muchas definiciones. Nosotros

<sup>1</sup>En la bibliografía sobre sistemas operativos no existe unanimidad para designar el concepto de una secuencia de código en ejecución. Muchos libros utilizan el término proceso mientras que los manuales de las empresas y los programadores de sistemas se refieren a tarea. La razón de esta dualidad puede ser debida a que el término proceso puede ser algo confuso cuando se utiliza para describir un entorno de multiprogramación, ya que un sistema de multiprocesamiento es entendido normalmente como un sistema con varios procesadores, en vez de un sistema con un solo procesador en el que se ejecutan varios procesos concurrentemente. Con el término multitarea se designa un sistema operativo que soporta ejecución concurrente de programas sobre un solo procesador sin soportar necesariamente formas elaboradas de gestión de memoria

nos quedaremos con una primera definición muy simple que es la siguiente: un proceso en un programa en ejecución. En ella estamos resaltando la diferencia entre programa (entidad estática) y proceso (entidad dinámica). Un usuario del sistema no ejecuta procesos, sino programas. Un programa es una colección de instrucciones y datos que se encuentran almacenados en un fichero, es decir, es algo pasivo. Si queremos ejecutarlo, el programa debe pasar previamente por una serie de fases, que pueden verse en la figura 3.1.



**Figura 3.1:** Conversión de un programa fuente en proceso

En ella podemos ver que el programa fuente debe ser previamente compilado para obtener un módulo objeto. Éste debe ser enlazado con otros módulos objeto para formar un único módulo de carga ejecutable. Para poderlo ejecutar es necesario cargarlo en memoria, labor realizada por el cargador. Es en ese momento cuando el programa se convierte en proceso.

Un proceso no sólo consiste en una copia del programa (conjunto de instrucciones y datos), sino que además el sistema operativo le añade cierta información adicional para poder gestionarlo.

A medida que vayamos profundizando en nuestro estudio de los procesos veremos que este concepto es más complejo y sutil de lo que parece a primera vista, y de hecho, abarca dos conceptos separados y potencialmente independientes, uno relativo a la propiedad de los recursos y el otro relativo a la ejecución.

---

y gestión de ficheros. La multiprogramación es un concepto más general que designa a un sistema operativo que proporciona ejecución concurrente de programas, gestión de memoria y gestión de ficheros. Por tanto, podemos decir que un sistema operativo de multiprogramación es también un sistema operativo multitarea, mientras que lo inverso no siempre es cierto.

La definición de sistema operativo dada por Denning, nos da idea de la importancia del concepto de proceso: «Sistema operativo es el software de un sistema informático que asiste al hardware para realizar funciones de gestión de procesos.» Analizando esta definición, podemos ver que los requisitos principales que debe cumplir un sistema operativo multiprogramado están relacionados con la gestión de procesos. Así, una de las funciones principales del sistema operativo es la creación de procesos de usuario. Además, es responsabilidad del sistema operativo la ejecución intercalada de los procesos con objeto de obtener un mayor rendimiento del procesador, siempre que proporcione un tiempo de respuesta razonable (en el capítulo 4 se estudiará en más profundidad este aspecto). Por otro lado, el sistema operativo también se encarga de la asignación de recursos a los procesos con objeto de evitar la aparición de problemas como el interbloqueo (éstos se estudiarán en el capítulo 6). Igualmente, otra de las funciones del sistema operativo es facilitar la comunicación entre los distintos procesos de usuario (como se verá en el capítulo 5).

Dado que los procesos son una pieza clave dentro de los sistemas operativos, vamos a empezar el estudio en detalle de estos últimos, examinando cómo representan y controlan a los procesos.

## 3.2 Vida de un proceso

La vida de un proceso viene delimitada por su creación y su terminación. Durante el período de vida de un proceso, éste pasa por una serie de estados; el cambio de un estado a otro se denomina **transición de estado** o **cambio de estado**, y puede producirse por necesidad del sistema o del propio proceso. De este modo, se puede definir la ejecución de un proceso como la progresión a través de un conjunto de estados.

En cualquier momento, los estados de todos los procesos del sistema y de los recursos (asignados o libres) definen lo que se conoce como **estado global del sistema**. A partir de esta información, el sistema operativo utiliza sus algoritmos de planificación para decidir cómo asignar los recursos disponibles a los procesos que los solicitan, de tal forma que el rendimiento resultante satisfaga los objetivos de diseño.

### 3.2.1 Creación de un proceso

Para la creación de un proceso nuevo, el sistema operativo tiene que realizar una serie de acciones que se pueden dividir en dos etapas:

1. Creación de las estructuras de datos que se necesitan para la administración del proceso.
2. Asignación del espacio de direcciones que va a utilizar el proceso.

La creación de un nuevo proceso puede ser debida a diferentes motivos, que incluso pueden variar de un sistema a otro. Algunos sucesos que pueden dar lugar a la creación de un nuevo proceso pueden ser:

- En un sistema por lotes se crea un nuevo proceso cuando se envía un trabajo a ejecutar.
- En un sistema interactivo se crea un proceso cuando un nuevo usuario quiere acceder a él.
- El sistema operativo puede crear un proceso para proporcionar un servicio a otro, por ejemplo, para imprimir un fichero. Así el proceso que hace la petición puede proseguir su ejecución.
- Un proceso existente puede querer crear otro, por ejemplo para realizar dos acciones en paralelo. Cuando el sistema operativo crea un proceso a instancias de otro, la acción se conoce como **generación de procesos**. Cuando un proceso genera a otro, el generador se conoce como **proceso padre** y el generado como **hijo**. De esta forma se establece una jerarquía entre los procesos.

Hay que hacer notar que en los tres primeros casos, el nuevo proceso se crea por iniciativa del sistema operativo y es transparente al propio proceso, mientras que en el último caso es un proceso de usuario el que lo causa.

### 3.2.2 Terminación de un proceso

En la terminación de un proceso también pueden distinguirse dos etapas: en primer lugar, el sistema operativo le retira el espacio de direcciones del que disponía el proceso, y posteriormente destruye las estructuras de datos que creó para su manejo.

Hay muchas razones por las que un proceso puede terminar, la más inmediata es porque termina su ejecución de forma normal; en este caso, el proceso ejecutará un servicio para indicar que ha terminado. Las otras razones son debidas a errores que se producen durante la ejecución del proceso. Algunos de éstos pueden ser:

- El proceso intenta acceder a una localización de memoria a la cual no tiene permitido el acceso.
- El proceso intenta acceder a un recurso al que no tiene acceso, o intenta hacerlo de una forma improcedente, por ejemplo, intenta escribir en un fichero de sólo lectura.
- El proceso intenta realizar una operación aritmética no permitida.
- El proceso intenta ejecutar una instrucción no válida.
- El sistema operativo o el usuario pueden dar por terminado un proceso, por ejemplo, por la existencia de un interbloqueo.

Algunos sistemas operativos están diseñados de forma que al terminar un proceso terminan todos los que han sido generados por él. De este modo un proceso puede terminar porque lo pide el proceso que lo generó. Esto es útil ya que permite el cierre ordenado del sistema de una forma fácil, matando al proceso inicial que es el padre de todos los demás.

### 3.2.3 Estados de un proceso

A la hora de diseñar un sistema operativo hay que definir un modelo claro del comportamiento que van a tener los procesos en este sistema. Esto es lo que se conoce como **modelo de estados**; que nos indica en qué estados puede encontrarse un proceso, y qué razones pueden hacer que un proceso pase de un estado a otro. Estos cambios de estado vienen impuestos, en gran medida, por la competencia que se establece entre los procesos a la hora de compartir los recursos del sistema. A continuación se describirán dos modelos de estados posibles.

#### 3.2.3.1 Modelo de cinco estados

Podemos definir un modelo simple en el que los procesos pueden estar en cinco estados diferentes. Éstos podrían ser:

**Nuevo** Un proceso que acaba de ser creado pero no ha sido admitido al conjunto de procesos ejecutables por el sistema operativo.

**Listo** Están en este estado aquellos procesos que poseen todos los recursos necesarios para ejecutarse excepto el procesador.

**Ejecución** Se encuentra en este estado el proceso que tiene el control del procesador. En un ordenador con un solo procesador sólo podemos tener un proceso en este estado en un instante dado.

**Bloqueado** Los procesos en este estado no pueden ejecutarse hasta que no ocurra un cierto suceso, como por ejemplo la terminación de una operación de E/S.

**Terminado** Un proceso que ha sido sacado del conjunto de procesos ejecutables por el sistema operativo, bien porque se ha detenido o porque ha sido abortado por alguna razón.

Los estados nuevo y terminado son construcciones útiles para el manejo de procesos. Si consideramos que tanto la creación como la terminación de un proceso se realizan en dos etapas, tal como apuntamos en el apartado 3.2.1; el estado nuevo corresponderá a un proceso que acaba de ser definido, es decir, el sistema operativo ha creado las estructuras de datos necesarias para manejarlo, pero todavía no le ha asignado un espacio de direcciones, es decir, no se ha comprometido a ejecutarlo. De forma similar, un proceso sale del sistema en dos etapas; primero, el proceso termina su ejecución de forma normal o anormal y pasa al estado de terminado. En este punto, el proceso ya no se puede elegir para ser ejecutado. Sin embargo, el sistema operativo mantiene durante un tiempo las estructuras y otra información asociada al proceso. Esto proporciona tiempo para que ciertos procesos extraigan información sobre el proceso terminado. Una vez pasado este tiempo, el sistema operativo puede borrar los datos correspondientes al proceso. Ejemplos de programas que pueden necesitar información sobre los procesos que terminan son los de contabilidad, que controlan el tiempo de CPU y otros recursos utilizados por el proceso con fines de facturación, asimismo programas que monitorizan el uso y rendimiento de la máquina pueden tomar información sobre cada proceso que se ejecuta.

Un **diagrama de transición de estados** es un grafo dirigido, cuyos nodos representan los estados que pueden alcanzar los procesos y las ramas representan los sucesos que hacen que un proceso cambie de un estado a otro. El correspondiente a nuestro modelo se representa en la figura 3.2, en ella podemos ver que las transiciones que pueden darse entre estados son las siguientes:

**Nada → nuevo** Se crea un nuevo proceso para ejecutar un programa. Este suceso puede ocurrir porque un nuevo usuario inicia una sesión en un sistema de tiempo compartido, un proceso existente crea un proceso hijo, etc.

**Nuevo → listo** El sistema operativo pasará un proceso del estado de nuevo al de listo cuando esté preparado para encargarse de él. La cantidad de memoria

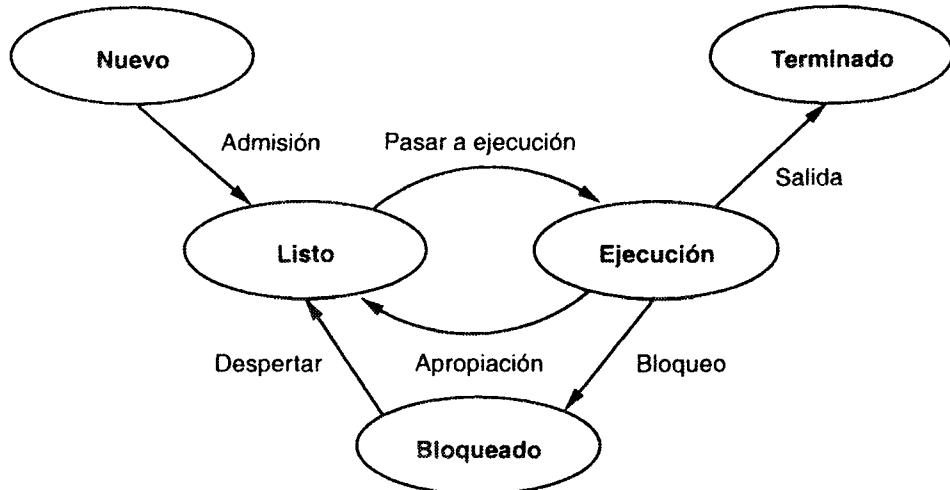


Figura 3.2: Modelo de procesos de cinco estados

principal disponible en un sistema representa un límite para el número de procesos activos que puede haber en él, de ahí, que el sistema no pueda admitir cualquier número de procesos.

**Listo → ejecución** Cuando el procesador queda libre habrá que seleccionar, entre los que se encuentran en estado listo, un nuevo proceso para ejecutarse. La cuestión de qué proceso es el que se elige se verá en el capítulo 4 en el que se tratará la planificación de procesos.

**Ejecución → terminado** El proceso que se está ejecutando actualmente finaliza su ejecución o bien es abortado debido a que se ha producido algún tipo de error.

**Ejecución → listo** El proceso que se está ejecutando actualmente pierde el control de la CPU para cedérsela a otro proceso. La razón más común para que se produzca esta transición es que el proceso en ejecución haya consumido el tiempo máximo de uso ininterrumpido del procesador. Si en un sistema el criterio para que un proceso tome el control de la CPU es su prioridad, el hecho de que llegue al sistema un proceso con mayor prioridad que el que se está ejecutando actualmente puede hacer que éste pierda el control de la CPU.

**Ejecución → bloqueado** Un proceso pasa al estado bloqueado si necesita algo sin lo que no puede continuar su ejecución. Por ejemplo, un proceso puede requerir un servicio del sistema operativo que éste no está preparado para

realizar de inmediato. Puede pedir un recurso que no esté disponible inmediatamente, o bien, puede necesitar que se realice una operación de E/S que tardará un tiempo en llevarse a cabo.

**Bloqueado → listo** Un proceso en estado bloqueado pasa al estado listo cuando ocurre el suceso que estaba esperando.

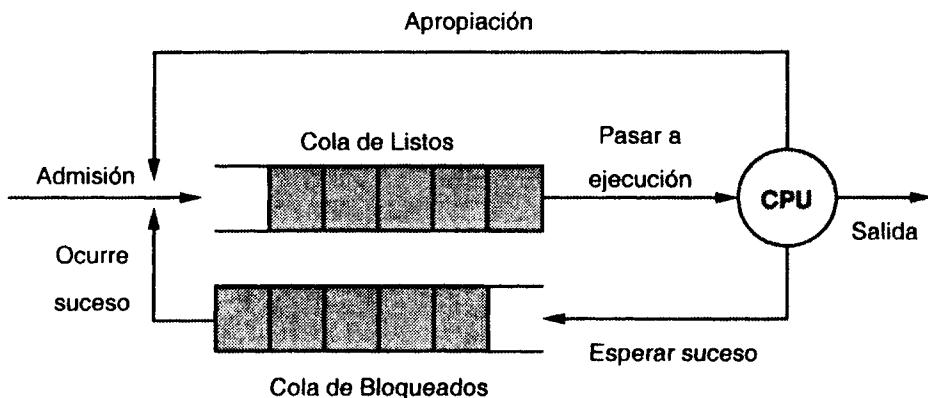
Con este modelo simple ya podemos empezar a apreciar algunos de los elementos de diseño del sistema operativo. Éste debe ser capaz de seguirle la pista a cada proceso del sistema, es decir, debe mantener información sobre cada proceso, como por ejemplo el estado actual y su localización en memoria. Los procesos en estado listo y bloqueado deberían mantenerse en colas, esperando su turno para ejecutarse. En la figura 3.3 aparecen estas dos colas y las razones por las que los procesos se mueven entre ellas.

Como se puede observar tenemos dos colas, una corresponde a los procesos listos y la otra a los procesos bloqueados. Cuando el sistema admite un nuevo proceso, éste se coloca en la cola de procesos listos. Si el sistema tiene que elegir un proceso para ejecutarlo, selecciona uno de la cola de procesos listos. Cuando a un proceso que se está ejecutando se le retira el procesador, puede ser debido a varias causas: ha terminado su ejecución, por lo que saldrá del sistema, ha finalizado el tiempo máximo de uso de la CPU, en este caso pasará a la cola de procesos listos, o bien espera que ocurra algún suceso, en este caso será colocado en la cola de procesos bloqueados. Finalmente, cuando se produzca el suceso que está esperando, pasarán de la cola de procesos bloqueados a la de procesos listos todos aquellos procesos que estuvieran esperando dicho suceso.

Esto último implica que, cuando ocurre un suceso, el sistema operativo debe rastrear la cola de procesos bloqueados, en busca de aquellos que estaban esperándolo. En sistemas donde puede haber cientos de procesos bloqueados, podría ser más eficiente tener una cola por cada suceso. Así cuando ocurre cierto suceso, todos los procesos que están en la cola asociada a dicho suceso pueden pasar a estado listo.

Respecto al orden de los procesos en las colas, depende tanto del tipo de cola como de otros aspectos. Así, en el caso de la cola de listos, estará ordenada en función del criterio de selección de procesos, del que se hablará en el capítulo 4. Por el contrario, la cola de bloqueados no necesita un orden especial entre los procesos, ya que no se sabe en qué orden van a ocurrir los sucesos que están esperando.

La implementación de las colas de procesos listos y bloqueados se puede realizar mediante una lista enlazada de bloques de datos que mantiene el sistema



**Figura 3.3:** Modelo de colas para el diagrama de la figura 3.2

operativo sobre cada proceso, esto se verá con más detenimiento en el apartado 3.4.

### 3.2.3.2 Estados suspendidos

El modelo de cinco estados visto anteriormente puede servir para construir un sistema operativo basado en él. Pero podemos considerar un modelo más complejo al que se añaden nuevos estados: los estados suspendidos (suspendido-listo y suspendido-bloqueado).

Un proceso suspendido es aquel que no está disponible de inmediato para su ejecución. Las razones por las que puede llegarse a esta situación pueden ser varias:

- Se pueden suspender procesos si se considera que la carga del sistema es alta. En este caso, se puede optar por sacar procesos de memoria principal y llevarlos a memoria secundaria (intercambio). Cuando la carga del sistema disminuya estos procesos podrán reanudarse, volviendo a memoria principal.
- Si un sistema está funcionando mal y es posible que falle, se puede querer suspender los procesos activos para reanudarlos cuando se haya corregido el problema.
- El sistema operativo puede suspender un proceso si sospecha que es el causante de un problema.

- Un proceso padre puede suspender un proceso hijo para coordinar su actividad con la de otros.
- Un usuario puede suspender un proceso si sospecha que los resultados parciales que está produciendo no son correctos. Cuando verifique si el proceso está funcionando correctamente o no, podrá proceder a reanudarlo o abortarlo.
- Consideraciones de temporización también pueden conducir a suspender un proceso. Por ejemplo, si un proceso se activa periódicamente pero está ocioso la mayor parte del tiempo, podría suspenderse mientras no esté activo. Un ejemplo de procesos de este tipo podría ser los que monitorizan la actividad de los usuarios.

En todos los casos, la activación de un proceso suspendido la pide el mismo agente que inicialmente solicitó la suspensión. Esta es una diferencia clara entre los procesos suspendidos y bloqueados, en este último caso el proceso pasa al estado listo cuando ocurre el suceso que estaba esperando, no existe una activación.

Debido a esta diferencia entre suspensión y bloqueo, los procesos suspendidos pueden encontrarse en dos estados diferentes:

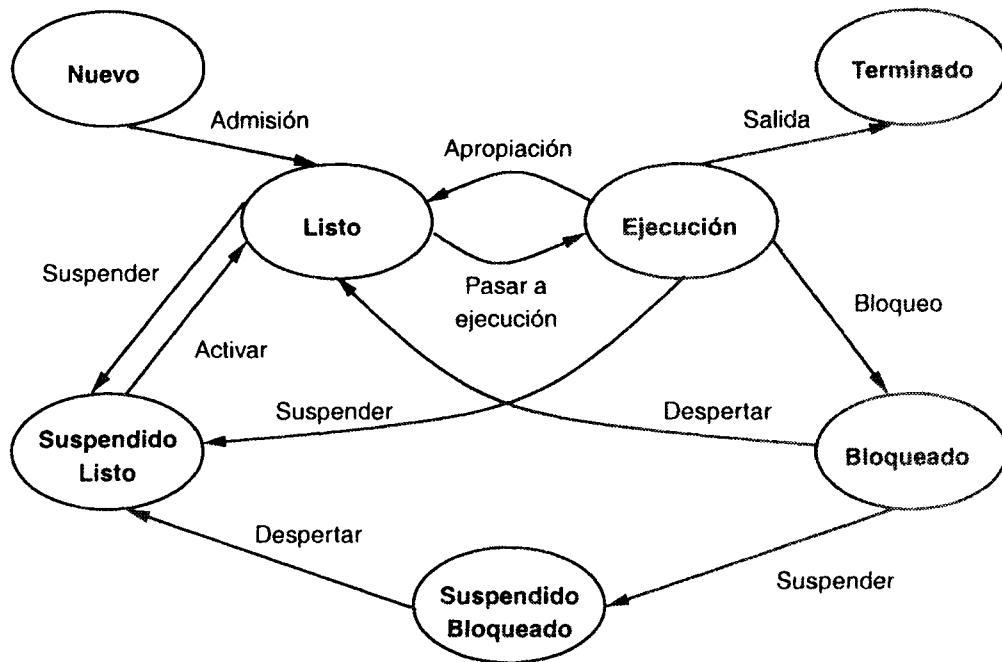
**Suspendido-bloqueado** Un proceso que está esperando a que ocurra un suceso y que además ha sido suspendido por alguna de las razones citadas. Ambas condiciones son independientes; si el suceso esperado ocurre, esto no habilita al proceso para su ejecución.

**Suspendido-listo** Un proceso que estaba en estado listo y ha sido suspendido, o bien, estaba en estado suspendido-bloqueado y ha ocurrido el evento que esperaba.

Al introducir los dos nuevos estados aparecen nuevas transiciones importantes (figura 3.4):

**Bloqueado → suspendido-bloqueado** Un proceso bloqueado es suspendido por algún agente: el sistema operativo, el proceso padre, etc.

**Suspendido-bloqueado → suspendido-listo** Esta transición se produce cuando ocurre el suceso que estaba esperando el proceso. Hay que hacer notar que esto requiere que la información de estado referente a los procesos suspendidos esté disponible para el sistema operativo.



**Figura 3.4:** Diagrama de transición de estados con estados suspendidos

**Suspendido-listo → listo** El agente que suspendió al proceso lo devuelve al estado listo.

**Listo → suspendido-listo** Un proceso que está en estado listo es suspendido por alguna de las razones citadas anteriormente.

**Ejecución → suspendido-listo** Se suspende el proceso que se está ejecutando actualmente.

### 3.3 Imagen de un proceso

Un proceso consta como mínimo de:

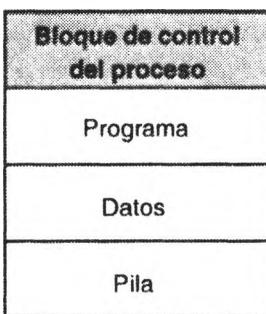
**Programa de usuario** El programa o conjunto de programas que van a ser ejecutados.

**Datos del usuario** Conjunto de localizaciones de memoria para almacenar las variables locales y globales, y las constantes definidas.

**Pilas** Cada proceso dispone de una pila del usuario que se utiliza para almacenar argumentos y otros datos relativos a funciones que se ejecutan en modo usuario. También tiene una pila del núcleo que se emplea en las llamadas al sistema y siempre que éste pasa de modo usuario a modo núcleo.

**Bloque de control del proceso**<sup>2</sup> Es el conjunto de atributos asociados al proceso que utiliza el sistema operativo para su control.

El conjunto formado por el programa, los datos, la pila y los atributos se suele denominar **imagen del proceso** (figura 3.5).



**Figura 3.5:** Imagen de un proceso

La forma en que se almacena en memoria la imagen de un proceso depende del esquema de gestión de la memoria que se esté usando. En el caso más simple, la imagen del proceso se mantiene como un bloque contiguo de memoria. Este bloque estará en memoria secundaria o en memoria principal, dependiendo del estado del proceso. En los sistemas operativos modernos la imagen del proceso consta de un conjunto de bloques que no necesitan estar cargados de forma contigua, permitiéndose que el proceso se ejecute sin que todos estos bloques estén cargados en memoria principal simultáneamente (sistemas de memoria virtual), es decir, sólo una parte de la imagen reside en memoria principal mientras el resto se encuentra en memoria secundaria. En cualquier caso, para que el sistema operativo pueda encargarse de controlar el proceso, al menos una pequeña porción de la imagen debe residir siempre en memoria principal, ésta es el bloque de control del proceso. El sistema debe mantener la información de dónde se encuentran en cada momento las distintas porciones de la imagen del proceso.

<sup>2</sup>Suele aparecer en la bibliografía de forma abreviada como BCP, o en su nomenclatura inglesa PCB (*Process Control Block*). También lo podemos encontrar con el nombre de descriptor del proceso.

### 3.3.1 Bloque de control del proceso

En un sistema de tiempo compartido, se requiere una gran cantidad de información sobre cada proceso para su manejo. Podemos considerar que ésta reside en el bloque de control del proceso. Dado que esta información se puede organizar de distintas formas en diferentes sistemas, sólo nos ocuparemos del tipo de información que utiliza el sistema operativo y no de cómo se organiza ésta.

La información contenida en el bloque de control del proceso se puede agrupar en tres tipos:

- Identificación del proceso.
- Información del estado del procesador.
- Información de control del proceso.

Con respecto a la identificación del proceso, en casi todos los sistemas operativos se asigna a cada proceso un identificador numérico único. Éste se suele utilizar como índice de la tabla de procesos y también puede ser utilizado en las tablas de memoria, E/S y ficheros para indicar en qué zona de memoria reside el proceso, o bien qué dispositivos de E/S o ficheros tiene asignados. Cuando los procesos se comunican entre sí, su identificador sirve para informar al sistema operativo del destino del mensaje. Cuando a los procesos se les permite generar otros, los identificadores se usan para indicar quién es el padre y quiénes los descendientes.

Además de su propio identificador, a un proceso se le puede asignar un identificador de usuario que indica quién es el usuario responsable de él.

Otro conjunto importante de información es la referente al estado del procesador, que está constituida fundamentalmente por el contenido de los registros de éste. Mientras el proceso está en ejecución, los registros mantienen información; cuando el proceso deja de ejecutarse, toda la información contenida en ellos debe guardarse para que pueda ser recuperada cuando el proceso reanude su ejecución. Esta información incluye normalmente, los registros visibles al usuario, los registros de control y estado, y los punteros a las pilas.

El tercer conjunto de información dentro del bloque de control del proceso, está compuesto por la información adicional que necesita el sistema operativo para controlar y coordinar los distintos procesos activos; ésta incluye el estado del proceso, su prioridad, los recursos que tiene asignados, punteros que permiten crear listas enlazadas de los BCP, etc. La podemos denominar información de control

**Identificación del proceso**

- Identificador del propio proceso<sup>3</sup>.
- Identificador del proceso que lo creó (proceso padre)<sup>4</sup>.
- Identificador del usuario responsable del proceso<sup>5</sup>.

**Información de estado del procesador**

- Registros visibles al usuario.
- Registros de control y estado: contador de programa (PC), palabra de estado del proceso (PSW), ...
- Punteros a las pilas.

**Información de control del proceso**

- Información sobre la planificación del proceso: la información concreta a guardar dependerá del algoritmo de planificación utilizado.
- Comunicación entre procesos.
- Privilegios del proceso.
- Manejo de memoria: puntero al espacio de direcciones que ocupa el proceso.
- Propiedad de los recursos y utilización.
- Punteros a otros procesos.

---

**Tabla 3.1:** Elementos típicos de un bloque de control de un proceso

<sup>3</sup>En la bibliografía en inglés suele aparecer como PID (*Process IDentifier*).

<sup>4</sup>En la bibliografía en inglés suele aparecer como PPID (*Parent Process IDentifier*).

<sup>5</sup>También suele aparecer como UID (*User IDentifier*)

del proceso. A medida que avancemos en el estudio de los sistemas operativos, se verá más clara la necesidad de estos elementos.

La tabla 3.1 lista los distintos tipos de información que requiere el sistema operativo para cada proceso.

Resumiendo, podemos decir que el bloque de control del proceso es la estructura de datos central y más importante en un sistema operativo. Cada uno de ellos contiene toda la información que éste necesita sobre un proceso. Podríamos decir que el conjunto de bloques de control de los procesos define el **estado del sistema operativo**.

## 3.4 Control de los procesos

Como ya hemos comentado anteriormente, el sistema operativo se encarga de asignar el procesador a los procesos para su ejecución, les asigna recursos y responde a las peticiones de servicios que hacen éstos. Para realizar estas funciones el sistema operativo necesita mantener información acerca de los procesos que existen en el sistema. Para ello construye y mantiene tablas de información sobre cada una de las entidades que maneja. En general, suele mantener cuatro tipos de tablas con información sobre la memoria, E/S, ficheros y procesos. Aunque los detalles pueden variar de un sistema operativo a otro, prácticamente todos ellos mantienen estos cuatro tipos de información.

El sistema operativo debe mantener **tablas de procesos** para administrarlos. En la figura 3.6 se representa una posible implementación de ésta, con una entrada por cada proceso donde se almacena su bloque de control. El identificador del proceso constituye el índice en dicha tabla. El contenido de cada entrada puede variar de un sistema a otro. En algunos, en lugar de almacenar el bloque de control, cada entrada sólo contiene un puntero a la zona de memoria donde está ubicado.

El bloque de control del proceso puede contener punteros que permiten enlazarlos entre sí. Así, las colas que se describieron en el apartado 3.2 pueden ser implementadas como listas enlazadas de bloques de control de procesos, como se muestra en la figura 3.6, donde se refleja la cola de procesos listos, y la de bloqueados.

En la figura 3.6 también aparecen los registros del procesador. Así, el registro de índice del proceso contiene el identificador del proceso que está actualmente ejecutándose, es decir, su PID. El contador de programa apunta a la próxima instrucción a ejecutar. Los registros base y límite definen la región de memoria

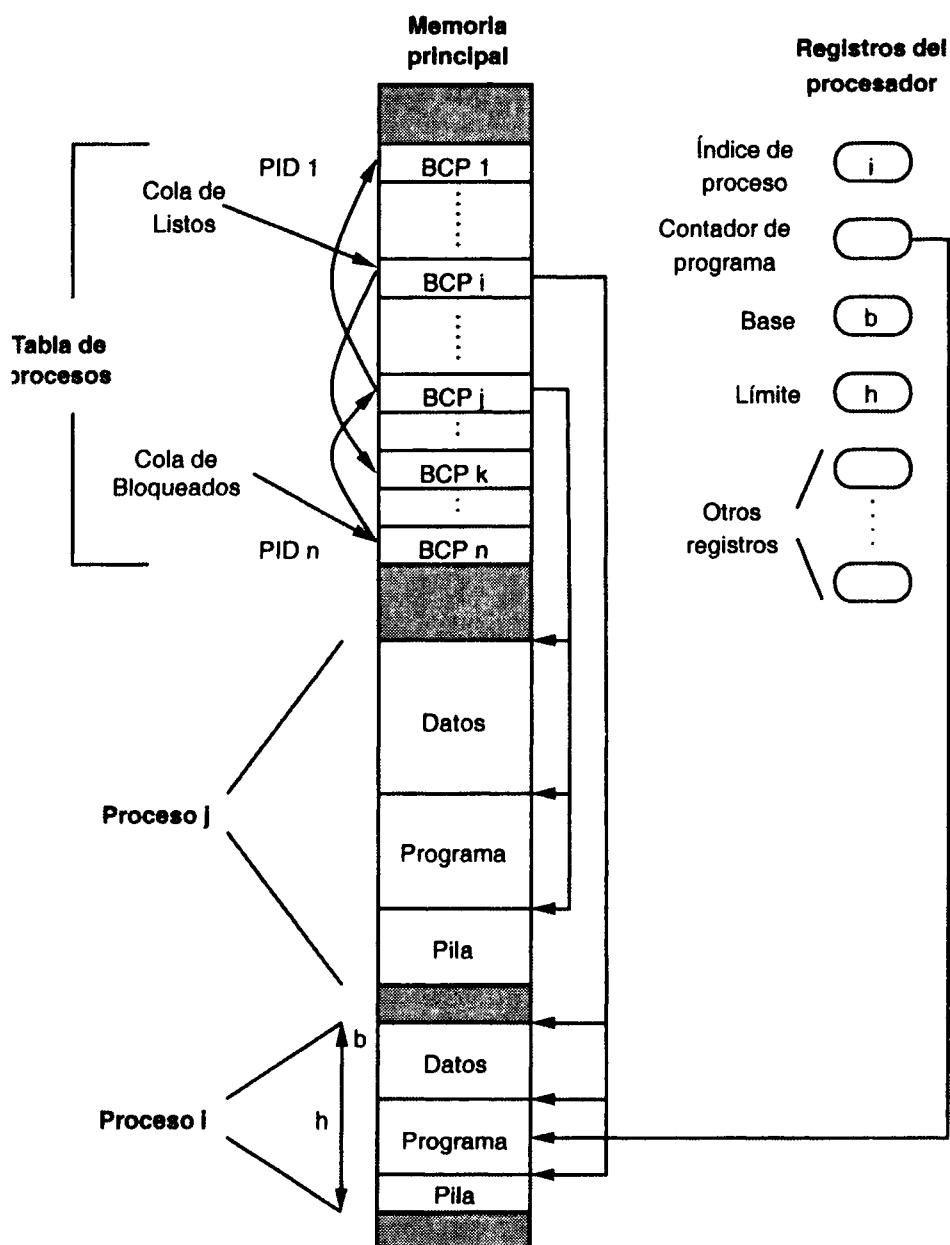


Figura 3.6: Implementación de la tabla de procesos

que ocupa el proceso. Tanto el contador de programa como las referencias a datos que se realizan se interpretan como relaciones relativas al registro base y no deben exceder el valor del registro límite. Esto permite que no existan interferencias entre los procesos. Todos estos registros se almacenan en el BCP cuando el proceso abandona el estado de ejecución.

Las **tablas de memoria** le permiten conocer el estado de la memoria principal y de la secundaria. Suelen incluir la siguiente información:

- Memoria principal asignada a los procesos.
- Memoria secundaria asignada a los procesos.
- Atributos de protección de zonas de memoria principal y secundaria, por ejemplo, puede indicar qué procesos tienen acceso a una zona de memoria compartida.
- Cualquier otra información que se necesite para administrar la memoria.

Estas estructuras de información para el manejo de la memoria se verán con más detalle en los capítulos 7 y 8.

El sistema operativo utiliza las **tablas de E/S** para llevar el control de los dispositivos. En un instante dado, un dispositivo de E/S puede estar disponible o asignado a un proceso particular. Si se está realizando una operación de E/S, el sistema operativo necesita saber el estado de ésta y la localización en memoria principal que se está usando como fuente o destino de la transferencia de E/S. La gestión de la E/S se estudiará en el capítulo 9.

Por último, las **tablas de ficheros** proporcionan información sobre qué ficheros existen, su localización en memoria secundaria, estado actual, y otros atributos.

Todas las tablas de las que hemos hablado (memoria, ficheros, E/S y procesos) deben estar, de alguna forma, relacionadas entre sí, ya que la memoria, los ficheros y los dispositivos de E/S se administran en nombre de los procesos. Las tablas de procesos deben contener referencias directas o indirectas a los recursos que están utilizando éstos.

Anteriormente se ha dicho que el sistema operativo crea y mantiene estas tablas. Nos podríamos preguntar ¿cómo sabe el tamaño deben tener? El sistema operativo debe tener algún conocimiento del ambiente básico, por ejemplo, cuánta memoria principal existe, qué dispositivos de E/S hay, de qué tipo son éstos, etc. En algunos sistemas, algunos de estos datos pueden ser especificados durante

el proceso de configuración por el administrador, mientras que en otros están establecidos en el código fuente.

### 3.5 Gestión de procesos

Una de las funciones del núcleo de un sistema operativo es la gestión de los procesos. Las tareas que se engloban dentro de esta función son las siguientes:

- Creación y terminación de procesos.
- Cambio de procesos.
- Gestión de los bloques de control de los procesos.
- Planificación y despacho de procesos.
- Sincronización y soporte para la comunicación entre los procesos.

Una vez estudiados los estados de un proceso y las estructuras que mantiene el sistema para su control vamos a describir de forma más detallada los pasos que da el sistema para la creación de procesos. A continuación se tratará las operaciones implicadas en el cambio de proceso. La gestión de los bloques de control de los procesos hace referencia a todas las operaciones implicadas desde que se crean hasta que se destruyen, tales como el cambio de cola, modificación de la información referente al estado del procesador o al control del proceso. La función de planificación y despacho será tratada en el capítulo 4, mientras que la sincronización y comunicación entre procesos se estudiará en el capítulo 5.

#### 3.5.1 Creación de procesos

En el apartado 3.2.1 hablamos de los sucesos que podían conducir a la creación de un nuevo proceso, a continuación vamos a describir más detalladamente los pasos que da el sistema para crearlos:

1. Asignar un identificador único al nuevo proceso. Se añade una nueva entrada a la tabla de procesos, que contiene una entrada por proceso.
2. Asignar un espacio de direcciones al proceso. Se debe asignar espacio para todos los elementos de la imagen del proceso. Por tanto, el sistema operativo debe conocer cuánto ocupará el espacio de direcciones privado del usuario.

(programa y datos) y la pila del usuario. Estos valores se pueden asignar por omisión, teniendo en cuenta el tipo de proceso, o se pueden establecer teniendo en cuenta la petición del usuario cuando se crea el trabajo. Si un proceso es generado por otro, el proceso padre puede pasar al sistema operativo los valores necesarios como parte de la petición de creación del proceso. Si el nuevo proceso va a compartir algún espacio de direcciones ya existente, se deben establecer los enlaces apropiados. Por último, se debe asignar espacio para el bloque de control del proceso.

3. Inicializar el bloque de control del proceso. La porción de identificación del proceso contiene el identificador de éste más otros identificadores necesarios, tal como el del proceso padre. La porción de información del estado del procesador normalmente se inicializará con la mayor parte de sus entradas a cero, excepto el contador del programa (se establece al punto de entrada del programa) y los punteros a la pila del sistema. La porción de información de control del proceso se inicializa con los valores por omisión o con los valores pedidos por el proceso. Por ejemplo, el estado del proceso será inicializado como listo. La prioridad se puede establecer inicialmente al valor más bajo, a menos que se haga una petición explícita de un valor más alto. Inicialmente, el proceso no poseerá ningún recurso (dispositivos de E/S y ficheros), a no ser que se pidan explícitamente o sean heredados del padre.
4. Se debe introducir en la cola de procesos correspondiente a su estado inicial, estableciéndose los enlaces adecuados.
5. Si es necesario se crearán o expandirán otras estructuras de datos. Por ejemplo, si el sistema operativo mantiene un fichero de contabilidad para cada proceso con fines de facturación o de estudio del rendimiento del sistema, se deberá crear el fichero correspondiente.

### 3.5.2 Cambio de proceso

De una forma simple podemos decir que el **cambio de proceso** consiste en que el proceso que se está ejecutando actualmente pierde el control del procesador, lo toma el sistema operativo y éste le da el control a otro proceso. A la hora de estudiar el cambio de proceso hemos de tener en cuenta varias dos cuestiones:

1. ¿Qué sucesos pueden provocar un cambio de proceso?
2. ¿Qué debe hacer el sistema operativo con las distintas estructuras de datos que controla, para llevar a cabo un cambio de proceso?

Nos ocuparemos en primer lugar de los sucesos que pueden dar lugar a que un proceso pierda el control de la CPU y lo tome el sistema operativo. La tabla 3.2 muestra un resumen.

Suceso	Mecanismo
Externo a la ejecución de la instrucción actual.	Interrupción.
Error asociado a la ejecución de la instrucción actual.	Excepción.
Petición de un servicio.	Llamada al sistema o paso de mensajes.

**Tabla 3.2:** Sucesos que interrumpen la ejecución de un proceso

Las **interrupciones** son uno de los sucesos que pueden provocar un cambio de proceso. Se pueden distinguir dos clases de interrupciones, las causadas por algún suceso externo e independiente del proceso que se está ejecutando, como por ejemplo una interrupción del reloj; y las que son debidas a que el proceso que se está ejecutando produce un error o condición de excepción. Las primeras se suelen conocer como **interrupciones** y las segundas como **excepciones**.

Cuando se produce una interrupción ordinaria, se transfiere el control en primer lugar a un manejador de interrupciones del sistema, que hace algunas tareas internas y luego le da el control a la rutina del sistema operativo relacionada con ese tipo particular de interrupción.

Cuando se produce una excepción, el sistema operativo determina si el error es fatal. Si es así, el proceso que se está ejecutando actualmente pasa al estado de terminado y continuaría con la ejecución de otro proceso. Si no, la acción del sistema operativo dependerá de la naturaleza del error y el diseño de aquel.

Por último, el sistema operativo puede ser activado por la petición de un servicio realizada por el programa que se está ejecutando. Por ejemplo, un proceso de usuario ejecuta una instrucción que pide una operación de E/S, como la apertura de un fichero. Esta petición produce una transferencia a una rutina que se encarga de proporcionar el servicio.

Cuando se produce una interrupción, una excepción o la petición de un servicio, se llevan a cabo las siguientes acciones:

1. El procesador se prepara para transferir el control a la rutina correspondiente. Para empezar, debe guardarse toda aquella información que pueda ser alterada por la ejecución de la rutina y que necesitaremos recuperar cuando se reanude la ejecución del proceso que hemos interrumpido, es decir, el

**contexto del proceso.** Éste estará constituido por los registros del procesador y será guardado en la pila del núcleo. Esto es lo que se conoce como **cambio de contexto**.

2. A continuación establecerá el valor del contador del programa a la dirección de comienzo de la rutina adecuada, para así proceder a la lectura de la primera instrucción de ésta.

En la figura 3.7 un proceso de usuario es interrumpido después de la instrucción  $N$ . El contenido de todos los registros se colocan en la pila, se actualiza el puntero a la pila y el contador de programa apunta a la dirección de la rutina.

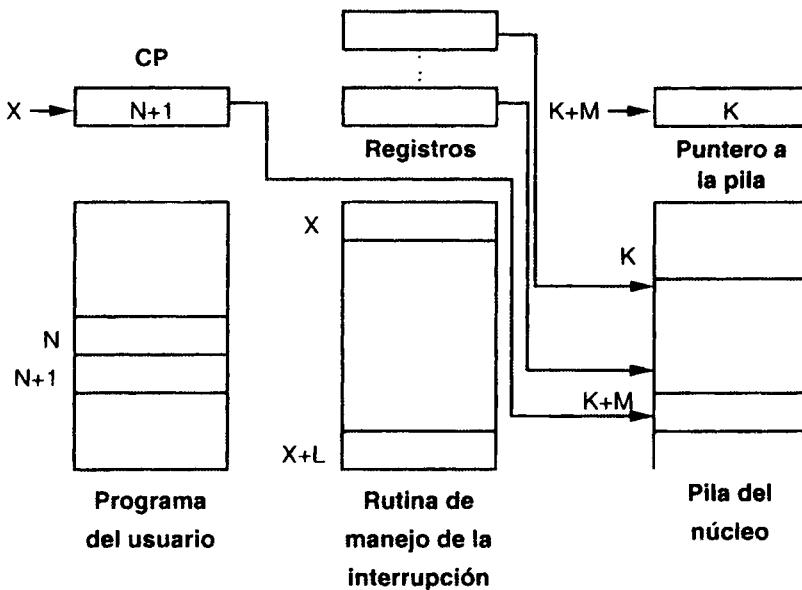


Figura 3.7: Cambios que se producen al atender una interrupción

¿Qué ocurre cuando termina la ejecución de la rutina? El sistema operativo debe decidir entre continuar la ejecución del proceso que se estaba ejecutando o sustituirlo por otro. Esto va a depender de la causa que motivó la interrupción de la ejecución del proceso.

Si se continúa con la ejecución del proceso interrumpido, se recuperarán los valores de los registros del procesador guardados en la pila, entre ellos el contador de programa. Esto va a permitir continuar con la ejecución del proceso en la instrucción siguiente donde fue interrumpido.

Si se sustituye el proceso que se estaba ejecutando por otro, se produce lo que se conoce como cambio de proceso. En este caso, habrán de realizarse las siguientes acciones:

1. Guardar el contexto del proceso, que se encuentra en la pila del núcleo, en su bloque de control.
2. Actualizar el estado del proceso interrumpido en su bloque de control. También deben ser actualizados otros campos importantes, incluyendo la razón por la que deja el estado de ejecución y la información de contabilidad.
3. Mover el bloque de control del proceso a la cola apropiada.
4. Seleccionar otro proceso para su ejecución. Esto se estudiará con más profundidad en el capítulo 4.
5. Actualizar el bloque de control del proceso seleccionado. Esto incluye cambiar el estado del proceso a ejecución.
6. Actualizar las estructuras de datos de manejo de la memoria.
7. Restaurar el contexto del proceso seleccionado tal como estaba en el momento en que salió del estado de ejecución por última vez, cargando los valores previos del contador de programa y otros registros.

Vemos pues que el cambio de proceso requiere considerablemente más esfuerzo que el cambio de contexto. Cuando se produce un cambio de proceso, lo primero que se realiza es el cambio de contexto, y lo último, justo antes de otorgar el control del procesador al otro proceso, es restaurar su contexto.

Podemos decir, por tanto, que no es lo mismo un cambio de proceso que un cambio de contexto. Estos dos conceptos son tratados a menudo en la bibliografía como uno solo, denominándose cambio de contexto al cambio de proceso y no dándose ninguna denominación especial al primero.

Otra operación relacionada con las anteriores es el cambio de estado. Siempre que se produce un cambio de proceso se llevan a cabo dos cambios de estado, el proceso que estaba ejecutándose pasa a un estado distinto, y uno de los procesos en estado listo pasa a ejecución.

### 3.6 Hilos de ejecución

Hasta ahora se ha presentado el concepto de proceso como una entidad que reúne las dos características siguientes:

**Unidad propietaria de recursos** Un proceso tiene asignado un espacio de direcciones para mantener su imagen, así como el control de otros recursos, tales como dispositivos de E/S y ficheros.

**Unidad de despacho** Un proceso es un camino o traza de ejecución a través de uno o más programas. Es la entidad que toma el control del procesador cuando el sistema operativo se lo cede. Un proceso tiene un estado y una prioridad de despacho.

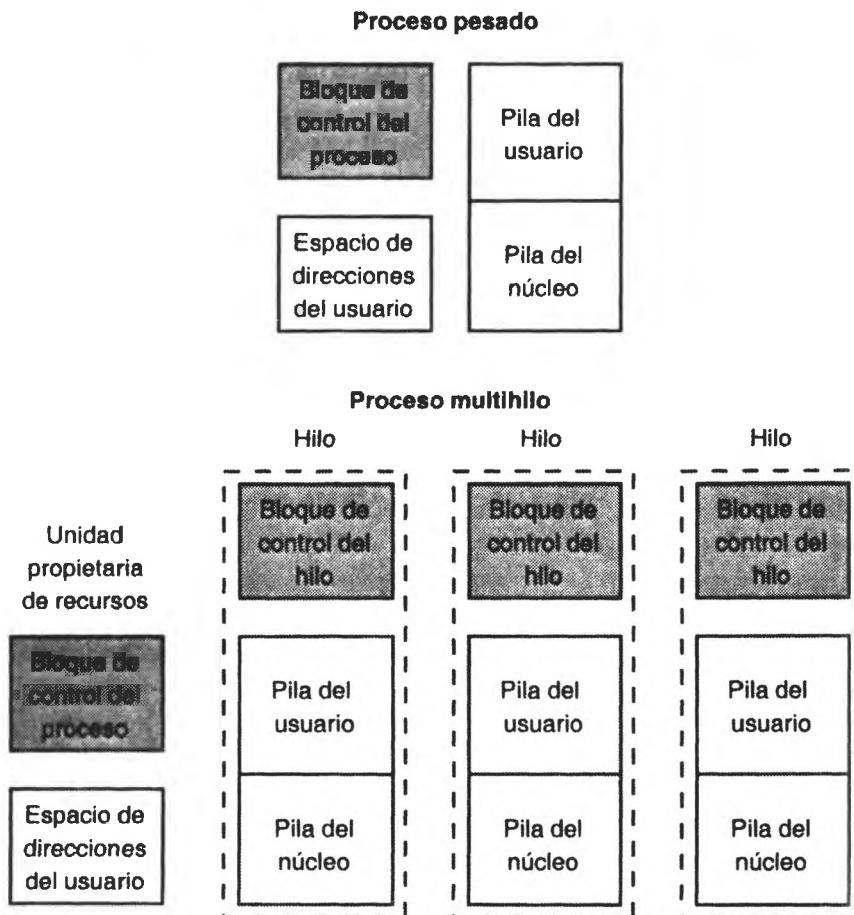
En los sistemas operativos tradicionales estas dos características van unidas, hablándose de un **proceso pesado**. En sistemas más recientes se han separado, pasando a denominarse **hilo**<sup>6</sup> a la unidad de despacho, mientras que la unidad propietaria de los recursos se sigue denominando proceso. Esta separación permite que un proceso pueda tener asociados múltiples hilos, al contrario que en el enfoque tradicional donde podemos decir que el proceso pesado consta de único hilo. Sistemas operativos tales como OS/2, la versión de UNIX que hace Sun, Windows NT, el sistema Mach, o LINUX utilizan el modelo de múltiples hilos por proceso, denominándose sistemas multihilo.

En un ambiente multihilo, el proceso lleva asociado el espacio de direcciones de la imagen y los recursos asignados, tales como ficheros, dispositivos de E/S. Dentro de un proceso puede haber uno o más hilos, cada uno con su estado, su contexto, su pila, almacenamiento estático para variables locales y derechos de acceso a la memoria y a los recursos del proceso.

En la figura 3.8 se muestra la diferencia entre un proceso pesado y un proceso con múltiples hilos. Un proceso pesado se representa mediante su bloque de control y lleva asociados un espacio de direcciones y las pilas de usuario y núcleo. Sin embargo, en un modelo multihilo, el proceso tiene su bloque de control y su espacio de direcciones, mientras que cada uno de los hilos tiene también su propio bloque de control y las pilas de usuario y núcleo. La existencia de un bloque de control por hilo es necesaria debido a que cada hilo tiene su propio estado, prioridad y contexto.

Los posibles estados en que puede encontrarse un hilo son ejecución, bloqueado y listo. Normalmente, no tiene sentido asociar estados suspendidos con los hilos, sino que éstos van asociados a los procesos. Por ejemplo, si se intercambia un proceso, todos sus hilos tendrán que estar necesariamente intercambiados puesto que comparten el espacio de direcciones del proceso. De forma similar, la terminación de un proceso implica la terminación de todos los hilos que lo componen.

<sup>6</sup>Traducción de la palabra inglesa *thread*. También se emplea el término **proceso ligero** para denominarlo.



**Figura 3.8:** Proceso pesado frente a proceso multihilo

Las principales ventajas que se obtienen con la utilización de hilos están relacionadas con mejoras en el rendimiento. Por ejemplo, si una aplicación se puede implementar como un conjunto de unidades de ejecución relacionadas, va a ser más eficiente implementarla como un conjunto de hilos dentro de un mismo proceso, que como una colección de procesos separados. Esto es debido a que se necesita menos tiempo para crear un hilo dentro de un proceso existente, que para crear un proceso nuevo. También se va a necesitar menos tiempo para terminar un hilo y para cambiar entre dos hilos del mismo proceso. Un ejemplo de aplicación que podría hacer uso de esta ventaja es un servidor de ficheros. En este caso, se crearía un nuevo hilo cada vez que llegara una petición. Puesto que un servidor manejará muchas peticiones, se crearán y destruirán muchos hilos en un corto período de

tiempo.

Otro aspecto en el que los hilos proporcionan eficiencia es en la comunicación entre diferentes programas que se ejecutan. Dado que los hilos de un proceso comparten el mismo espacio de direcciones y los recursos (tales como ficheros), la comunicación entre ellos no requiere de la intervención del núcleo. Sin embargo, en la mayor parte de los sistemas operativos, es el núcleo el que proporciona los mecanismos necesarios para la comunicación entre procesos independientes.

## 3.7 Procesos en LINUX

El concepto de proceso en LINUX es el mismo que para cualquier otro sistema operativo. Al ser un sistema multiprogramado, puede haber muchos procesos residentes en el sistema de forma simultánea.

### 3.7.1 Imagen de un proceso

En LINUX, la imagen de un proceso está compuesta por los siguientes elementos:

**El segmento de datos** Contiene los datos y es exclusivo de cada proceso. Se divide en datos inicializados, es decir, datos cuya situación no cambia a lo largo de la ejecución del proceso y datos no inicializados. El tamaño de este segmento puede variar en tiempo de ejecución.

**El segmento de texto o código** Es una copia del bloque de texto del programa. Puede ser compartido por varios procesos y es de solo lectura. El texto, al igual que los datos inicializados, se toman del fichero ejecutable.

**El segmento de pila** Lo crea el núcleo al arrancar el proceso, gestionando dinámicamente su tamaño. La pila se compone de una serie de bloques lógicos, llamados **marcos de pila**, que se introducen cuando se llama a una función y se sacan cuando se vuelve de la función. Un marco de pila se compone de los parámetros de la función, las variables locales a ésta y la información necesaria para restaurar el marco de pila anterior a la llamada a la función (contador de programa y puntero de pila). El segmento de pila es exclusivo para cada proceso. Dado que los procesos se pueden ejecutar en dos modos: modo usuario y modo núcleo, el sistema maneja dos pilas por separado, la pila del usuario y la pila del núcleo.

**Bloque de control del proceso** Esta estructura de datos es la que contiene toda la información sobre el proceso, y permanece en memoria, incluso cuando

éste no está residente en ella.

### 3.7.2 El bloque de control de procesos

En LINUX es una estructura compleja que contiene información sobre la identidad del proceso, estado, tiempos de ejecución, zona de memoria para el intercambio, punteros a la tabla de memoria, los ficheros y sistemas de ficheros asociados, hilos, procesador donde se ejecuta, etc. La información que contiene es la siguiente:

**Identificadores** Son los siguientes:

- Identificación del proceso (PID y GID) y del padre (PPID), que especifican las relaciones entre procesos.
- Identificación del usuario (UID y GID, reales y efectivos) para determinar los privilegios del proceso.

**Información del procesador** En el caso de ejecutarse en un entorno multiprocesador.

**El contador de programa** Contiene la dirección de la siguiente instrucción que debe ejecutar la CPU.

**Registros de propósito general** Contiene información útil para el funcionamiento del programa como mensajes de error, señales, etc.

**Otros registros** Almacenan el valor de los registros hardware en los cambios de contexto del proceso.

**Puntero a la pila** Contiene la dirección de la siguiente entrada en la pila del usuario o del núcleo, dependiendo del modo de ejecución en que estuviese en el momento de la apropiación.

**Punteros a la imagen** Sirve para localizar el proceso, tanto en memoria principal como en memoria secundaria.

**Punteros a otras estructuras del sistema** Contiene la información útil sobre los ficheros y sistemas de ficheros.

**Parámetros para la planificación del proceso** Tales como prioridad, tiempo consumido de CPU, tiempo de E/S, estado de espera, etc., y demás parámetros que se utilizan para determinar cuál es el siguiente proceso a ejecutarse.

**Señales** Enumera las señales que ha recibido el proceso y no ha tratado todavía.  
Las señales pueden estar activas o ser ignoradas.

**Miscelánea** Entre ellos se encuentran:

- Tamaño del proceso.
- Punteros a los bloques de control anterior y posterior en la tabla de procesos.
- Descriptores de señales a la espera.
- Punteros al proceso padre y al proceso hijo más joven.
- Indicador para saber si es intercambiable.
- Parámetros de entrada y de salida.

### 3.7.3 La tabla de procesos

El núcleo del sistema ve a un proceso como una entrada en la tabla de procesos que apunta a su bloque de control. El tamaño de ésta es fijo, por tanto, el número máximo de procesos que puede haber en el sistema está limitado. También se limita el número de procesos que puede crear un usuario; este límite se establece en la mitad del número de procesos totales. Por motivos de seguridad, mantiene como mínimo, cuatro entradas libres para el administrador del sistema. De este modo, si el sistema está colapsado en cuanto al número de procesos, el superusuario podrá crear cuatro procesos para poder realizar sus labores de administración.

### 3.7.4 Estados de un proceso

LINUX reconoce seis estados, empleando dos estados de ejecución para indicar si el proceso se está ejecutando en modo usuario o modo supervisor. La figura 3.9 muestra el diagrama de estados de LINUX, donde aparecen los siguientes estados:

**Nuevo** El proceso acaba de ser creado y está en un estado de transición; el proceso existe, pero no está preparado para ejecutarse. Este estado es el inicial para todos los procesos, excepto para el primero que se crea en el sistema, llamado proceso init.

**Listo** El proceso está esperando su turno para poder ejecutarse en la CPU. El planificador antes de escoger un proceso comprueba si alguno de los bloqueados puede pasar a este estado, y después escoge uno entre los procesos listos.

### Ejecutándose en modo usuario

### Ejecutándose en modo supervisor

**Bloqueado** El proceso se encuentra esperando una señal o evento para poder continuar con su ejecución. Diferencia dos tipos de bloqueos, uno causado por el funcionamiento del proceso y otro por algo ajeno a él, como por ejemplo cuando el usuario ordena pararlo. En este último estado, el proceso puede ser reiniciado externamente, o bien, terminar completamente cuando el proceso padre lo haga.

**Terminado** Es un estado intermedio en la terminación del proceso. Durante el tiempo que el proceso está en este estado envía al proceso padre un registro que contiene el código de salida y algunos datos estadísticos tales como los tiempos de ejecución.

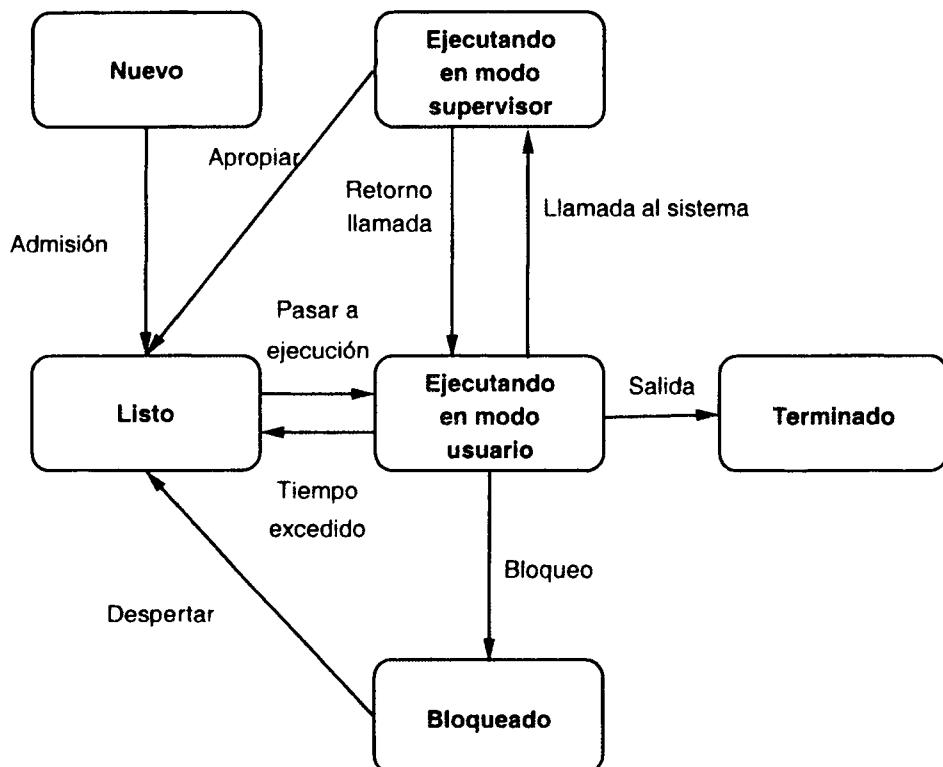


Figura 3.9: Diagrama de estados de un proceso en LINUX

### 3.7.5 Llamadas al sistema

LINUX dispone de una serie de llamadas al sistema relacionadas con la gestión de procesos, éstas aparecen en la tabla 3.3.

Nombre	Descripción
clone	Crea un contexto de ejecución.
fork	Crea un proceso.
exec	Ejecuta un programa.
wait	Espera la terminación de un proceso.
exit	Termina un proceso.

Tabla 3.3: Llamadas al sistema relacionadas con procesos

La llamada `fork` sirve para crear un nuevo proceso. Ésta crea una copia exacta del proceso que realiza la llamada, éste es el proceso padre y el nuevo proceso que se crea es el hijo. Cuando se realiza una llamada `fork` el sistema realiza las siguientes operaciones:

1. El núcleo busca una entrada libre en la tabla de procesos y la reserva.
2. Si la encuentra, copia toda la información del proceso padre en la entrada del proceso hijo.
3. Asigna un identificador al proceso hijo.
4. Comparte inicialmente todos los segmentos del proceso padre bajo la técnica de **copia-en-escritura**<sup>7</sup>.

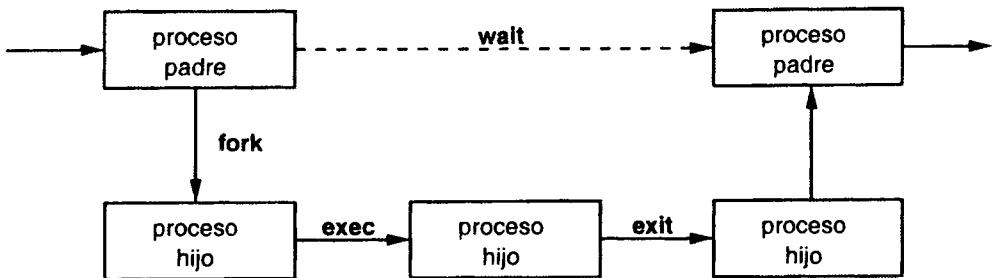
La llamada `exec` ejecuta un programa, es decir, sustituye el código y los datos oprimidos del proceso padre por la nueva imagen contenida en el fichero que se le pasa como argumento. Es decir, carga un programa en la zona de memoria del proceso que lo ejecuta, sobrescribiendo los segmentos del programa antiguo con los del nuevo.

Cuando un proceso termina voluntariamente hace la llamada al sistema `exit`, esta tiene las siguientes consecuencias:

1. Se liberan los recursos asignados al proceso.
2. Se descarga de memoria el contexto del proceso.

<sup>7</sup>Cuando se intente realizar una modificación en alguno de los segmentos, se reserva memoria para éste, se copia su contenido y posteriormente se modifica la copia.

En la figura 3.10 se ve de forma gráfica la utilización de las llamadas `fork`, `exec`, `wait` y `exit`.



**Figura 3.10:** Sincronización entre proceso padre e hijo

### 3.7.6 Hilos

LINUX es un sistema multiproceso y multihilo, siguiendo el modelo visto en el apartado 3.6. Para hacer uso de los hilos se dispone de una biblioteca de funciones que nos permite, entre otras cosas, crearlos, sincronizarlos, etc. La tabla 3.4 nos muestra algunas de las funciones que proporciona.

Nombre	Descripción
<code>pthread_create</code>	Crea un hilo.
<code>pthread_join</code>	Espera la terminación del hilo dado como argumento.

**Tabla 3.4:** Algunas funciones de LINUX relacionadas con los hilos

La función `pthread_create` crea un bloque de control para el nuevo hilo mediante la llamada `clone`. El nuevo hilo comparte todos los recursos asignados al proceso mediante punteros a las estructuras del proceso (figura 3.11).

En las figuras 3.12, 3.13, 3.14, 3.15 y 3.16 podemos ver el funcionamiento interno de los hilos. En la primera tenemos un proceso normal que incorpora una serie de funciones para crear hilos.

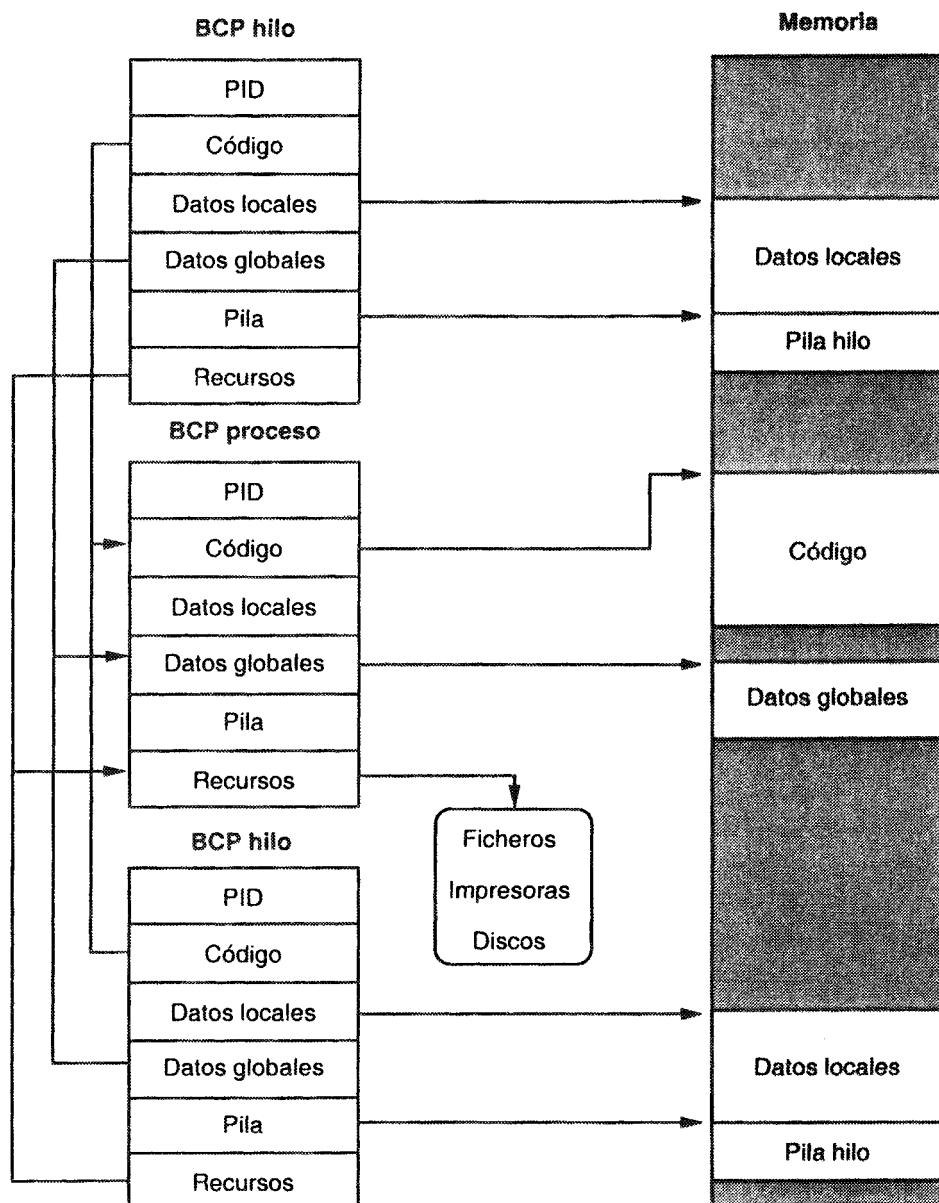
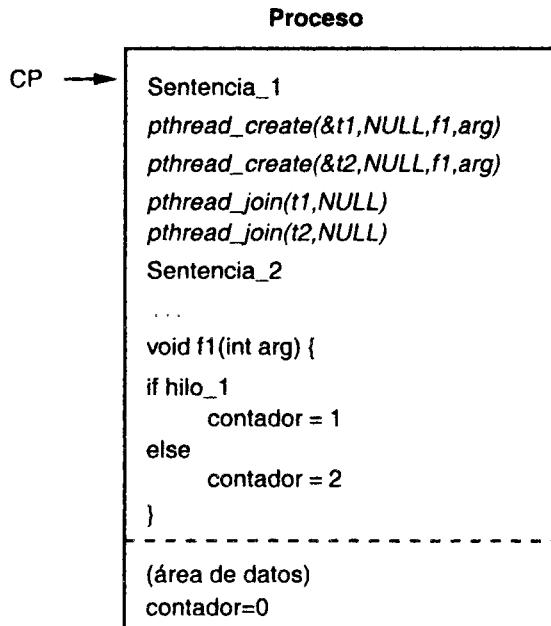
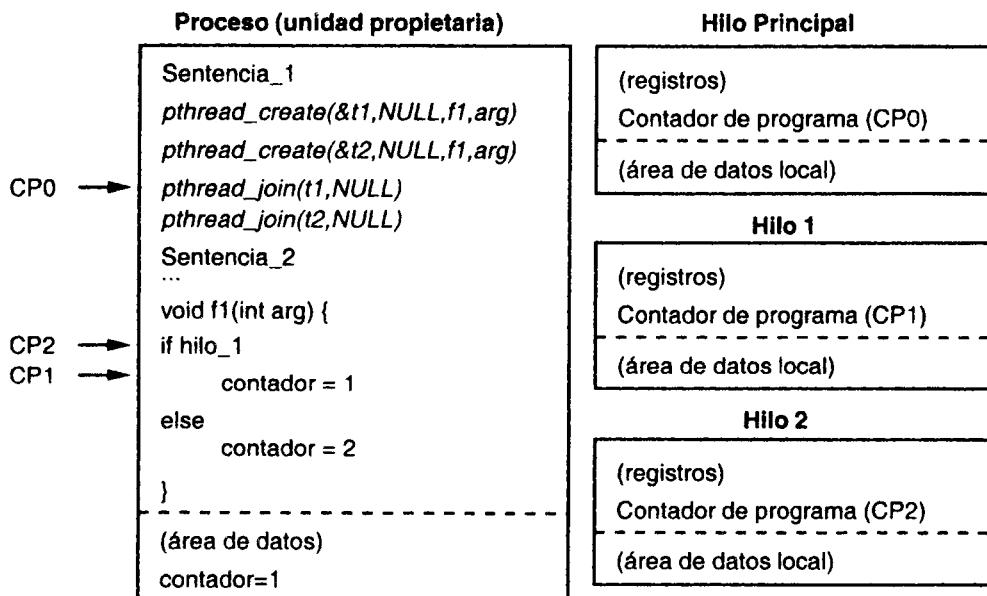


Figura 3.11: Relación entre el proceso y el hilo



**Figura 3.12:** Ejemplo de creación de hilos (paso 1)



**Figura 3.13:** Ejemplo de creación de hilos (paso 2)

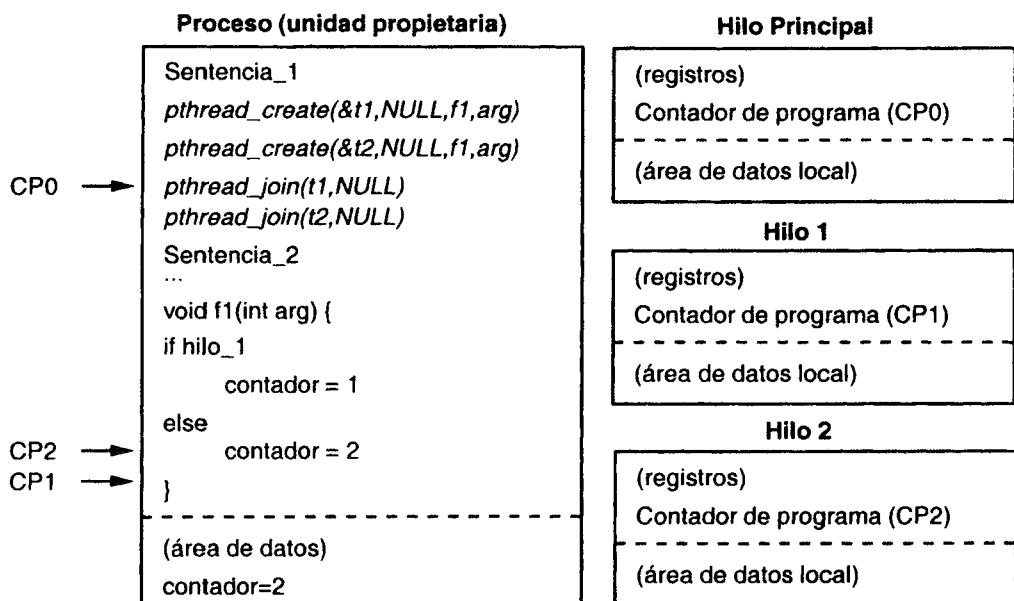


Figura 3.14: Ejemplo de creación de hilos (paso 3)

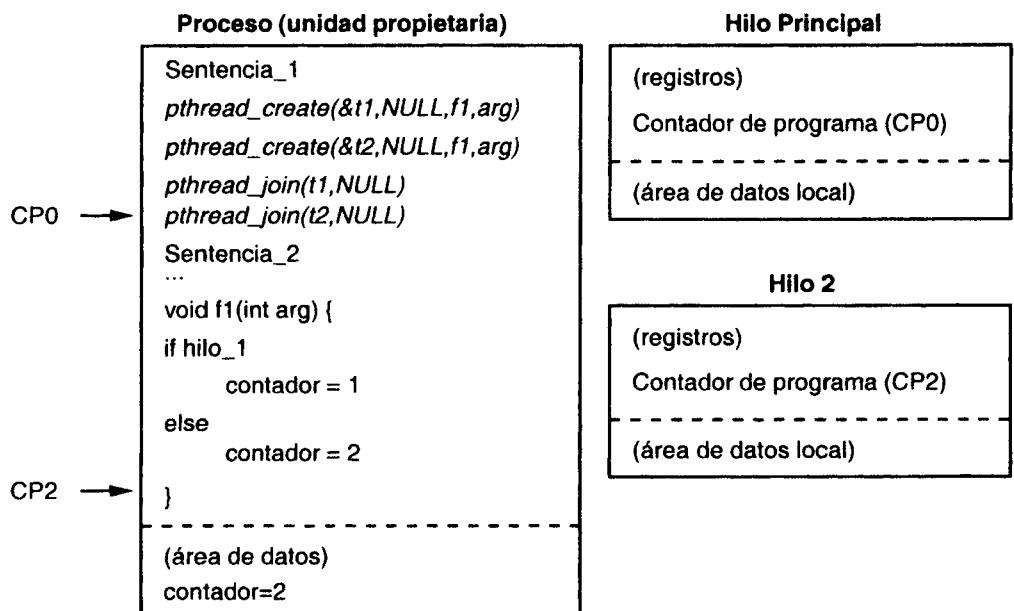
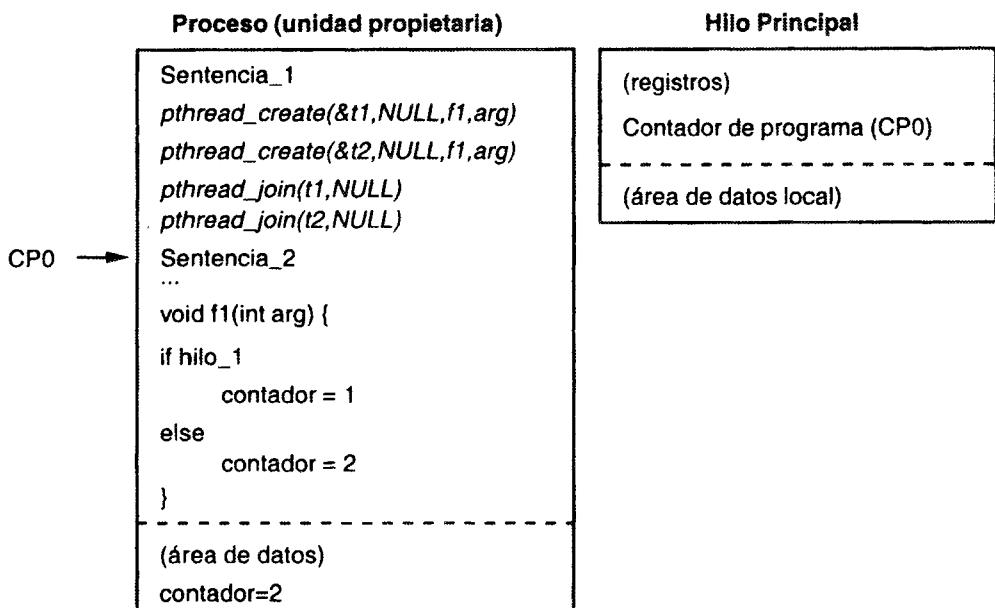


Figura 3.15: Ejemplo de creación de hilos (paso 4)



**Figura 3.16:** Ejemplo de creación de hilos (paso 5)

En la figura 3.13, nuestro proceso, con la función `pthread_create`, ha creado dos hilos y está ejecutando la función `pthread_join`. En ese momento, distinguimos el denominado **hilo principal**, que será el encargado de llevar la ejecución del proceso original. Si este hilo finaliza, terminan todos los demás. Podemos decir que se establece una especie de jerarquía entre hilos. De este modo, disponemos de los dos hilos creados para ejecutar la función `f1`, y del hilo principal que espera la terminación del primero. Tenemos dos hilos activos (hilos 1 y 2), y uno bloqueado (hilo principal). Si se ejecuta la orden `ps` en el sistema, se pueden visualizar tanto los hilos como la unidad propietaria de recursos.

En la figura 3.14 continúan ejecutándose los dos hilos y podemos ver cómo comparten los recursos a través de la unidad propietaria o proceso. Para ello, basta observar como ha evolucionado la variable `contador` entre las figuras 3.13 y 3.14.

En la figura 3.15 finaliza el hilo 1. En ese momento, el hilo principal, al ejecutar la función `pthread_join` esperará la finalización del hilo 2. Finalmente, en la figura 3.16, finaliza el hilo 2, y el hilo principal continúa con la ejecución.

## 3.8 Resumen

La unidad de funcionamiento del sistema es el proceso. Éste, al contrario que el programa, es dinámico, pasando durante su vida por una serie de estados. El conjunto de estados posibles en un sistema y las transiciones que pueden darse entre ellos definen el modelo de estados.

El sistema operativo se tiene que encargar de la gestión de los procesos, es decir, de su creación, planificación, terminación, cambios de estado, asignación de recursos, etc. Para ello requiere de una serie de tablas, de procesos, memoria, E/S y ficheros. La estructura que mantiene toda la información que necesita el sistema sobre cada proceso es el bloque de control.

El sistema operativo se encarga de realizar dos operaciones fundamentales: el cambio de contexto y el cambio de proceso. El primero es el elemento principal en la multiprogramación encargándose de guardar los registros de la CPU, para posteriormente cuando vuelva a tomar la CPU pueda continuar con su ejecución en el mismo punto por donde estaba. El segundo permite que cuando un proceso esté esperando algún evento se le pueda retirar la CPU y asignarla a otro.

Los sistemas operativos más recientes han introducido un nuevo concepto, el hilo. Éste permite separar las dos características englobadas en el concepto tradicional de proceso, la unidad propietaria de los recursos y la unidad de despacho.

## 3.9 Ejercicios

1. ¿Qué diferencias existen entre un proceso y un programa? ¿Y entre un proceso y un hilo?
2. ¿Por qué no tiene sentido mantener la lista de procesos bloqueados en orden de prioridad? ¿En qué circunstancias puede tener sentido ordenarla de esta forma?
3. Dibuje un diagrama de colas para un modelo de estados que incorpora los siguientes: nuevo, listo, suspendido listo, bloqueado, suspendido bloqueado, ejecutándose y terminado.
4. Explique las diferencias entre excepción, interrupción, error aritmético-lógico y llamada al sistema.
5. Explique las ventajas y desventajas de usar múltiples hilos en lugar de múltiples procesos. Sugiera una o más aplicaciones donde sea beneficioso el uso de los hilos.

6. Supongamos que nuestro sistema posee una cola de procesos bloqueados para cada evento que puede ocurrir. ¿Es posible que un proceso se quede bloqueado por dos o más eventos? Razone la respuesta y en caso afirmativo dé un ejemplo, y explique cómo solucionar el problema de tener un proceso bloqueado por dos o más eventos.
7. Diferencias entre la imagen de un proceso y su bloqueo de control.
8. ¿Por qué es necesario introducir dos estados suspendidos (suspendido bloqueado y suspendido listo) en lugar de uno sólo?
9. Diga si la siguiente afirmación es verdadera o falsa y razónela: «Un cambio de proceso da lugar siempre a un cambio de contexto, pero un cambio de contexto no tiene por qué ocasionar un cambio de proceso.»
10. ¿Qué relación existe entre la tabla de procesos, el bloqueo de control del proceso y las diferentes colas de procesos (listos, bloqueados, etc.)?
11. ¿Qué características presenta el modelo de estados de los procesos en LINUX?

# Capítulo 4

## Planificación

---

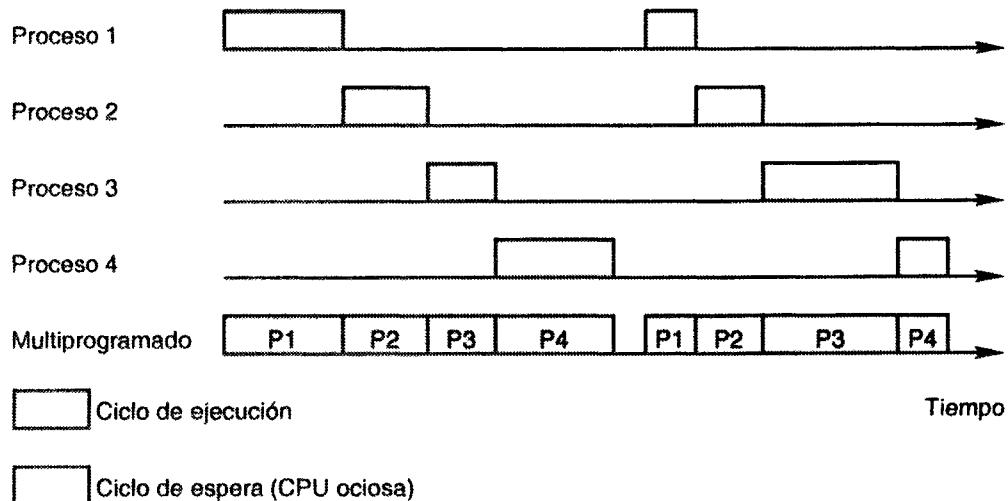
Un aspecto clave de los sistemas de multiprogramación es la planificación del procesador, ya que se ocupa de seleccionar entre todos los procesos que están a la espera a cuál de ellos se le va a asignar el procesador. En este capítulo se describirán los tres niveles de planificación existentes en los sistemas actuales, para centrarnos posteriormente en el funcionamiento y características de los principales algoritmos de planificación del procesador, así como los diferentes métodos para su evaluación.

### 4.1 Introducción

Un sistema de multiprogramación se caracteriza por la existencia de varios procesos residiendo simultáneamente en memoria principal. Dado que estamos considerando un sistema en el que sólo existe un procesador, los procesos se tienen que alternar en el uso de éste. Por este motivo, el sistema operativo debe ejercer una labor de planificación, determinando en cada instante qué proceso tomará el control del procesador.

La planificación produce un efecto de intercalamiento en la ejecución de los procesos, como se muestra en la figura 4.1. En ella, mientras un proceso espera a que termine de realizarse una operación de E/S, el procesador ejecuta otro. Esto permite aumentar el número de procesos que se ejecutan por unidad de tiempo,

es decir, el rendimiento del sistema (*throughput*).



**Figura 4.1:** Intercalamiento en la ejecución de varios procesos

## 4.2 Niveles de planificación

La planificación afecta al rendimiento del sistema porque determina qué procesos esperarán y cuales progresarán. Fundamentalmente, la planificación es una cuestión de manejo de colas para minimizar la demora de los procesos en éstas y optimizar el rendimiento. En muchos sistemas, la actividad de planificación se divide en tres tipos: **planificación a largo, medio y corto plazo**, o también podemos encontrarnos con la denominación planificación de alto nivel, nivel intermedio y bajo nivel, respectivamente.

La figura 4.2 relaciona los distintos tipos de planificación con el diagrama de transiciones entre estados visto en el capítulo 3. La planificación a largo plazo toma la decisión de llevar o no un nuevo proceso al sistema y cuál. La planificación a medio plazo es una parte de la función de intercambio. Se encarga de tomar la decisión de añadir o quitar un proceso a los que están disponibles para su ejecución. Finalmente, la planificación a corto plazo es la que se ejecuta con más frecuencia, y decide qué proceso, de todos los que se encuentran en estado listo, será el siguiente en ejecutarse.

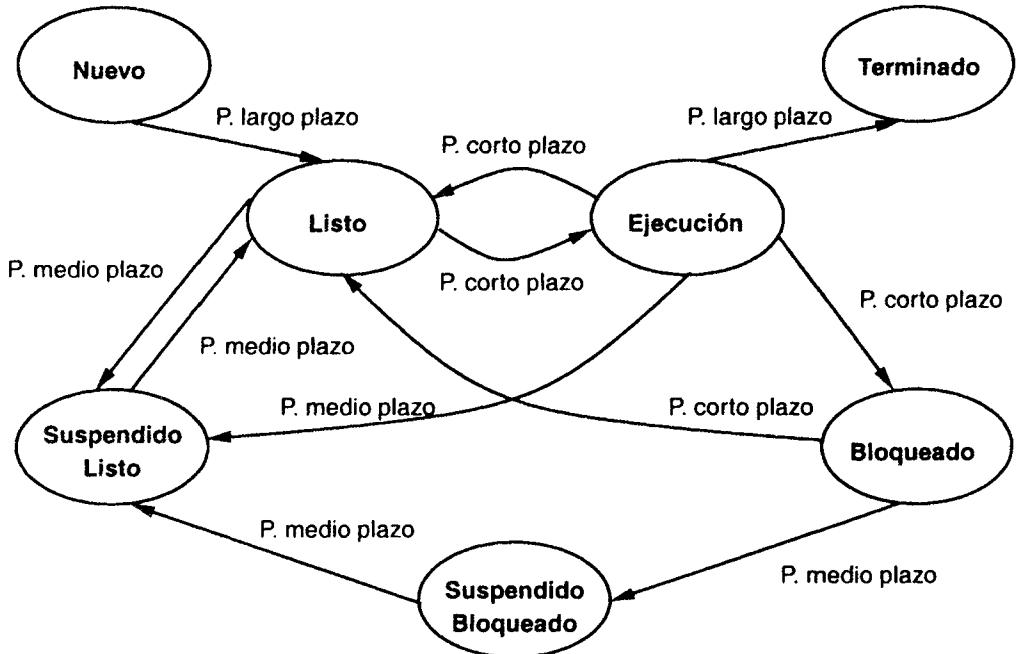


Figura 4.2: Planificación y transiciones de estado de los procesos

#### 4.2.1 Planificación a largo plazo

El planificador a largo plazo determina qué programas serán admitidos en el sistema para su ejecución, controlando el grado de multiprogramación. Cuando se admite un nuevo programa se convierte en proceso y se añade a la cola de listos, gestionada por el planificador a corto plazo.

En un sistema el planificador a largo plazo toma dos tipos de decisiones, si puede o no aceptar un nuevo proceso y, si esto es posible, cuál será.

Para tomar la decisión de crear o no un proceso nuevo se suele tener en cuenta el grado de multiprogramación, ya que cuantos más procesos haya en el sistema menos tiempo que se le dedicará a cada uno. Por otro lado, si el grado de multiprogramación es muy bajo el procesador puede llegar a estar ocioso si están todos bloqueados a la espera de algún evento. El planificador a largo plazo va a controlar el grado de multiprogramación para ofrecer un buen servicio a los procesos activos y para obtener un buen rendimiento del sistema.

La decisión de qué trabajo admitir depende del tipo de sistema operativo. En un sistema por lotes, puede hacerse en el orden de llegada o usar otro criterio,

tales como la prioridad, tiempo de ejecución estimado, etc. En un sistema de tiempo compartido, cuando un usuario intenta conectarse se genera una petición de creación de un proceso. Si el sistema no ha alcanzado el límite máximo de usuarios permitidos, se crea el proceso; en caso contrario, se niega la petición de conexión y se indica mediante un mensaje.

#### 4.2.2 Planificación a medio plazo

La planificación a medio plazo es parte de la función de intercambio. Normalmente, la decisión de pasar un proceso de memoria principal a memoria secundaria está basada en la necesidad de modificar el grado de multiprogramación. En el capítulo 8 se verán los aspectos implicados en el intercambio.

#### 4.2.3 Planificación a corto plazo

El objetivo principal de la planificación a corto plazo es asignar el procesador a los diferentes procesos que compiten por él, de forma que se optimicen uno o más aspectos del comportamiento del sistema. Para ello, el planificador a corto plazo evaluará en ciertos instantes una **función de selección** para todos los procesos listos. Ésta obtendrá un valor para cada proceso, de forma que al proceso que obtenga el valor más alto se le asignará la CPU. En caso de igualdad será necesaria una **regla de arbitraje** para resolver el conflicto. Si la probabilidad de que esto ocurra se considera baja, se podrá elegir el proceso al azar; en caso contrario, la regla de arbitraje deberá ser una nueva función de selección.

Los instantes en que se suele evaluar la función de selección, conocidos como **modo de decisión**, son los siguientes:

1. Cuando un proceso finaliza su ejecución.
2. Cuando un proceso cambia desde el estado de ejecución al estado bloqueado (por ejemplo, porque realiza una petición de E/S, o hace una llamada `wait` para esperar a que un proceso hijo termine).
3. Cuando un proceso cambia desde el estado de ejecución al estado de listo (por ejemplo, cuando ocurre una interrupción).
4. Cuando un proceso pasa desde el estado bloqueado, nuevo o suspendido-listo al estado listo (por ejemplo, se termina una operación de E/S).

Teniendo esto en cuenta, podemos distinguir dos tipos básicos de modos de decisión: **no apropiativo** y **apropiativo**.

Cuando la planificación tiene lugar sólo bajo las dos primeras circunstancias, tenemos una política de planificación no apropiativa, es decir, una vez que un proceso comienza a ejecutarse sólo se le puede retirar la CPU si se bloquea o termina.

En los demás casos se habla de planificación apropiativa, es decir, el proceso que se está ejecutando actualmente puede ser interrumpido y pasar a estado listo. Esta decisión la puede tomar el sistema operativo cuando llega un nuevo proceso, cuando un proceso bloqueado pasa a estado listo, cuando se activa un proceso suspendido, o periódicamente al producirse una interrupción del reloj.

Aunque las políticas apropiativas introducen mayor sobrecarga que las no apropiativas, suelen proporcionar un mejor servicio a los procesos. Esto es debido a que reparten mejor el tiempo del procesador entre los procesos, evitando que uno solo lo monopolice.

### 4.3 Algoritmos de planificación

A continuación describiremos distintas políticas de planificación de la CPU. Con objeto de compararlas, utilizaremos como ejemplo el conjunto de procesos que aparece en la tabla 4.1. Podemos considerar los procesos del ejemplo como trabajos por lotes, siendo el tiempo de servicio su tiempo total de ejecución. Otra posibilidad es considerarlos como procesos en curso que de forma alternativa usan el procesador y realizan operaciones de E/S; en este caso, el tiempo de servicio representa la duración de la siguiente ráfaga de CPU.

Proceso	P1	P2	P3	P4	P5
Hora de llegada	0	1	3	5	8
Tiempo de servicio	4	7	3	6	2
Prioridad	4	1	2	3	5

Tabla 4.1: Conjunto de procesos

#### 4.3.1 Primero en llegar, primero en ser servido

El algoritmo de planificación de la CPU más simple es primero en llegar, primero en ser servido<sup>1</sup>. Se trata de una política de planificación no apropiativa, donde los

<sup>1</sup>Este algoritmo puede aparecer en la bibliografía con las siglas FIFO (*First In First Out*) o bien FCFS (*First Come First Served*).

procesos simplemente se van ejecutando en orden de llegada. Cuando un nuevo proceso llega al sistema se incorpora a la cola de procesos en estado listo; cuando el proceso actual deja de ejecutarse (bien porque termina su ejecución o porque se bloquea) se elige al más antiguo de la cola. De forma matemática, la función de selección para un proceso  $P_i$  se puede expresar como  $f(P_i) = t_e$ , donde  $t_e$  es el tiempo que lleva el proceso en la cola de listos.

La figura 4.3 muestra el diagrama de Gantt para la ejecución de los procesos de nuestro ejemplo. A partir de éste podemos determinar el tiempo de finalización de cada proceso, así como calcular el tiempo de retorno, es decir, el tiempo total que el proceso pasa en el sistema (el tiempo de servicio más el tiempo de espera). Una magnitud más significativa es el **tiempo de retorno normalizado**, que es el cociente entre el tiempo de retorno de un proceso y su tiempo de servicio. Este valor indica la demora relativa que sufre un proceso. El valor mínimo para este cociente es 1; valores mayores indican un nivel decreciente de servicio. Otro valor significativo es el tiempo de respuesta, que se define como el intervalo de tiempo que transcurre desde que se envía una petición hasta que se empieza a recibir respuesta.

P1	P2	P3	P4	P5
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22				

Figura 4.3: Algoritmo FIFO

La tabla 4.2 muestra los valores obtenidos para todos estos parámetros para cada proceso, así como los valores medios.

	P1	P2	P3	P4	P5	Media
T. de respuesta	0	3	8	9	12	6,4
T. de espera	0	3	8	9	12	6,4
T. de retorno	4	10	11	15	14	10,8
T. de retorno normalizado	1,00	1,43	3,67	2,50	7,00	3,12

Tabla 4.2: Resultados para el algoritmo FIFO

Este algoritmo al ser no apropiativo perjudica a los procesos que necesitan ráfagas de CPU cortas (limitados por la E/S) frente a los que necesitan ráfagas largas (limitados por la CPU). Esto es debido a que estos últimos utilizan la CPU durante intervalos largos de tiempo, en los cuales todos los demás procesos deben esperar. Esto hace que los tiempos de retorno normalizado para los procesos

cortos sean muy altos. Así, en el ejemplo anterior el proceso P5 con un tiempo de servicio de 2 unidades presenta un tiempo de retorno normalizado de 7.

Este inconveniente hace que FIFO por sí sola no sea una política de planificación atractiva, pero combinada con otras puede ser más interesante.

#### 4.3.2 El proceso más corto primero

Para reducir la ventaja que se le da a los procesos largos en FIFO se puede utilizar la política del proceso más corto primero<sup>2</sup>. Se trata de una política no apropiativa en la que se selecciona para tomar el control de la CPU, el proceso con el tiempo de procesamiento esperado más corto. Matemáticamente, la función de selección para un proceso  $P_i$  se puede definir como  $f(P_i) = 1/s$ , siendo  $s$  el tiempo de servicio estimado para la siguiente ráfaga de CPU del proceso.

La figura 4.4 muestra el orden en que se ejecutarían los procesos de nuestro ejemplo y la tabla 4.3 los valores obtenidos para los tiempos.

P1	P3	P4	P5	P2
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22				

Figura 4.4: Algoritmo SPN

	P1	P2	P3	P4	P5	Media
T. de respuesta	0	14	1	2	5	4,4
T. de espera	0	14	1	2	5	4,4
T. de retorno	4	21	4	8	7	8,8
T. de retorno normalizado	1,00	3,00	1,33	1,33	3,50	2,03

Tabla 4.3: Resultados para el algoritmo SPN

Este algoritmo favorece a los procesos limitados por la E/S frente a los limitados por la CPU. Si comparamos los resultados obtenidos con los de FIFO, podemos ver que el tiempo de retorno se ha reducido significativamente para los procesos más cortos (P5 y P3), mientras que el proceso más largo (P2) obtiene peor resultado que con el algoritmo FIFO. Sin embargo, el servicio global a los procesos mejora puesto que los valores medios para los tiempos de respuesta, de retorno y de retorno normalizado han disminuido.

<sup>2</sup>Este algoritmo puede aparecer en la bibliografía con las siglas SPN (*Shortest Process Next*) o bien SJF (*Shortest Job First*).

Al favorecer a los procesos con ráfagas de CPU cortas, este algoritmo puede presentar el **bloqueo indefinido** o **muerte por inanición** de los procesos que necesitan ráfagas de CPU largas; es decir, un proceso de estas características se puede quedar esperando indefinidamente el uso de la CPU mientras estén llegando a la cola de listos procesos que necesitan ráfagas cortas.

SPN hace uso del tiempo de servicio estimado para la siguiente ráfaga de CPU. En los sistemas por lotes se puede pedir al programador que lo suministre al sistema. Para prevenir un uso incorrecto del sistema, si la estimación del programador está muy por debajo del valor real el sistema puede dar por finalizada la ejecución del proceso.

Sin embargo, en los sistemas interactivos este parámetro es difícil de conocer a priori, por lo que se hace necesario disponer de un método de estimación. El más simple se basa en la duración de las ráfagas anteriores:

$$S_{n+1} = \frac{1}{n} \sum_{i=1}^n T_i$$

donde:

$T_i$  es la duración real de la  $i$ -ésima ráfaga de CPU.

$S_i$  es el valor estimado para la  $i$ -ésima ráfaga.

$S_1$  es el valor estimado para la primera, que no se puede calcular.

Para facilitar los cálculos, se puede escribir la ecuación anterior de la siguiente forma:

$$S_{n+1} = \frac{1}{n} T_n + \frac{n-1}{n} S_n$$

#### 4.3.3 Tiempo restante más corto

El algoritmo del tiempo restante más corto<sup>3</sup> es la versión apropiativa del SPN. Por tanto, cada vez que un proceso llega a la cola de listos se evalúa la función de selección, que es el tiempo de ejecución que le queda para completar la ráfaga actual. Así, si llega a la cola de listos un proceso con una ráfaga de CPU menor que

---

<sup>3</sup>Se suele conocer con las siglas SRT (*Shortest Remaining Time*).

el tiempo de servicio que le queda al que se está ejecutando, éste será apropiado por el nuevo proceso. Matemáticamente, la función de selección para un proceso  $P_i$  se puede definir como  $f(P_i) = 1/(s - e)$ , donde  $s$  es el tiempo de servicio total estimado para la siguiente ráfaga de CPU del proceso y  $e$  el tiempo de ejecución que ya ha dedicado a esa ráfaga. Al igual que en SPN, es necesario estimar los tiempos de servicio de cada ráfaga de CPU del proceso.

La figura 4.5 muestra el patrón de ejecución de los procesos del ejemplo y en la tabla 4.4 aparecen los resultados que se obtienen.

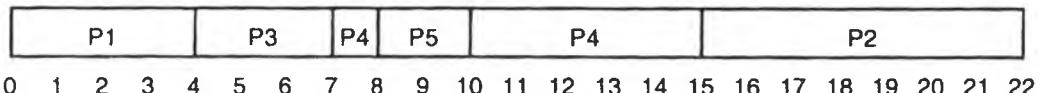


Figura 4.5: Algoritmo SRT

	P1	P2	P3	P4	P5	Media
T. de respuesta	0	14	1	2	0	3,4
T. de espera	0	14	1	4	0	3,8
T. de retorno	4	21	4	10	2	8,2
T. de retorno normalizado	1,00	3,00	1,33	1,67	1,00	1,60

Tabla 4.4: Resultados para el algoritmo SRT

SRT proporciona un servicio inmediato a los procesos con ráfagas de CPU más cortas. En nuestro ejemplo, los dos procesos más cortos (P4 y P5) obtienen un tiempo de retorno normalizado de 1.

Al igual que el algoritmo SPN, puede presentar la inanición de los procesos con ráfagas de CPU largas, siendo más acusado aún que con SPN, debido a que incluso cuando el proceso largo tenga el control de la CPU, la llegada de un proceso corto provocará que se le quite.

Este algoritmo introduce una cierta sobrecarga en el sistema con respecto a los anteriores, ya que se deben registrar los tiempos de ejecución transcurridos.

#### 4.3.4 Tasa de respuesta más alta

El algoritmo de tasa de respuesta más alta<sup>4</sup> intenta minimizar el tiempo de retorno normalizado de cada proceso. Para ello, define la **tasa de respuesta** como el

<sup>4</sup>Puede aparecer con las siglas HRRN (*Highest Response Ratio Next*).

cociente entre el tiempo de espera más el de servicio y el tiempo de servicio. La función de selección para un proceso  $P_i$  será  $f(P_i) = (t_e + s)/s$ , donde  $t_e$  es el tiempo que lleva el proceso esperando en la cola de listos, y  $s$  es el tiempo de servicio estimado para la siguiente ráfaga de CPU. Al igual que ocurría con SPN y SRT, es necesario conocer de antemano el tiempo de servicio de los procesos.

Se trata de un algoritmo no apropiativo, por lo que sólo cuando un proceso termina o se bloquea, el planificador escoge el de tasa de respuesta más alta.

Este método favorece a los procesos más cortos, pero al tener en cuenta el tiempo de espera se evita la inanición de los procesos largos.

En la figura 4.6 se muestra el patrón de ejecución de los procesos del ejemplo siguiendo este algoritmo, recogiéndose los valores de los tiempos obtenidos en la tabla 4.5.

	P1	P2	P3	P4																			
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Figura 4.6: Algoritmo HRRN

	P1	P2	P3	P4	P5	Media
T. de respuesta	0	3	8	11	6	5,6
T. de espera	0	3	8	11	6	5,6
T. de retorno	4	10	11	17	8	10,0
T. de retorno normalizado	1,00	1,43	3,67	2,83	4,00	2,59

Tabla 4.5: Resultados para el algoritmo HRRN

Si comparamos los resultados obtenidos con los del algoritmo SPN, podemos observar que los procesos largos obtienen mejores tiempos de retorno normalizado (P2), mientras que los cortos obtienen tiempos ligeramente mayores (P5).

#### 4.3.5 Asignación por turnos

El algoritmo de asignación por turnos<sup>5</sup> fue diseñado especialmente para los sistemas de tiempo compartido. Atiende a los procesos en el orden de llegada a la cola de listos, pero introduce la apropiación basada en las interrupciones producidas por un reloj. Cuando el planificador elige un proceso para su ejecución, inicializa

<sup>5</sup>En la bibliografía en lengua inglesa se le denomina *Round Robin*.

el temporizador para producir una interrupción transcurrido cierto tiempo. Ese intervalo de tiempo se denomina **cuanto**. Si el proceso en ejecución agota el cuento, se producirá la interrupción y el planificador lo situará al final de la cola de listos, eligiendo el primer proceso de ésta para su ejecución. Si antes de agotar el cuanto el proceso abandona la CPU (se bloquea o termina), el planificador elegirá un nuevo proceso para su ejecución. En este caso, la función de selección coincide con la del algoritmo FIFO,  $f(P_i) = t_e$ .

El principal aspecto de diseño de este algoritmo es el tamaño del cuanto, es decir, el intervalo de tiempo de uso de la CPU que se le da a cada proceso. No existe una regla fija que determine cuál es su tamaño óptimo. Si elegimos un cuanto muy pequeño, los procesos cortos tendrán un tiempo de retorno relativamente pequeño, pero se producirán muchas interrupciones y cambios de proceso con la consiguiente sobrecarga que esto conlleva. Si se elige un cuanto muy grande, este algoritmo se comportaría como un FIFO.

Este algoritmo es especialmente idóneo para sistemas interactivos donde se requiere que los tiempos de respuesta sean cortos para todos los procesos.

Las figuras 4.7 y 4.8 muestra cómo se ejecutarían los procesos del ejemplo siguiendo el algoritmo de asignación por turnos con un cuanto de 2 y 4, respectivamente. En las tablas 4.6 y 4.7 aparecen los valores de los tiempos. Se puede observar como los procesos más cortos, como el P5, mejoran de forma significativa con este algoritmo.

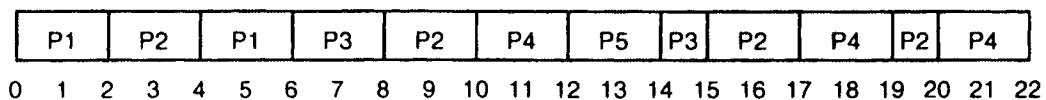


Figura 4.7: Algoritmo RR con cuanto de 2 unidades

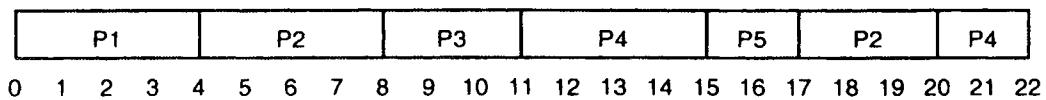


Figura 4.8: Algoritmo RR con cuanto de 4 unidades

Si comparamos los resultados obtenidos en los ejemplos anteriores (tablas 4.6 y 4.7), podemos observar que al aumentar el cuanto el tiempo medio de respuesta aumenta. Por otro lado, al aumentar el cuanto, se verán favorecidos aquellos procesos cuya ráfaga de CPU sea menor o igual que el nuevo cuanto (P3), disminuyendo su tiempo de retorno normalizado. Sin embargo, los procesos cuya

	P1	P2	P3	P4	P5	Media
T. de respuesta	0	1	3	5	4	2,6
T. de espera	2	12	9	11	4	7,6
T. de retorno	6	19	12	17	6	12,0
T. de retorno normalizado	1,50	2,71	4,00	2,83	3,00	2,81

**Tabla 4.6:** Resultados para el algoritmo RR con un cuento de 2 unidades

	P1	P2	P3	P4	P5	Media
T. de respuesta	0	3	5	6	7	4,2
T. de espera	0	12	5	11	7	7,0
T. de retorno	4	19	8	17	9	11,4
T. de retorno normalizado	1,00	2,71	2,67	2,83	4,50	2,74

**Tabla 4.7:** Resultados para el algoritmo RR un cuento de 4 unidades

ráfaga de CPU era menor o igual que el cuento pequeño (P5) ahora tendrán que esperar más, aumentando su tiempo de retorno normalizado.

#### 4.3.6 Prioridades

Un aspecto importante de la planificación es el uso de prioridades. En muchos sistemas, a cada proceso se le asigna una prioridad, y el planificador elegirá siempre un proceso de mayor prioridad, por tanto, la función de selección será  $f(P_i) = p_i$ , donde  $p_i$  es la prioridad del proceso.

La planificación por prioridad puede ser apropiativa o no apropiativa. Si se está utilizando un algoritmo apropiativo, cuando llega un proceso a la cola de listos, se compara su prioridad con la del que se está ejecutando actualmente; si la prioridad del nuevo proceso es mayor, se le retirará la CPU al proceso actual para cedérsela al nuevo. Un algoritmo no apropiativo pondrá al nuevo proceso en la cola de listos.

En el ejemplo que estamos utilizando se ha dado una prioridad a cada proceso, donde el número mayor indica una prioridad superior. Las figuras 4.9 y 4.10 muestran cómo se ejecutarían los procesos del ejemplo siguiendo un algoritmo por prioridades no apropiativo y apropiativo, respectivamente. Las tablas 4.8 y 4.9 muestran los resultados obtenidos para estos algoritmos.

Comparando los resultados para las dos versiones del algoritmo podemos ver cómo el proceso de mayor prioridad (P5) se ve favorecido en la versión apropiativa,

P1	P3	P4					P5	P2														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Figura 4.9: Algoritmo por prioridades no apropiativo

P1	P3	P4	P5	P4	P3	P2																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

Figura 4.10: Algoritmo por prioridades apropiativo

	P1	P2	P3	P4	P5	Media
T. de respuesta	0	14	1	2	5	4,4
T. de espera	0	14	1	2	5	4,4
T. de retorno	4	21	4	8	7	8,8
T. de retorno normalizado	1,00	3,00	1,33	1,33	3,50	2,03

Tabla 4.8: Resultados para el algoritmo de prioridades no apropiativo

	P1	P2	P3	P4	P5	Media
T. de respuesta	0	14	1	0	0	3,0
T. de espera	0	14	9	2	0	5,0
T. de retorno	4	21	12	8	2	9,4
T. de retorno normalizado	1,00	3,00	4,00	1,33	1,00	2,07

Tabla 4.9: Resultados para el algoritmo de prioridades apropiativo

ya que disminuye su tiempo de retorno normalizado. Por contra, un proceso de baja prioridad (P3) incrementa su tiempo de retorno normalizado en la versión apropiativa. Además, el proceso de menor prioridad (P2) se ve discriminado en las dos versiones.

Al igual que en los algoritmos SPN y SRT también se puede dar la inanición de los procesos con una baja prioridad, siendo este problema más grave en la versión apropiativa. Una forma de evitarlo sería considerar el **envejecimiento** de los procesos. Esta técnica incrementa gradualmente la prioridad de los procesos que esperan, de forma, que pasado cierto tiempo su prioridad podría llegar a ser mayor que la de los demás, permitiéndole recuperar el control de la CPU. La función de selección que representaría esta técnica podría ser  $f(P_i) = p_i + \alpha \times t_e$ , donde  $p_i$

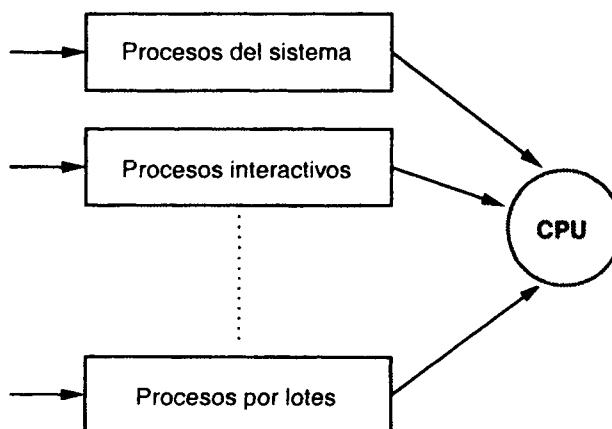
es la prioridad base del proceso,  $t_e$  es el tiempo que lleva esperando y  $\alpha$  es un coeficiente que regula el envejecimiento ( $\alpha \geq 1$ ).

#### 4.3.7 Planificación en varios niveles

Todas las estrategias estudiadas anteriormente siguen un único criterio de planificación. Sin embargo es posible combinar varios criterios en un algoritmo, un ejemplo de esto es la planificación en varios niveles.

En un sistema es posible encontrar procesos con diferentes necesidades de planificación, por ejemplo, las necesidades respecto al tiempo de respuesta no son las mismas para un proceso por lotes y para uno interactivo. Teniendo esto en cuenta se podrían clasificar los procesos según sus necesidades. Esto podría emplearse para dividir la cola de procesos listos en varias, cada una con una prioridad diferente. A su vez cada cola podría tener su propio algoritmo de planificación, que respondería a las necesidades de los procesos que van a ella.

Cuando un proceso entra al sistema se asigna a una cola, que será la empleada cada vez que se encuentre en estado listo. Cuando hay que elegir un proceso, se tomará uno de la cola de mayor prioridad; si durante su ejecución llegase un proceso a una cola con prioridad superior, tomaría el control de la CPU. Un posible implementación puede verse en la figura 4.11.



**Figura 4.11:** Planificación en varios niveles

#### 4.3.8 Planificación en varios niveles con realimentación

Otra posible combinación de estrategias de planificación es el algoritmo de varios niveles con realimentación. Una posible implementación de éste se puede ver en la figura 4.12.

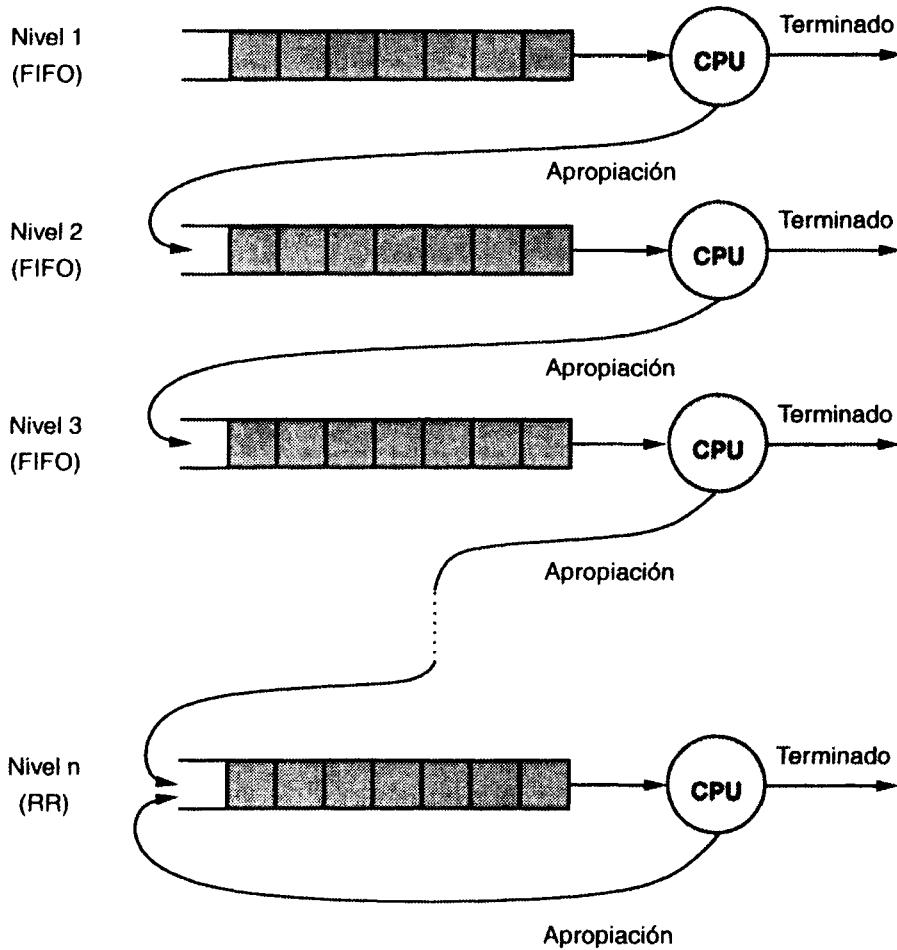


Figura 4.12: Planificación en varios niveles con realimentación

Cuando un proceso entra por primera vez en el sistema, se coloca en la cola de mayor prioridad. Cuando vuelve al estado listo después de haberse ejecutado por primera vez pasa a la cola de nivel inmediatamente inferior, y así sucesivamente va bajando a colas de prioridad inferior. Dentro de cada cola, excepto en la de más baja prioridad, se utiliza un algoritmo FIFO. Una vez que un proceso está

en la cola de menor prioridad no puede seguir bajando, por lo que siempre vuelve a ésta, es decir, se utiliza la asignación por turnos.

Hay múltiples variaciones de este esquema. Una posibilidad es establecer un cuento para cada cola, de forma que si el proceso lo agota pasa a la cola de nivel inferior. Los cuantos pueden ser iguales para todas las colas o diferentes. La figura 4.13 muestra el orden de ejecución de los procesos de nuestro ejemplo utilizando un cuento de una unidad en un sistema de tres colas. La tabla 4.10 muestra los valores que se obtienen para los tiempos.

P1	P2	P1	P3	P2	P4	P3	P4	P5	P1	P2	P3	P4	P1	P2	P4	P4	P2	P4	P2
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Figura 4.13: Planificación en 3 niveles con realimentación para  $q=1$

	P1	P2	P3	P4	P5	Media
T. de respuesta	0	0	0	0	0	0
T. de espera	11	14	7	10	0	8,4
T. de retorno	15	21	10	16	2	12,8
T. de retorno normalizado	3,75	3,00	3,33	2,67	1,00	2,75

Tabla 4.10: Resultados para 3 niveles con realimentación ( $q=1$ )

Como podemos ver, un proceso corto se completará de forma rápida sin bajar demasiado en la jerarquía de colas. Sin embargo, un proceso largo irá bajando más, pudiendo llegar a la última cola. Este algoritmo, por tanto, favorece a los procesos cortos frente a los largos, sin necesidad de conocer a priori su tiempo de servicio. El problema que aparece es el aumento que experimenta el tiempo de retorno para los procesos largos. Incluso es posible que éstos sufren inanición si regularmente van entrando nuevos procesos al sistema. Para compensar esto, podemos variar el valor del cuento según la cola, es decir, a medida que tenemos una cola de menor prioridad se le asignará un cuento mayor.

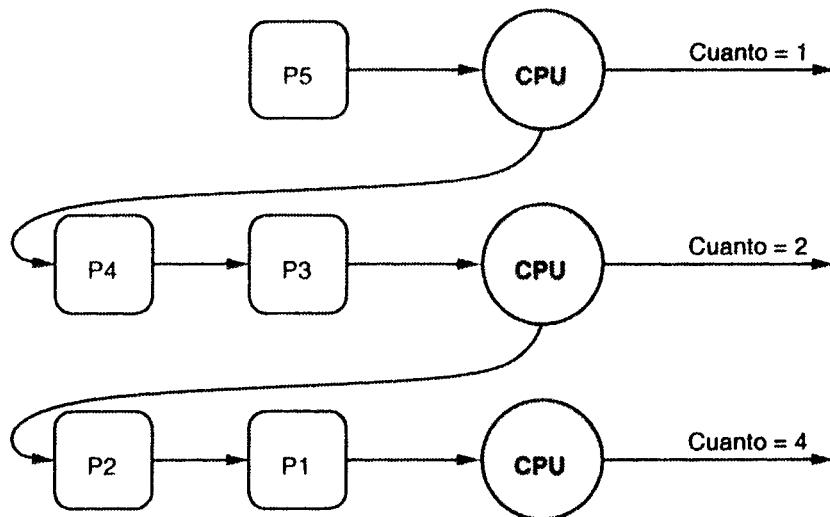
La figura 4.14 muestra el orden de ejecución de los procesos de nuestro ejemplo en un sistema con 3 colas y un cuento variable igual a  $2^n$  donde  $n$  es el número de la cola, siendo el valor inicial  $n = 0$ . La tabla 4.11 indica los resultados de los tiempos obtenidos. En la figura 4.15 podemos ver el estado de las distintas colas en el instante  $t = 8$ , cuando acaba de llegar el proceso P5.

El algoritmo de planificación de varios niveles con realimentación es el más general que se puede emplear, permitiendo adaptarlo a las necesidades de nuestro

P1	P2	P1	P3	P4	P2	P5	P3	P4	P5	P1	P2	P4	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figura 4.14: Planificación en 3 niveles con realimentación para  $q=2^n$ 

	P1	P2	P3	P4	P5	Media
T. de respuesta	0	0	1	0	0	0,2
T. de espera	11	11	5	11	4	8,4
T. de retorno	15	18	8	17	6	12,8
T. de retorno normalizado	3,75	2,57	2,67	2,83	3,00	2,96

Tabla 4.11: Resultados para 3 niveles con realimentación ( $q=2^n$ )Figura 4.15: Instante  $t = 8$  para el ejemplo de planificación en 3 niveles con realimentación para  $q=2^n$ 

sistema mediante la configuración de diversos parámetros. Entre éstos se encuentran:

- El número de colas.
- El algoritmo de planificación de cada cola.

- El método utilizado para determinar cuando se pasa un proceso a una cola de menor prioridad, por ejemplo, los procesos podrían dar más de una vuelta en la misma cola.
- El método utilizado para determinar cuando se pasa un proceso a una cola de mayor prioridad, para esto se podría fijar un tiempo máximo de espera de los procesos en cada cola.
- El método utilizado para determinar en qué cola entrará un proceso cuando se desbloquea.

## 4.4 Evaluación de algoritmos de planificación

En el apartado 4.3 hemos visto muchos algoritmos de planificación cada uno con sus propias características, la cuestión es ¿qué algoritmo de planificación elegiremos para un sistema particular? La selección de uno puede ser una tarea difícil.

El primer problema que nos encontramos es definir el criterio que vamos a usar para seleccionar el algoritmo. A continuación veremos algunos de los que podemos emplear para el planificador a corto plazo.

### 4.4.1 Criterios del planificador a corto plazo

Se puede establecer un conjunto de criterios para la elección del algoritmo de planificación. Éstos se pueden clasificar en dos grupos: los orientados al usuario y los orientados al sistema. Los primeros están relacionados con el comportamiento de un sistema tal como lo percibe el propietario de un proceso individual. Los segundos tienen como objetivo principal es el uso eficiente del procesador.

También se pueden clasificar los criterios de otra forma: aquellos que están relacionados con el rendimiento del sistema y los que no lo están. Normalmente los primeros son cuantitativos, mientras que los segundos son cualitativos.

En las tablas 4.12 y 4.13 se hace un resumen de todos estos criterios de planificación. Los criterios son interdependientes, y es imposible optimizarlos todos simultáneamente, de forma que el diseño de una política de planificación implica un compromiso entre requisitos opuestos. Por ejemplo, para proporcionar un buen tiempo de respuesta se requerirá un algoritmo de planificación que cambie los procesos frecuentemente, incrementando así la carga del sistema y reduciendo el rendimiento.

La elección del criterio dependerá de la naturaleza y uso del sistema. Así, en la mayor parte de los sistemas interactivos se emplea el tiempo de respuesta. Por contra, en sistemas por lotes se busca mejorar el tiempo de retorno.

<b>Criterios relacionados con el rendimiento orientados al usuario</b>	
<b>Tiempo de respuesta</b>	Es el intervalo de tiempo desde que se envía una petición hasta que se empieza a atender.
<b>Tiempo de retorno</b>	Es el intervalo de tiempo entre el envío de un proceso y su terminación. Incluye el tiempo de ejecución más el tiempo de espera de los recursos, incluido el procesador.
<b>Plazos</b>	Para los sistemas de tiempo real su objetivo es terminar los procesos en un cierto plazo.
<b>Otros criterios orientados al usuario</b>	
<b>Previsibilidad</b>	Un proceso debería ejecutarse en la misma cantidad de tiempo, aproximadamente, independientemente de la carga del sistema.

**Tabla 4.12:** Criterios de planificación orientados al usuario

<b>Criterios relacionados con el rendimiento orientados al sistema</b>	
<b>Rendimiento</b>	Es el número de procesos terminados por unidad de tiempo. Este valor depende de la longitud promedio de los procesos, pero también se ve afectado por la política de planificación.
<b>Utilización del procesador</b>	Es el porcentaje de tiempo que el procesador está ocupado. Es un criterio importante para los sistemas de tiempo compartido; en sistemas monousuario y de tiempo real este criterio tiene menos importancia que otros.
<b>Otros criterios orientados al sistema</b>	
<b>Equidad</b>	Los procesos deberían tratarse de la misma forma, y ninguno debería quedar relegado.
<b>Prioridades</b>	Se favorece a los procesos de prioridad más alta.
<b>Ocupación de recursos</b>	Se deben mantener los recursos del sistema ocupados.

**Tabla 4.13:** Criterios de planificación orientados al sistema

Una vez decidido el criterio que se va a utilizar habrá que evaluar diferentes algoritmos frente a él. Existen diversos métodos de evaluación, tales como la analítica, la simulación y la implementación.

#### 4.4.2 Evaluación analítica

Este tipo de evaluación utiliza el algoritmo y la carga del sistema para producir una fórmula o número que evalúa el rendimiento del algoritmo para esa carga.

##### 4.4.2.1 Modelo determinista

Partiendo de una carga particular determina el rendimiento que obtendría con cada algoritmo. Es un método simple pero requiere conocer con exactitud la carga que tendrá el sistema.

El ejemplo que hemos ido desarrollando durante el estudio de los diferentes algoritmos de planificación es una muestra de este tipo de evaluación. En la tabla 4.14 podemos ver una comparativa de los resultados obtenidos para todos los algoritmos, donde  $t_r$  es el tiempo de respuesta,  $t_e$  es el tiempo de espera,  $t_R$  es el tiempo de retorno y  $t_{RN}$  el tiempo de retorno normalizado. Así, el algoritmo SRT obtiene mejores resultados que los demás excepto en el tiempo de respuesta, donde el realimentado con un cuanto de 1 en todos los niveles obtiene mejor resultado.

Algoritmo	$t_r$	$t_e$	$t_R$	$t_{RN}$	Media
FIFO	6,4	6,4	10,8	3,12	6,68
SPN	4,4	4,4	8,8	2,83	4,91
SRT	3,4	3,8	8,2	1,60	4,25
Prioridades no apr.	4,4	4,4	8,8	2,03	4,91
Prioridades apr.	3,0	5,0	9,4	2,07	4,87
RR ( $q=2$ )	2,6	7,6	12,0	2,81	6,25
RR ( $q=4$ )	4,2	7,0	11,4	2,74	6,34
HRRN	5,6	5,6	10,0	2,59	5,95
Real. ( $q=1$ )	0	8,4	12,8	2,75	5,99
Real. ( $q=2^n$ )	0,2	8,4	12,8	2,96	6,09

**Tabla 4.14:** Comparación de resultados de distintos algoritmos de planificación

##### 4.4.2.2 Modelos de colas

Los procesos que se ejecutan en un sistema varían de un día para otro por lo que no suele ser posible el empleo del modelo determinista. Sin embargo, es más fácil determinar la distribución de tiempo de CPU y de operaciones de E/S que necesitan los procesos. A partir de estas distribuciones es posible calcular el

rendimiento promedio, la utilización de la CPU, el tiempo de espera, etc, para la mayoría de los algoritmos.

En este caso, el sistema de computación se describe como una red de servidores, teniendo asociada una cola de espera cada uno de ellos. La CPU es un servidor con una cola de procesos listos, así como el sistema de E/S con sus colas de dispositivos. Conociendo las velocidades de llegada y las de servicio, podemos calcular la utilización, la longitud promedio de la cola, el tiempo promedio de espera, etc. Esta área de estudio se conoce como **análisis de colas**, que queda fuera del ámbito de este libro.

#### 4.4.3 Simulación

Se simula mediante software un sistema de computación, donde los principales componentes del sistema se representan mediante estructuras de datos. El simulador tiene una variable que representa al reloj, a medida que el valor de esta variable aumenta, el simulador modifica el estado del sistema para reflejar las actividades de los dispositivos, los procesos y el planificador. Durante la simulación, se generan estadísticas mediante las cuales se realiza la evaluación del algoritmo de planificación.

#### 4.4.4 Implementación

Los métodos anteriores tienen una exactitud limitada. La única forma completamente exacta de evaluar un algoritmo de planificación es codificarlo y ponerlo en el sistema para ver cómo trabaja.

La principal dificultad de este método de evaluación es su costo. El gasto no sólo es el de codificación del algoritmo y la modificación del sistema operativo para soportarlo, sino también la reacción de los usuarios a los continuos cambios en el sistema. Un sistema de máquina virtual podría facilitar este tipo de evaluación.

### 4.5 Planificación de hilos

Los sistemas operativos actuales además de los procesos introducen los hilos (ver capítulo 3). La aparición de éstos influye en la planificación de la CPU, debido a que ahora la unidad de ejecución no es el proceso, sino el hilo. En estos sistemas pueden adoptarse dos alternativas respecto a la planificación de hilos:

**Planificación incorporada en el núcleo** En este caso el sistema es capaz de planificar un hilo o un proceso.

**Planificación realizada por una función de biblioteca** El núcleo se encarga de planificar procesos, y mientras un proceso tenga asignada la CPU, la biblioteca es la que incorpora las modificaciones necesarias para realizar una planificación entre los hilos de ese proceso; esto es lo que emplea el sistema LINUX.

La primera alternativa es más compleja de implementar pero presenta más ventajas. Una de ellas, es que en un sistema multiprocesador el núcleo es capaz de asignar hilos del mismo proceso a distintos procesadores, ganando en rendimiento; mientras que con la segunda todos los hilos del mismo proceso deben ejecutarse en el mismo procesador.

Otra ventaja de la primera alternativa es que se obtiene una planificación de hilos más equitativa. Basta con observar el siguiente ejemplo. Consideremos un proceso *A* con un único hilo, y otro proceso *B* con cien hilos. Supongamos que tenemos una planificación de hilos implementada en el espacio del usuario, donde cada proceso recibe un cuento de tiempo. Esto implica que durante el cuento de tiempo del proceso *A*, se ejecuta el único hilo que posee, mientras que durante el cuento del proceso *B*, el tiempo se lo tienen que repartir entre los cien hilos. Está claro que el hilo del proceso *A* se ejecutará unas cien veces más rápido que los otros, al recibir más tiempo de CPU. Si tenemos una planificación de hilos implementada en el propio núcleo, será el sistema operativo el que determine qué hilo se ejecutará, por lo que todos los hilos, con independencia del proceso al que pertenezcan, recibirán la misma cantidad de tiempo de uso del procesador.

## 4.6 Planificación en LINUX

Para los procesos de tiempo compartido se utiliza un algoritmo de asignación por turnos combinado con prioridades dinámicas, con un cuento de tiempo de 200 milisegundos. Para evitar que algún proceso no obtenga la CPU durante períodos muy largos, utiliza un contador que se actualiza cada vez que un proceso abandona el estado de ejecución. Éste es función de la prioridad del proceso y del tiempo que lleva en la cola de listos. Los valores de este contador se encuentran en el rango de 0 a 20.

El planificador de procesos elige para su ejecución el proceso con un contador más alto. Cuando un proceso llega a la cola de listos se comprueba si su contador es mayor que el del proceso en ejecución, en cuyo caso se solicita una reordenación

de los procesos listos, para adjudicar de nuevo el control de la CPU.

**Algoritmo 4.1****Planificador de procesos**

```

para todo proceso
hacer
  si el proceso tiene un intervalo de tiempo de ejecución
  entonces
    si el intervalo de tiempo ha expirado
    entonces
      Enviar una señal de alarma (SIGALRM) al proceso
      Si se requiere, restaurar el intervalo de tiempo
    fin si
  fin si
  si el proceso está bloqueado (TASK_INTERRUPTIBLE)
  entonces
    si el proceso ha recibido su señal o ha expirado su tiempo de bloqueado
    entonces
      Cambiar el proceso a estado de listo (TASK_RUNNING)
    fin si
  fin si
fin para
para todo proceso
hacer
  si el proceso está listo (TASK_RUNNING)
  entonces
    Comprobar si es el proceso que tiene el contador más alto
    fin si
fin para
si todos los procesos listos tienen el contador con valor cero
entonces
  para todo proceso
  hacer
    Recalcular el contador de acuerdo a su prioridad
  fin para
fin si
Otorgar el control de la CPU al proceso con el contador más alto

```

Cada vez que se agote el cuento de tiempo, al proceso que acaba de terminar se le decrementa en una unidad su contador. Si los procesos que están listos tienen un contador a 0 se recalcula el valor de éste, según la siguiente regla:

$$\text{contador} = \text{contador} - \text{anterior}/2 + \text{prioridad}$$

De este modo, se tiene en cuenta la prioridad y la historia del proceso. Ésta consiste en tomar la mitad del valor del contador la última vez que se recalcularon los contadores. Con la influencia de la prioridad, los procesos por lotes que suelen

tener menor prioridad que los interactivos, recibirán menos tiempo el uso de la CPU para favorecer la ejecución de los procesos con más prioridad.

Las prioridades van desde -20 a 20, siendo -20 la mayor prioridad. Se establece la prioridad 0 como la asignada a un proceso por omisión. Los usuarios pueden disminuir la prioridad de sus procesos, mientras que el administrador puede aumentarla. El rango de prioridades es:

- -20 a -1 para los procesos que se ejecutan en modo supervisor.
- 0 a 20 para los procesos en modo usuario.

A los procesos de tiempo real se les da un valor al contador de 1000 al que se añade su propia prioridad. De esta forma, el planificador ofrece garantías sobre las prioridades relativas, pero no sobre la rapidez de la ejecución del proceso. Esto hace que sólo sea válido para sistemas de tiempo real blando.

Un pseudocódigo del planificador de procesos aparece en el algoritmo 4.1.

#### 4.6.1 Planificación en sistemas multiprocesadores

LINUX soporta multiprocesamiento simétrico, es decir, un sistema capaz de distribuir el trabajo equilibradamente entre todos los procesadores del sistema. De este modo, los procesos se podrán ejecutar en distintos procesadores.

Para conseguir esta planificación, cada procesador ejecutará el planificador separadamente, pero con objeto de preservar la integridad del sistema, se impone la restricción de que sólo un procesador puede estar ejecutando un proceso en modo núcleo. El algoritmo empleado es el mismo, pero se da una ligera ventaja a un proceso que se haya ejecutado anteriormente en el mismo procesador, porque supone mucho trabajo adicional trasladarlo a otro.

En el bloque de control del proceso se guarda información sobre el procesador en el que se está ejecutando, así como el último procesador donde se ejecutó. Incluso se puede restringir que un proceso se pueda ejecutar en uno o más procesadores mediante el empleo de una máscara. De esta forma, cuando el planificador tenga que seleccionar un nuevo proceso para ejecutar, no podrá escoger entre aquellos que tengan su número de procesador activo en la máscara.

## 4.7 Resumen

La evolución dinámica de los procesos en el sistema consiste en una serie de transiciones entre los diferentes estados por los que puede pasar: nuevo, listo, bloqueado, terminado, ejecución, etc. Los planificadores son los elementos del sistema operativo encargados de realizar estas transiciones.

Existen tres planificadores en el sistema: el de largo plazo, medio plazo y corto plazo. Cada uno tiene un objetivo distinto. El primero es el responsable de la entrada y salida al sistema, es decir, de la admisión de nuevos procesos.

El planificador a medio plazo tiene la función de suspender o activar procesos en función de las necesidades del sistema. De este modo, estos dos planificadores son los encargados de controlar el grado de multiprogramación.

Finalmente, el planificador a corto plazo se encarga de otorgar el control del procesador a uno de los procesos que se encuentran en estado listo. Este planificador es uno de los elementos que más se ejecutan dentro del sistema operativo, y tiene especial importancia en su rendimiento, debido a que si un proceso pasa a estado bloqueado, sobre él recae la responsabilidad de que el sistema pueda continuar ejecutando otro proceso.

La existencia de numerosos algoritmos para el planificador a corto plazo (FIFO, SPN, SRT, RR, HRRN, etc.) hace necesario establecer qué criterios nos van a permitir evaluarlos para decidir cuál es el más adecuado para nuestro sistema.

## 4.8 Ejercicios

1. Entre los siguientes algoritmos de planificación: FIFO, SRT, SJF, HRRN, Multinivel, Multinivel con realimentación y Prioridad, determine cuál de ellos necesitan para su funcionamiento una información previa de cada proceso, y qué tipo de información.
2. En relación a los algoritmos de planificación FIFO, RR y Multinivel con realimentación, explique las diferencias existentes entre ellos en relación al modo de discriminar los trabajos.
3. Determine de los siguientes algoritmos cuáles son idóneos para un planificador a largo plazo, y cuáles para un planificador a corto plazo. Explique sus razones.
  - (a) FIFO.
  - (b) RR.

- (c) SJF.  
(d) Prioridad.
4. Explique qué tipo de planificación (apropiativa o no apropiativa) utilizaría en los siguientes sistemas:
- (a) Controlador aéreo.  
(b) Central de alarmas.  
(c) Gestión de préstamos de una biblioteca.  
(d) Proceso de matrícula de una universidad.
5. Suponga los siguientes procesos:
- | Proceso | Tiempo de servicio |
|---------|--------------------|
| P1      | 24 s               |
| P2      | 3 s                |
| P3      | 3 s                |
- Si empleamos un algoritmo FIFO, calcule el tiempo medio de retorno si el orden de llegada es  $P_1$ ,  $P_2$ , y  $P_3$ . Igual si el orden es  $P_3$ ,  $P_2$ ,  $P_1$ . ¿Qué puede decir acerca de la influencia del orden de llegada de los procesos en el algoritmo FIFO?
6. ¿Es posible que el algoritmo de planificación SJF pueda provocar la inanición de los procesos largos? En caso afirmativo, indique un ejemplo.
7. ¿Qué criterio de planificación incumple el algoritmo SRT? Explique los motivos.
8. ¿Qué relación existe en un algoritmo de asignación por turnos el número de cambios de procesos con el tamaño del cuanto? Razona la respuesta.
9. ¿Qué elementos posee un planificador a corto plazo? ¿Cuáles son sus funciones?
10. ¿Qué ventajas e inconvenientes presenta que un algoritmo de planificación por prioridades disponga o no la técnica del envejecimiento? Razona la respuesta.
11. Supongamos que tenemos que ejecutar los siguientes procesos en un procesador, donde la prioridad más alta es la prioridad máxima:

Proceso	Hora de llegada	Tiempo de servicio	Prioridad
1	0	8	3
2	1	4	1
3	2	9	2
4	3	5	4

- (a) Realice un diagrama de Gantt que ilustre la ejecución de estos procesos utilizando la planificación FIFO. Calcule los tiempos de respuesta, espera, retorno y retorno normalizado para cada proceso, así como los respectivos tiempos medios.
- (b) Igual al anterior pero para un algoritmo SPN.
- (c) Igual al anterior pero para un algoritmo SRT.
- (d) Igual al anterior pero para un algoritmo HRRN.
- (e) Igual al anterior pero para un algoritmo de asignación por turnos, con cuanto de 3 unidades.
- (f) Igual al anterior pero para un algoritmo de prioridades apropiativo.
12. Dada la siguiente secuencia de procesos a ser ejecutados en un sistema monoprocesador, donde la prioridad más alta es la máxima prioridad:
- | Proceso | Tiempo de servicio | Prioridad | Hora de llegada |
|---------|--------------------|-----------|-----------------|
| 1       | 10                 | 3         | 0               |
| 2       | 1                  | 1         | 1               |
| 3       | 2                  | 3         | 2               |
| 4       | 1                  | 4         | 3               |
| 5       | 5                  | 2         | 3               |
- (a) Realice un diagrama de Gantt que ilustre la ejecución de estos procesos utilizando la planificación FCFS. Calcule los tiempos de respuesta, espera, retorno y retorno normalizado para cada proceso, así como los respectivos tiempos medios.
- (b) Igual al anterior pero para un algoritmo SJF.
- (c) Igual al anterior pero para un algoritmo SRT.
- (d) Igual al anterior pero para un algoritmo HRRN.
- (e) Igual al anterior pero para un algoritmo de asignación por turnos, con cuanto de 3 unidades.
- (f) Igual al anterior pero para un algoritmo de prioridades apropiativo.
13. Suponga un sistema multinivel con tres colas. Los procesos ingresarán en una cola en función de la prioridad. Si la prioridad es menor de 3 se pondrá en la tercera cola, si es menor de 6 se pondrá en la segunda cola, y en caso contrario en la primera cola.

El algoritmo de planificación entre colas es por prioridades apropiativo. Además, la primera cola emplea el algoritmo de prioridades apropiativo, la segunda cola utiliza el algoritmo del trabajo más corto primero, y la tercera cola una asignación por turno ( $q=2$ ).

Dibujar el diagrama de Gantt, e indicar los tiempos de respuesta, retorno, espera y retorno normalizado para los procesos siguientes:

Proceso	Hora de llegada	Tiempo de servicio	Prioridad
P1	0	3	8
P2	2	5	3
P3	4	7	0
P4	6	4	9
P5	8	6	5
P6	10	2	1
P7	12	9	10

14. Dibujar el diagrama de Gantt y calcular los tiempos de espera, retorno, respuesta y retorno normalizado para los procesos del ejercicio anterior, suponiendo un sistema con una planificación multinivel con realimentación de cuatro colas, donde el cuento en cada cola viene dado por  $q = 2^n$ .

---

Parte 3

# CONCURRENCIA

---



# Capítulo 5

## Sincronización y comunicación

---

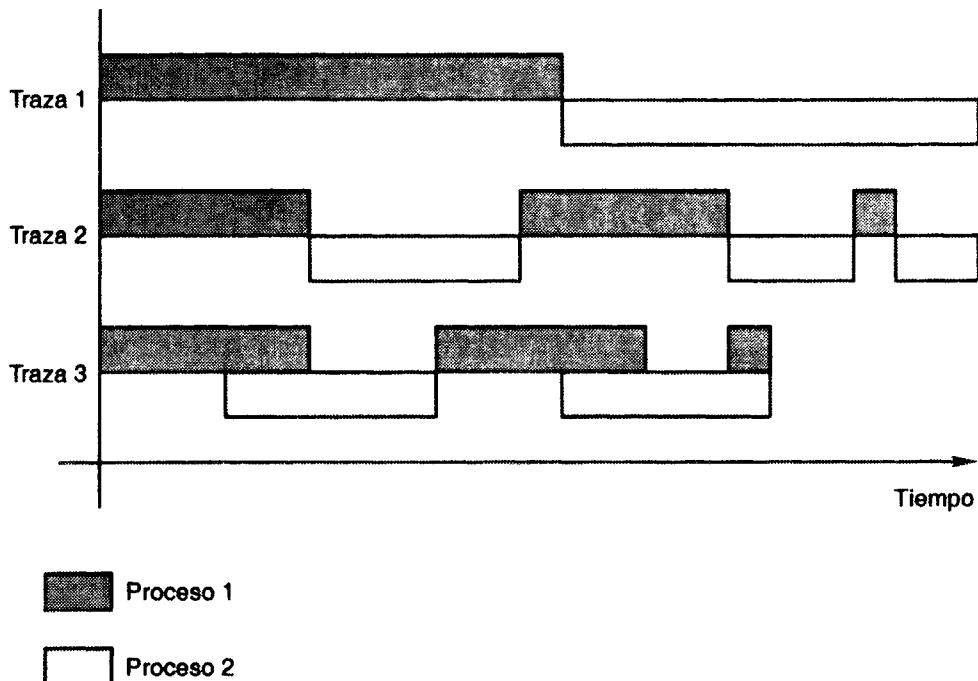
Los sistemas operativos actuales permiten la multiprogramación. Sin embargo, esta concurrencia simultánea de varios procesos en el sistema no está exenta de problemas, debido sobre todo a que no existen suficientes recursos para todos los procesos, algunos necesitan comunicarse o sincronizarse entre sí, etc. En este capítulo veremos los problemas que se presentan, así como las herramientas que se suelen utilizar para resolverlos.

### 5.1 Introducción

La multiprogramación ha permitido aumentar el rendimiento de los sistemas operativos al simultanear la ejecución de varios procesos, aprovechando de este modo los tiempos ociosos del procesador. Esto ha motivado que la traza de ejecución de los procesos a lo largo del tiempo pueda tener varias formas. La figura 5.1 muestra las distintas trazas de ejecución que pueden suceder para dos procesos cualesquiera.

La primera traza corresponde a un sistema sin multiprogramación, donde los procesos se ejecutan de forma secuencial. Es el más simple pero el que menor rendimiento obtiene.

La segunda traza corresponde a un sistema multiprogramado con un único



**Figura 5.1:** Trazas de ejecución de un conjunto de procesos

procesador, donde los procesos se **intercalan** en el tiempo para dar la apariencia de ejecución simultánea. Aunque no se consigue un procesamiento paralelo real y existe sobrecarga debida a los cambios de procesos, la ejecución intercalada produce beneficios importantes.

La tercera corresponde a un sistema multiprocesador, en éstos no sólo es posible el intercalamiento de procesos, sino que también se puede realizar el **solapamiento**.

En los dos últimos casos existe un procesamiento concurrente, puesto que pueden existir varios procesos de forma simultánea en el sistema. Esta concurrencia presenta peculiaridades diferentes dependiendo del tipo de sistema, aún así, todos se enfrentan a los mismos problemas, y existen mecanismos comunes de acción. Los problemas que plantea la concurrencia son debidos a la imposibilidad de predecir la velocidad relativa con la que se van a ejecutar los procesos y, por tanto, el orden en que van a ocurrir los sucesos. Sin embargo, lo que siempre debemos tener en cuenta es que los resultados de un proceso deben ser independientes de la velocidad relativa de ejecución de éste respecto a los demás procesos concurrentes.

Para comprender cómo se puede abordar la independencia de la velocidad, hace falta estudiar cómo interaccionan los procesos, así como las características de los distintos recursos que necesitan para su ejecución.

## 5.2 Recursos

En cualquier sistema de computación existen una serie de recursos en un número finito para compartir entre todos los procesos. Una de las funciones del sistema operativo es la gestión de éstos. Como ejemplo, podemos citar los dispositivos (impresoras, discos, cintas, etc.), procesadores, memoria principal, ficheros, datos, etc.

Programa 5.1	Utilización de un recurso
<pre>#include &lt;unistd.h&gt; #include &lt;sys/types.h&gt; #include &lt;sys/stat.h&gt; #include &lt;fcntl.h&gt; #define TAM_VECTOR 255  main() {     char vector[TAM_VECTOR];     int fd;      /* se solicita el recurso */     fd = open ("fichero", O_RDONLY);     /* se usa el recurso */     read (fd, vector, TAM_VECTOR);     printf("%s", vector);     /* se libera el recurso */     close(fd); }</pre>	

Durante la ejecución de un proceso, éste utiliza un conjunto de recursos. La secuencia de pasos que debe dar el proceso para utilizarlos es la siguiente:

1. **Solicitud** Cuando un proceso solicita un recurso, el sistema operativo comproba si éste ha sido pedido y asignado a otro. Para ello cuenta con una tabla del sistema donde se registra qué recursos están libres y cuáles asignados, y si es así, a qué proceso lo está. Si un proceso pide un recurso que está asignado a otro, se añade a una cola de procesos que están esperando ese recurso, es decir, pasa a estado bloqueado hasta que esté disponible. Si existen recursos del tipo solicitado, se le asigna.

2. **Uso** El proceso puede operar con el recurso.
3. **Liberación** Una vez que termina su utilización, el proceso libera el recurso.

La solicitud, uso y liberación de los recursos sólo se puede realizar a través de los servicios del sistema. Así, son ejemplos de solicitud y liberación de recursos las llamadas al sistema **open** y **close** de apertura y cierre de ficheros, respectivamente. Las llamadas **read** y **write** para la lectura y escritura en un fichero son ejemplos de uso de un recurso. El programa 5.1 muestra un ejemplo de solicitud, uso y liberación de un recurso; en este caso, un fichero.

Los recursos se pueden clasificar atendiendo a distintos criterios:

- En función del número de procesos que lo pueden utilizar:

**Recursos compatibles** Pueden ser usados por más de un proceso simultáneamente. Éstos no causan problemas de concurrencia entre procesos, pues pueden usarlos simultáneamente. Ejemplos de este tipo de recurso son los discos, ficheros de sólo lectura, etc.

**Recursos no compatibles o críticos** En este caso pueden ser usados por más de un proceso pero no simultáneamente como en el caso anterior. Aquí sí es necesario tener un control de la concurrencia entre los procesos. Ejemplos de este tipo son las impresoras, las cintas magnéticas, etc.

- En función del tiempo de vida del recurso:

**Recursos reutilizables** Son aquellos que no se agotan, ni destruyen con su uso. En este caso, los procesos obtienen un recurso que liberan posteriormente para que pueda ser reutilizado por otros procesos. Ejemplos de este tipo de recurso lo constituyen el procesador, memoria principal y secundaria, dispositivos, y estructuras de datos como ficheros, bases de datos, etc. Existe, por norma general, un número fijo de recursos reutilizables.

**Recursos consumibles** Son los que pueden ser creados (producidos) y destruidos (consumidos). Normalmente, no hay límite en el número de recursos consumibles de un determinado tipo. Cuando un proceso lo adquiere, éste deja de existir. Ejemplos de este tipo de recurso son las interrupciones, señales, mensajes, e información en un *buffer* de E/S.

## 5.3 Interacción entre procesos

La existencia de procesos concurrentes en un sistema permite establecer relaciones entre ellos, que pueden ser de varios tipos:

- Competencia.
- Compartición.
- Comunicación.

Esas relaciones puede dar lugar a diversos problemas. Veamos a continuación qué aspectos debe controlar el sistema operativo en cada relación, con objeto de solventarlos.

### 5.3.1 Competencia

Los procesos concurrentes pueden relacionarse entre sí de distintas formas. Podemos tener procesos independientes, entre los que pueden establecerse una relación de competencia. Es decir, los procesos compiten por el acceso a recursos no compatibles. Se puede describir la situación de la siguiente forma:

*Dos o más procesos necesitan acceder a un recurso durante su ejecución. Cada proceso ignora la existencia de los otros, y su resultado no debe verse afectado por la ejecución de éstos. Por tanto, cada proceso debería dejar tal y como esté el estado de cada recurso que utilice.*

Entre los procesos que compiten por el uso de un recurso no existe intercambio de información. Sin embargo, el comportamiento de un proceso puede verse afectado por la ejecución de otros competidores, en el sentido de que puede necesitar un recurso que esté ocupado por otro, por lo que pasará del estado de ejecución a un estado bloqueado.

En este caso, el sistema debe regular el acceso a los recursos que son requeridos por más de un proceso, es decir, ha de gestionarlos. Para ello ha de determinar quién accede primero a él, es decir, solucionar las **condiciones de competencia** sobre un mismo recurso. En este tipo de situaciones es necesario solucionar tres tipos de problemas.

El primer problema es la **exclusión mutua**, que se conoce también como el **problema de la sección crítica**. Supongamos que dos o más procesos quieren

acceder a un recurso crítico único, como la impresora. A la porción del programa que lo usa se le conoce como **sección crítica**. Es importante que no haya más de un proceso dentro de una sección crítica en un instante dado (siempre que en ellas se haga uso de los mismos recursos). No podemos confiar sólo en el sistema operativo para conseguir esto, sino que tendremos que forzarlo de forma expresa.

El forzar la exclusión mutua crea dos problemas adicionales. Uno de ellos es el **interbloqueo (deadlock)**, que será tratado en el capítulo 6. Un ejemplo de este tipo de problema aparece cuando dos procesos, P1 y P2, compiten por dos recursos críticos, R1 y R2. Cada proceso necesita acceder a ambos recursos para seguir ejecutándose. Podemos encontrarnos en una situación donde cada proceso tiene un recurso asignado y está esperando el otro (P1 con R1 y P2 con R2); como ninguno libera el recurso que tiene asignado ambos procesos están en un interbloqueo.

El tercer problema que aparece es la inanición o bloqueo indefinido. Consiste en que un proceso queda indefinidamente esperando la obtención de un recurso que se reparten entre otros procesos. Supongamos tres procesos P1, P2 y P3, que requieren un acceso periódico a un recurso R. Consideremos la situación en la que P1 está en posesión del recurso, y tanto P2 como P3 están esperando. Cuando P1 sale de su sección crítica, bien a P2 o a P3 se le debería dar acceso al recurso R. Si se le da acceso a P2 y antes de que éste complete su sección crítica, P1 lo requiere de nuevo, podría dársele el recurso a P1, y así turnarse P1 y P2 en el uso de éste, quedándose P3 esperando indefinidamente.

El control de la competencia implica necesariamente al sistema operativo porque es él quien asigna los recursos a los procesos. Por este motivo, ha de tener mecanismos para que los procesos expresen su necesidad de exclusión mutua. Más adelante veremos las distintas formas de hacerlo.

### 5.3.2 Compartición

Los procesos concurrentes también pueden cooperar mediante la compartición de recursos. En este caso, los procesos comparten el acceso al mismo objeto, como un *buffer* de E/S, variables compartidas, ficheros o bases de datos. Los procesos deben utilizar los datos compartidos de forma adecuada.

Dado que los datos están soportados sobre recursos, de nuevo aparecen los problemas comentados anteriormente: exclusión mutua, interbloqueo e inanición. La única diferencia es que los procesos pueden acceder a los datos de dos modos diferentes, lectura y escritura, y sólo las operaciones de escritura deben ser mutuamente exclusivas.

Sin embargo, aparte de estos problemas, se introduce un nuevo requisito: la **coherencia de los datos**. Consiste en proteger las zonas de actualización de los datos compartidos con el fin de que las actualizaciones realizadas sean correctas.

A continuación se presenta un ejemplo simple en el que varios datos deben ser actualizados. Supongamos dos variables  $p$  y  $q$  para las cuales debe mantenerse la condición  $p = q$ . Es decir, cualquier programa que actualice un valor debe también actualizar el otro para mantener la relación. Consideremos los siguientes procesos:

P1:       $p = p + 2;$   
               $q = q + 2;$

P2:       $q = 2 * q;$   
               $p = 2 * p;$

Si el estado inicial es consistente, cada proceso tomado de forma separada debe dejar los datos compartidos en un estado consistente. Consideremos ahora la siguiente ejecución concurrente, en la cual los dos procesos respetan la exclusión mutua sobre cada ítem individual ( $p$  y  $q$ ):

$p = p + 2;$   
 $q = 2 * q;$   
 $q = q + 2;$   
 $p = 2 * p;$

Al final de esta secuencia de ejecución, la condición  $p = q$  no se cumple. El problema puede ser evitado declarando la secuencia completa en cada proceso como una sección crítica, incluso cuando estrictamente hablando no está implicado ningún recurso crítico. Así, vemos que el concepto de sección crítica, planteado en la relación de competencia, es también importante en el caso de la cooperación por compartición.

### 5.3.3 Comunicación

Los procesos concurrentes también pueden comunicarse, manifestando una cooperación entre ellos. Típicamente la comunicación se puede realizar por medio de intercambio de mensajes. Las primitivas para enviar y recibir mensajes pueden ser proporcionadas como parte de un lenguaje de programación o por el núcleo del sistema operativo, mediante servicios del sistema.

Dado que los procesos no comparten nada en el acto de intercambio de mensajes, no es necesario controlar la exclusión mutua. Sin embargo, la inanición y el interbloqueo sí están presentes. Así, podemos tener a dos procesos bloqueados,

esperando cada uno una comunicación del otro, o bien esperando una comunicación de un proceso muerto. Como ejemplo de bloqueo indefinido, consideremos tres procesos, P1, P2, y P3, los cuales exhiben el siguiente comportamiento. P1 está intentando repetidamente comunicarse bien con P2 o con P3, y éstos están intentando comunicarse con P1. Podría darse una secuencia en la que P1 y P2 intercambiarian información repetidamente, mientras P3 está bloqueado esperando comunicarse con P1. No hay interbloqueo porque P1 permanece activo.

## 5.4 Exclusión mutua

Consideremos un sistema en el que existen procesos concurrentes donde cada uno tiene un segmento de código, denominado sección crítica. Para que estos procesos puedan ejecutarse de forma adecuada, hay que asegurar que cuando un proceso está ejecutando su sección crítica, ningún otro pueda ejecutar la suya. Es decir, la ejecución de las secciones críticas debe hacerse bajo exclusión mutua.

Para conseguir esto se debe diseñar un protocolo para hacer uso de los recursos críticos. Cualquier solución que se de para conseguir exclusión mutua debe cumplir los siguientes requisitos:

1. Forzar la exclusión mutua: sólo un proceso debe estar a un tiempo en su sección crítica, siempre que los procesos accedan desde su sección crítica a los mismos recursos.
2. Progresión: debe garantizar que cualquier proceso que se detiene o bloquea fuera de su sección crítica, no debe afectar a la capacidad de los que compiten por acceder al recurso crítico.
3. Espera limitada: no se debe postergar indefinidamente el acceso de un proceso a su sección crítica. Cuando ningún proceso está en su sección crítica, cualquier otro que solicite entrar en la suya deberá poder hacerlo sin dilación. Además, un proceso permanece en su sección crítica sólo por un tiempo finito.
4. No se pueden hacer suposiciones acerca de las velocidades relativas de los procesos ni de su número.

### 5.4.1 Soluciones software

Durante muchos años los investigadores que han tratado este tema han intentado, con más o menos fortuna, construir algoritmos que consigan la exclusión mutua.

La solución que se suele dar es insertar en el programa unas instrucciones adicionales, que van a ser ejecutadas cuando el proceso desee entrar o salir de su sección crítica. Esto es lo que se conoce como el **pre-protocolo** y el **post-protocolo**, respectivamente. La forma general que presenta una solución a la exclusión mutua es la que se muestra en el algoritmo 5.1.

**Algoritmo 5.1****Esquema para resolver la exclusión mutua**

```
bucle
    seccion_no_critica;
    pre_protocolo;
    seccion_critica;
    post_protocolo;
fin bucle
```

Cuando el programador realiza un algoritmo para solucionar la exclusión mutua, toda la responsabilidad de que esa solución sea correcta y que garantice todas las condiciones descritas anteriormente recae sobre él.

La creación de un algoritmo no es tan fácil, pues cuando la situación es de mediana complejidad y el número de procesos que interviene es alto, las soluciones son difíciles de implementar y de comprobar su bondad y corrección.

A continuación veremos los algoritmos de Dekker, Peterson y Lamport, utilizando para ello el lenguaje C, que será el empleado en la mayor parte de los programas que se muestran.

#### 5.4.1.1 Algoritmo de Dekker

El algoritmo de Dekker da solución a la exclusión mutua para dos procesos. Nosotros vamos a desarrollar el algoritmo en cuatro pasos que son incorrectos y que nos van a permitir ilustrar los principales fallos que se suelen cometer en el desarrollo de programas concurrentes.

##### Primer intento

El programa 5.2 muestra este primer intento. Se define una variable global **Turno** que puede tomar los valores 1 y 2, que va a indicar qué proceso puede entrar en su sección crítica. Inicialmente, **Turno** tiene un valor arbitrario, por ejemplo 1. Cuando un proceso desea entrar en su sección crítica ejecutará un pre-protocolo que consta de una sentencia que se repite hasta que el valor de **Turno** sea el adecuado. Al salir de la sección crítica, el proceso pone el valor de la variable **Turno** igual al del otro proceso.

**Programa 5.2**

Primer intento

```

#include "primero.h"
#define SIEMPRE 1

int Turno = 1;

void Proceso_1 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_1();
        while (Turno != 1)
            ;
        Seccion_critica_1();
        Turno = 2;
    }
}

void Proceso_2 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_2();
        while (Turno != 2)
            ;
        Seccion_critica_2();
        Turno = 1;
    }
}

```

Este algoritmo consigue la exclusión mutua pero presenta una serie de inconvenientes:

- Supongamos que **Proceso\_2** se para en la sección no crítica: en este caso el valor de **Turno** nunca volverá a cambiar de 2 a 1. Cuando **Proceso\_1** entre en su sección crítica y cambie el valor de **Turno** a 2, como **Proceso\_2** está parado nunca cambiará el valor de **Turno**, por lo que **Proceso\_1** no podrá volver a ejecutar su sección crítica.
- Este algoritmo obliga a los procesos a alternarse estrictamente en el uso de sus secciones críticas. Si **Proceso\_1** necesitara entrar en su sección crítica varias veces por segundo, mientras que **Proceso\_2** se contentara con entrar una vez por hora, esta solución no lo permitiría.
- Otro problema que presenta este algoritmo es el de la **espera activa**: mientras un proceso está en su sección crítica el otro agota su cuento de CPU en un bucle de espera.

## Segundo intento

Un segundo intento para resolver la exclusión mutua es el que aparece en el programa 5.3. En este algoritmo se dota a cada proceso de una variable, C1 y C2, que indican cuando un proceso está en su sección crítica. Inicialmente valen 1, permitiéndose así que cualquiera de los dos procesos entre en su sección crítica. Cuando uno de ellos, por ejemplo Proceso\_1, desea entrar, pone antes su variable a 0, impidiéndose así que el otro proceso entre. Cuando el proceso termina de ejecutar su sección crítica pone la variable a 1. Así, si un proceso se para en su sección no crítica, su variable valdrá 1, permitiendo que el otro pueda entrar en la suya.

Programa 5.3

Segundo intento

```
#include "segundo.h"
#define SIEMPRE 1

int C1 = 1;
int C2 = 1;

void Proceso_1 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_1();
        while (C2 != 1)
            ;
        C1 = 0;
        Seccion_critica_1();
        C1 = 1;
    }
}

void Proceso_2 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_2();
        while (C1 != 1)
            ;
        C2 = 0;
        Seccion_critica_2();
        C2 = 1;
    }
}
```

Sin embargo, este algoritmo no satisface el requisito de que ambos procesos no puedan estar a la vez dentro de su sección crítica. Veamos la prueba. Consideremos la siguiente secuencia de instrucciones de los dos procesos empezando en el estado inicial:

1. Proceso\_1 comprueba C2 y encuentra C2 = 1.
2. Proceso\_2 comprueba C1 y encuentra C1 = 1.
3. Proceso\_1 pone C1 a 0.
4. Proceso\_2 pone C2 a 0.
5. Proceso\_1 entra en su sección crítica.
6. Proceso\_2 entra en su sección crítica.

Se llega a una situación en la que ambos procesos consiguen entrar simultáneamente en sus secciones críticas.

### Tercer intento

En el segundo intento hemos introducido las variables Ci cuyo objeto era indicar cuándo está el Proceso\_i en su sección crítica. Sin embargo, una vez que un proceso ha salido del bucle de comprobación, ya no se puede evitar su entrada en la sección crítica. Por tanto, el estado de computación alcanzado después del bucle y antes de la asignación de Ci es parte de la sección crítica, pero todavía Ci no indica este hecho. El tercer intento reconoce que el bucle debería ser considerado parte de la sección crítica trasladando la asignación de Ci antes del bucle. Esto se puede ver en el programa 5.4.

Con este algoritmo se consigue la exclusión mutua, pero desafortunadamente puede conducir a un interbloqueo si se ejecuta una instrucción de cada proceso alternativamente:

1. Proceso\_1 asigna 0 a C1.
2. Proceso\_2 asigna 0 a C2.
3. Proceso\_1 comprueba C2 y permanece en el bucle.
4. Proceso\_2 comprueba C1 y permanece en el bucle.

Tenemos una situación donde un conjunto de procesos (dos, en este caso) desea entrar en su sección crítica, pero ninguno de ellos lo conseguirá.

**Programa 5.4**

Tercer intento

```
#include "tercero.h"
#define SIEMPRE 1

int C1 = 1;
int C2 = 1;

void Proceso_1 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_1();
        C1 = 0;
        while (C2 != 1)
            ;
        Seccion_critica_1();
        C1 = 1;
    }
}

void Proceso_2 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_2();
        C2 = 0;
        while (C1 != 1)
            ;
        Seccion_critica_2();
        C2 = 1;
    }
}
```

**Cuarto intento**

En el tercer intento, cuando **Proceso\_i** pone la variable  $C_i$  a 0, no sólo indica su intención de entrar en su sección crítica, sino que se reserva su derecho a hacerlo. Esto puede producir un interbloqueo si ambos procesos insisten simultáneamente en entrar en sus secciones críticas.

El cuarto intento intenta remediar el problema requiriéndole a un proceso que abandone su intención de entrar en su sección crítica si descubre que existe un estado de controversia con el otro. El programa 5.5 presenta esta nueva solución. La secuencia de asignaciones a la misma variable ( $C_1 = 1; C_1 = 0$ ) tiene sentido en una ejecución concurrente, cosa que no ocurre en una ejecución secuencial. Puesto que se permite una intercalación arbitraria de las instrucciones de los dos procesos, **Proceso\_2** puede ejecutar un número arbitrario de instrucciones entre las dos asignaciones a  $C_1$ . En este caso, cuando **Proceso\_1** renuncia al intento de entrar en su sección crítica poniendo  $C_1$  a 1, **Proceso\_2** puede ahora ejecutar el

bucle una vez más entrando así en la suya.

---

**Programa 5.5**

Cuarto intento

```
#include "cuarto.h"
#define SIEMPRE 1

int C1 = 1, C2 = 1;

void Proceso_1 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_1();
        C1 = 0;
        while (C2 != 1) {
            C1 = 1;
            C1 = 0;
        }
        Seccion_critica_1();
        C1 = 1;
    }
}

void Proceso_2 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_2();
        C2 = 0;
        while (C1 != 1) {
            C2 = 1;
            C2 = 0;
        }
        Seccion_critica_2();
        C2 = 1;
    }
}
```

---

Esta solución presenta dos defectos: es posible la inanición de un proceso, así como una forma de interbloqueo conocida como *livelock*. Cuando ocurre este tipo de interbloqueo puede haber ejecuciones satisfactorias, pero también es posible describir una o más secuencias de ejecución en las cuales ningún proceso entra en su sección crítica. Veamos cómo se pueden producir estos problemas:

**Inanición** Recordemos que es posible una intercalación arbitraria de instrucciones. Por tanto se podría dar la secuencia de instrucciones siguiente:

1. Proceso\_1 pone C1 a 0.
2. Proceso\_2 pone C2 a 0 .

3. Proceso\_2 comprueba C1 y entonces pone C2 a 1.
4. Proceso\_1 completa un bucle:
  - Comprueba C2.
  - Entra en su sección crítica.
  - Restablece C1.
  - Ejecuta su sección no crítica.
  - Pone C1 a 0.
5. Proceso\_2 pone C2 a 0.

Hemos vuelto al estado descrito en la línea 2. Los mismos pasos pueden repetirse indefinidamente, de forma que es posible describir una secuencia de ejecución en la que Proceso\_1 entra en su sección crítica siempre que quiere, mientras que Proceso\_2 permanece indefinidamente en su pre-protocolo.

**Livelock** Consideremos ahora una secuencia de ejecución en la cual se alternan perfectamente las instrucciones de los dos procesos:

1. Proceso\_1 pone C1 a 0.
2. Proceso\_2 pone C2 a 0.
3. Proceso\_1 comprueba C2 y permanece en el bucle.
4. Proceso\_2 comprueba C1 y permanece en el bucle.
5. Proceso\_1 restablece C1 a 1.
6. Proceso\_2 restablece C2 a 1.
7. Proceso\_1 pone C1 a 0.
8. Proceso\_2 pone C2 a 0.
9. Proceso\_1 comprueba C2 y permanece en el bucle.
10. Proceso\_2 comprueba C1 y permanece en el bucle.

Esta secuencia de ejecución puede continuar indefinidamente. Así tenemos la situación que define un interbloqueo: dos procesos que desean entrar en su sección crítica, pero ninguno lo consigue. Sin embargo, la más ligera desviación de la secuencia descrita permitirá a uno de los procesos entrar en su sección crítica. Por lo que clasificamos el problema como *livelock* en vez de interbloqueo.

Los dos problemas descritos anteriormente, inanición y *livelock*, pueden ser aceptables si la probabilidad de que ocurran es baja. Un diseñador de sistemas

debe sopesar la probabilidad y los posibles efectos de estos problemas frente a la complejidad de un algoritmo más sofisticado que no los sufra.

---

**Programa 5.6****Algoritmo de Dekker**

```
#include "dekker.h"
#define SIEMPRE 1

int C1 = 1, C2 = 1, Turno = 1;

void Proceso_1 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_1();
        C1 = 0;
        while (C2 != 1) {
            if (Turno == 2) {
                C1 = 1;
                while (Turno != 1)
                    ;
                C1 = 0;
            }
        }
        Seccion_critica_1();
        C1 = 1;
        Turno = 2;
    }
}

void Proceso_2 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_2();
        C2 = 0;
        while (C1 != 1) {
            if (Turno == 1) {
                C2 = 1;
                while (Turno != 2)
                    ;
                C2 = 0;
            }
        }
        Seccion_critica_2();
        C2 = 1;
        Turno = 1;
    }
}
```

---

## Solución correcta

El algoritmo de Dekker (programa 5.6) es como el cuarto intento expuesto, excepto en que el derecho a insistir es pasado explícitamente entre los procesos. Las variables individuales en cada proceso asegurarán la exclusión mutua, pero si se detecta alguna controversia, un proceso, por ejemplo **Proceso\_1**, consultará una variable adicional **Turno** para saber si realmente le corresponde a él entrar en su sección crítica. Si no es así, restablecerá **C1** y le cederá el paso a **Proceso\_2**, esperando a que cambie el valor de **Turno**.

Cuando **Proceso\_2** complete la ejecución de su sección crítica, cambiará el valor de **Turno** a 1, liberando a **Proceso\_1**. Incluso si **Proceso\_2** hace inmediatamente otra petición para entrar en la sección crítica, éste se verá bloqueado por el valor de **Turno**, una vez que **Proceso\_1** haya relanzado su petición. El algoritmo de Dekker es totalmente correcto.

### 5.4.1.2 Algoritmo de Peterson

El algoritmo de Dekker resuelve el problema de la exclusión mutua, pero es complejo de entender. En 1981, Peterson propuso un algoritmo elegante y más simple que podemos ver en el programa 5.7. Al igual que el de Dekker, resuelve el problema de exclusión mutua para dos procesos.

En este caso las variables **Ci** indican la posición de cada proceso con respecto a la exclusión mutua y la variable **Turno** resuelve los conflictos de simultaneidad.

Se puede demostrar fácilmente que cumple la exclusión mutua. Una vez que **Proceso\_1** pone **C1** a 0, **Proceso\_2** no puede entrar en su sección crítica. Si **Proceso\_2** está aún en la suya, **C2** vale 0, y **Proceso\_1** está bloqueado para entrar en la sección crítica. Por otro lado, se impide el interbloqueo. Supongamos que **Proceso\_1** está bloqueado en su bucle de espera. Esto significa que **C2** vale 0 y **Turno** vale 1. **Proceso\_1** puede entrar en su sección cuando **C2** valga 1 o **Turno** pase a valer 2. Consideremos los siguientes casos:

1. **Proceso\_2** no está interesado en entrar en la sección crítica. Este caso es imposible, porque implica que **C2** vale 1.
2. **Proceso\_2** está esperando para entrar en su sección crítica. Este caso también es imposible porque si **Turno** vale 2 entonces **Proceso\_1** puede entrar.
3. **Proceso\_2** entra en su sección crítica varias veces y monopoliza el acceso a ella. No puede pasar porque **Proceso\_2** está obligado a dar a **Proceso\_1** una oportunidad poniendo **Turno** a 2 antes de entrar en su sección crítica.

## Programa 5.7

## Algoritmo de Peterson

```

#include "peterson.h"
#define SIEMPRE 1

int C1 = 1, C2 = 1, Turno = 1;

void Proceso_1 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_1();
        C1 = 0;
        Turno = 1;
        while ((C2 == 0) && (Turno == 1))
            ;
        Seccion_critica_1();
        C1 = 1;
    }
}

void Proceso_2 (void) {
    while (SIEMPRE) {
        Seccion_no_critica_2();
        C2 = 0;
        Turno = 2;
        while ((C1 == 0) && (Turno == 2))
            ;
        Seccion_critica_2();
        C2 = 1;
    }
}

```

#### 5.4.1.3 Algoritmo de Lamport

Dekker y Peterson proporcionan la solución para la exclusión mutua de dos procesos. En 1974, Lamport propone un algoritmo para resolver la exclusión mutua para  $N$  procesos que se conoce con el nombre de **algoritmo de la panadería**.

En este caso se utilizan dos vectores de  $N$  posiciones que pueden tomar valores enteros. Cuando un proceso desea entrar en su sección crítica toma un ticket numerado, con un número mayor que todos los anteriores, como se hace en una panadería, carnicería, etc. El proceso esperará a entrar en la sección crítica hasta que su número sea el más bajo.

En este caso no se necesita una variable por cada dos procesos como `Turno` en el algoritmo de Dekker. Se emplea un vector `Elegir` que se usa como bandera, para indicar a los otros procesos que un proceso está eligiendo un ticket. Este vector

toma valores 0 ó 1. Por otro lado, tenemos el vector `Numero` que es el que posee el número del ticket que le corresponde al proceso.

Cuando existen dos procesos con el mismo número de ticket, se le da preferencia al proceso con identificador más bajo. El algoritmo de la panadería es el que aparece en el programa 5.8.

---

**Programa 5.8**
**Algoritmo de Lamport**

```
#include "lamport.h"
#define SIEMPRE 1
#define NUM_PROCESOS 20

/* vectores inicializados a 0 */
extern int Elegir[NUM_PROCESOS];
extern int Numero[NUM_PROCESOS];

void Proceso_i (int i) {
    while (SIEMPRE) {
        Seccion_no_critica_i();
        Elegir[i] = 1;
        Numero[i] = 1 + maximo(Numero);
        Elegir[i] = 0;
        for (j = 0; j < NUM_PROCESOS; j++) {
            if (j != i) {
                while (Elegir[j] != 0)
                    ;
                while ((Numero[j] != 0) && ((Numero[j] < Numero[i])
                    || ((Numero[i] == Numero[j]) && (j < i))))
                    ;
            }
        }
        Seccion_critica_i();
        Numero[i] = 0;
    }
}
```

---

#### 5.4.2 Soluciones con ayuda del hardware

Hasta ahora hemos visto una serie de algoritmos para resolver la exclusión mutua. El empleo de éstos es conflictivo, en el sentido de que el programador es el que va a ser el responsable de prevenir la exclusión mutua. Otra posibilidad es contar con la asistencia del hardware a la hora de conseguirla.

Una forma de conseguir exclusión mutua sería la deshabilitación de las interrupciones cuando un proceso estuviese accediendo a un recurso crítico, de esta

modo nos aseguraríamos de que ningún otro proceso va a obtener el control de la CPU y va a acceder a ese mismo recurso. Es decir, se deshabilitarían las interrupciones antes de entrar en la sección crítica, para habilitarlas de nuevo después de salir de ella. Esto se puede ver en el programa 5.9.

Programa 5.9	Deshabilitación de interrupciones
<pre>#include "interrupciones.h" #define SIEMPRE 1  int main() {     while (SIEMPRE) {         deshabilitar_interrupciones();         seccion_critica();         habilitar_interrupciones();         seccion_no_critica();     } }</pre>	

Sin embargo, esta solución no es adecuada ya que presenta numerosos inconvenientes. Por un lado, se degrada la eficiencia del sistema ya que pierde su capacidad de intercalamiento; además, esta solución sólo sería válida en sistemas con un solo procesador.

Muchas máquinas proporcionan instrucciones hardware especiales que hacen más fácil la implementación de algoritmos para conseguir exclusión mutua.

Se han propuesto varias instrucciones máquina que llevan a cabo dos acciones de forma atómica, por ejemplo, comprobar y modificar el contenido de una palabra, o intercambiar el contenido de dos palabras. Dado que estas acciones se realizan en un solo ciclo de instrucción, no están sujetas a la interferencia de otras. Estudiaremos las dos instrucciones que se suelen implementar más a menudo.

#### 5.4.2.1 Instrucción comprobar\_y\_establecer

Se trata de una instrucción hardware que lee una variable, almacena su valor en algún lugar, y le da un nuevo valor. Esta instrucción, llamada a menudo `comprobar_y_establecer(L)`, se puede definir de diversas formas. Una de ellas es la que aparece en el programa 5.10, que se caracteriza por ejecutarse de forma atómica, es decir, sin que se produzca ninguna interrupción entre las sentencias que la componen.

**Programa 5.10****Instrucción comprobar\_y\_establecer**

```
int comprobar_y_establecer (int *var) {
    int resultado;

    resultado = *var;
    *var = 1;
    return resultado;
}
```

Con esta instrucción, el problema de la exclusión mutua para  $N$  procesos se resuelve fácilmente, como se observa en el programa 5.11. En él existe una variable global C con un valor inicial de 0. Cuando un proceso desea acceder a la sección crítica ejecuta la instrucción hardware, de forma que el primer proceso que la ejecute podrá entrar, dejando la variable C con un valor 1, imposibilitando la entrada de otros procesos. Cuando el proceso salga de la sección crítica, restablecerá la variable C a 0, permitiendo la entrada de otro proceso a la sección crítica.

**Programa 5.11****Exclusión mutua con comprobar\_y\_establecer**

```
#include "comprobar.h"
#define SIEMPRE 1

extern int C; /* inicialmente con valor 0 */

void Proceso_i (void) {
    while (SIEMPRE) {
        Seccion_no_critica_i();
        while (comprobar_y_establecer(&C))
            ;
        Seccion_critica_i();
        C = 0;
    }
}
```

**5.4.2.2 Instrucción intercambiar**

Otra instrucción de este tipo es intercambiar(A,B), que es equivalente a la ejecución sin interrupción del programa 5.12.

**Programa 5.12****Instrucción intercambiar**


---

```
void intercambiar (int *a, int *b) {
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```

---

La exclusión mutua se resuelve tal como se muestra en el programa 5.13, donde C es una variable global inicializada a 0. Además, cada proceso posee una variable local Li, inicializada a 1. Cuando un proceso quiere entrar en la sección crítica comprueba si su variable Li vale 0. Si toma este valor puede entrar, en caso contrario ejecutaría la instrucción hardware. Cuando sale de la sección crítica vuelve a ejecutar la instrucción **intercambiar** para restablecer la variable C a 0, lo que permitirá la entrada de otro proceso.

**Programa 5.13****Exclusión mutua con intercambiar**


---

```
#include "intercambiar.h"
#define SIEMPRE 1

extern int C; /* inicialmente con valor 0 */

void Proceso_i (void) {
    int Li = 1;

    while (SIEMPRE) {
        Seccion_no_critica_i();
        while (Li != 0) {
            intercambiar(&C, &Li);
        }
        Seccion_critica_i();
        intercambiar(&C, &Li);
    }
}
```

---

**5.4.2.3 Propiedades de las instrucciones hardware**

El uso de estas instrucciones hardware para forzar la exclusión mutua presenta una serie de ventajas:

- Es aplicable a cualquier número de procesos, tanto en sistemas monoproce-

sadores como en multiprocesadores con memoria compartida.

- Es un método simple y, por tanto, fácil de verificar.
- Puede usarse para soportar múltiples secciones críticas; cada una estará definida por su propia variable.

Pero también presenta una serie de inconvenientes:

- Emplea la espera activa, es decir, mientras un proceso está esperando para acceder a su sección crítica está consumiendo tiempo de CPU.
- Es posible que se produzca inanición. Cuando un proceso deja su sección crítica y hay más de un proceso esperando, la selección se realiza de un modo arbitrario, por lo que algún proceso puede quedarse indefinidamente esperando entrar en su sección crítica.

Debido a los inconvenientes que presentan tanto las soluciones software como las hardware es necesario recurrir a otros mecanismos para conseguir exclusión mutua.

## 5.5 Semáforos

El primer avance importante en el tratamiento de los problemas que plantean los procesos concurrentes se produce en 1965 con la aparición de un trabajo de Dijkstra [Dijk65]. Dijkstra estaba interesado en el diseño de un sistema operativo como un conjunto de procesos secuenciales que cooperan y en el desarrollo de mecanismos fiables y eficientes para conseguir esta cooperación. La solución propuesta fueron los **semáforos**.

Los semáforos son una herramienta de sincronización de procesos proporcionada por el sistema operativo y, por tanto, puede ser utilizada para conseguir exclusión mutua.

El principio fundamental en el que se basan es el siguiente: *dos o más procesos pueden cooperar por medio de señales simples, de tal forma que un proceso puede ser forzado a parar en un sitio específico hasta que reciba una determinada señal*. Para conseguir esto se utilizan unas variables especiales llamadas semáforos. Para transmitir una señal a través de un semáforo S, un proceso ejecuta la primitiva `signal(S)`. Para recibirla ejecuta la primitiva `wait(S)`; si la señal correspondiente no

ha sido transmitida todavía, el proceso se queda a la espera de que la transmisión tenga lugar.

Más formalmente, un semáforo  $S$  es una variable entera que sólo puede tomar valores no negativos. Sobre él se definen dos operaciones que son las únicas que pueden modificar su contenido, aparte de la inicialización: `wait` y `signal`. Estas operaciones son atómicas, es decir, se ejecutan de forma indivisible. Por lo que si un proceso está modificando el valor de un semáforo, ningún otro puede modificar simultáneamente el valor de éste. Además, en el caso de `wait`, la comprobación del valor del semáforo y su posible modificación también tienen que ser ejecutadas sin interrupción.

La definición clásica de Dijkstra se puede ver en el programa 5.14. Esta implementación tiene el inconveniente de que presenta espera activa. Cuando un proceso ejecuta la operación `wait` y el semáforo tiene valor 0 se queda ejecutando un bucle. Esto es un problema en un sistema de multiprogramación con un solo procesador, ya que se gastan ciclos de CPU que podrían ser utilizados por otros procesos. Este tipo de semáforo también se llama *spinlock* y es útil en los sistemas con múltiples procesadores ya que no hay que hacer un cambio de proceso. Esto es especialmente útil cuando el tiempo de bloqueo va a ser corto.

---

#### Programa 5.14

#### Semáforo de Dijkstra

```
#include "semáforo.h"
#define semáforo_t int

void wait (semáforo_t S) {
    while (S <= 0)
        ;
    S--;
}

void signal (semáforo_t S) {
    S++;
}
```

---

Con objeto de evitar esa espera activa se han propuesto diversas soluciones. Una de ellas consiste en modificar la definición de las operaciones `wait` y `signal`:

- wait(S)** Si  $S > 0$  entonces  $S = S - 1$ , en caso contrario se bloquea el proceso. Se dice entonces que el proceso está bloqueado en el semáforo  $S$ .
- signal(S)** Si hay procesos bloqueados en este semáforo, se despierta a uno de ellos; en caso contrario,  $S = S + 1$ .

Para la implementación de esas operaciones, el semáforo S se convierte en una estructura que posee un valor entero no negativo y una lista de procesos. De este modo, cuando un proceso debe esperar en el semáforo, se añade a la lista, y pasa al estado de bloqueado. Cuando un proceso ejecuta una operación **signal** sobre el semáforo S, despierta a uno de los bloqueados en la cola del semáforo S. Si no hay procesos bloqueados, aumenta el valor de S. El programa 5.15 muestra esta nueva implementación.

**Programa 5.15****Semáforo**

```
#include <semaforo.h>

typedef struct {
    int contador;
    cola_t cola;
} semaforo_t;

void wait (semaforo_t S) {
    if (S.contador > 0)
        S.contador--;
    else {
        bloquear_proceso();
        insertar_proceso_cola(S.cola);
    }
}

void signal (semaforo_t S) {
    proceso_t P;

    if (hay_procesos_bloqueados(S.cola)) {
        P = quitar_proceso_cola(S.cola);
        desbloquear_proceso(P);
    }
    else
        S.contador++;
}
```

Hemos visto los llamados **semáforos generales o contadores**, que se caracterizan por tomar un valor no negativo. Existe un tipo de semáforo denominado **semáforo binario** que sólo toma los valores 0 ó 1. En este caso, la operación **signal(S)** se define así: si hay procesos bloqueados en este semáforo, se despierta a uno de ellos; en caso contrario, **S = 1**.

Un sistema operativo puede proporcionar un único tipo de semáforo, ya que se puede realizar una implementación de un semáforo binario con uno general y viceversa. El programa 5.16 muestra la forma de implementar un semáforo

contador mediante semáforos binarios. Para ello se emplean dos semáforos binarios inicializados con valores 1 y 0, respectivamente, así como una variable C que corresponde al valor del semáforo contador que se quiere implementar. El semáforo S1 sirve para asegurar la exclusión mutua en el acceso a la variable C, mientras que el semáforo S2 es el empleado para bloquear a los procesos.

---

**Programa 5.16**
**Semáforos contadores con semáforos binarios**

```
#include "semáforo.h"

/* inicializados con valor 1 y 0, respectivamente */
semáforo_t S1, S2;
int C;

void wait(semáforo_t S) {
    wait(S1);
    C--;
    if (C < 0) {
        signal(S1);
        wait(S2);
    }
    signal(S1);
}

void signal(semáforo_t S) {
    wait(S1);
    C++;
    if (C <= 0)
        signal(S2);
    else
        signal(S1);
}
```

---

El programa 5.17 muestra la implementación de un semáforo binario mediante semáforos contadores. Es muy similar a la anterior, utilizando también dos semáforos inicializados con valor 1 y 0, respectivamente. La diferencia estriba en la operación `signal` que se asegura de que el valor del semáforo no sea superior a 1.

### 5.5.1 Exclusión mutua

En el programa 5.18 se muestra como se puede conseguir exclusión mutua para dos procesos utilizando semáforos. Un proceso que desea entrar en su sección crítica, por ejemplo `Proceso_1`, ejecuta un pre-protocolo que consta sólo de la instrucción `wait(S)`. Si `S = 1` entonces `S` puede ser disminuido y `Proceso_1` puede

entrar en su sección crítica. Cuando Proceso\_1 sale de ella, ejecuta el post-protocolo consistente sólo en la instrucción `signal(S)`; el valor de `S` se incrementará en 1. Sin embargo, si Proceso\_2 intenta entrar en su sección crítica antes de que Proceso\_1 haya salido, `S = 0` y Proceso\_2 quedará bloqueado en `S`. Cuando finalmente Proceso\_1 sale, la operación `signal(S)` despertará a Proceso\_2.

La generalización para  $N$  procesos es muy simple. Un proceso, antes de hacer uso de la sección crítica, realiza un `wait(S)` para saber si está libre el recurso. Cuando deja de utilizarlo realiza un `signal(S)` para dejar paso a otro proceso.

Programa 5.17

Semáforos binarios con semáforos contadores

```
#include "semáforo.h"

/* inicializados con valor 1 y 0, respectivamente */
semáforo_t S1, S2;
int C;

void wait(semáforo_t S) {
    wait(S1);
    C--;
    if (C < 0) {
        signal(S1);
        wait(S2);
    }
    signal(S1);
}

void signal(semáforo_t S) {
    wait(S1);
    C++;
    if (C <= 0)
        signal(S2);
    else {
        C = 1;
        signal(S1);
    }
}
```

## Programa 5.18

## Exclusión mutua con semáforos

```

#include <semaforo.h>
#define SIEMPRE 1

extern semaforo_t S; /* valor inicial de 1 */

void Proceso_1 (void) {
    while (SIEMPRE){
        Seccion_no_critica_1();
        wait(S);
        Seccion_critica_1();
        signal(S);
    }
}

void Proceso_2 (void) {
    while (SIEMPRE){
        Seccion_no_critica_2();
        wait(S);
        Seccion_critica_2();
        signal(S);
    }
}

```

### 5.5.2 El problema del productor/consumidor

Examinaremos ahora uno de los problemas más comunes con el que nos podemos encontrar en el procesamiento concurrente: el productor/consumidor. Tenemos uno o varios procesos productores generando algún tipo de datos y colocándolos en un *buffer*. Hay uno o varios procesos consumidores que sacan elementos del *buffer* uno a uno. El sistema tiene que restringir las operaciones sobre el *buffer* para evitar que dos procesos accedan a él simultáneamente. Es decir, sólo un agente (productor o consumidor) puede acceder al *buffer* en un instante dado. El *buffer* tiene una longitud finita  $N$ .

En la solución a este problema, que puede verse en el programa 5.19, se utilizan 3 semáforos. El semáforo **BUFFER** controla el acceso al *buffer*, es decir, controla que un consumidor o un productor no acceda a él mientras está accediendo otro proceso. El semáforo **ELEMENTO** controla que el *buffer* no esté vacío; si está vacío, el consumidor no podrá acceder a él. El semáforo **HUECO** lleva el control del número de espacios vacíos; si está lleno el productor no podrá acceder a él.

El proceso productor antes de introducir un elemento en el *buffer* debe esperar

a que haya hueco (`wait(HUECO)`), si lo hay deberá comprobar si el *buffer* está siendo utilizado por otro proceso (`wait(BUFFER)`), si está libre, podrá introducir el elemento. A continuación deberá liberar el *buffer* (`signal(BUFFER)`) y avisar de que hay un nuevo elemento en él (`signal(ELEMENTO)`).

**Programa 5.19****Productor/Consumidor con semáforos**

```
#include <semaforo.h>

#define SIEMPRE 1

extern semaforo_t BUFFER; /* valor inicial = 1 */
extern semaforo_t HUECO; /* valor inicial = capacidad buffer */
extern semaforo_t ELEMENTO; /* valor inicial = 0 */

void productor (void) {
    elemento_t elem;

    while (SIEMPRE) {
        elem = producir_elemento();
        wait(HUECO);
        wait(BUFFER);
        introducir_elemento(elem);
        signal(BUFFER);
        signal(ELEMENTO);
    }
}

void consumidor (void) {
    elemento_t elem;

    while (SIEMPRE) {
        wait(ELEMENTO);
        wait(BUFFER);
        elem = sacar_elemento();
        signal(BUFFER);
        signal(HUECO);
        consumir_elemento(elem);
    }
}
```

Del mismo modo, el proceso consumidor deberá esperar a que exista algún elemento en el *buffer* (`wait(ELEMENTO)`) antes de intentar sacarlo; si lo hay, comprobará si el *buffer* está libre (`wait(BUFFER)`), si es así podrá obtener el elemento. Una vez hecho esto liberará el *buffer* (`signal(BUFFER)`) y avisará de que hay un hueco nuevo (`signal(HUECO)`).

Hay que hacer notar la importancia del orden de las operaciones `wait` sobre los semáforos para que el funcionamiento sea correcto. Así, si intercambiamos el orden de las operaciones `wait` se puede producir un interbloqueo.

## 5.6 Monitores

Los semáforos proporcionan una herramienta útil y flexible para sincronizar procesos, sin embargo tienen el inconveniente de que puede ser difícil escribir programas grandes correctos usándolos, ya que el olvido de una operación `signal` puede conducir a un bloqueo de los procesos, siendo difícil detectar la causa del fallo.

El monitor es una construcción del lenguaje de programación que proporciona una funcionalidad equivalente a la de los semáforos pero son más fáciles de controlar. El concepto fue definido formalmente por primera vez por Hoare en 1974 ([Hoar74]). La construcción `monitor` ha sido implementada en varios lenguajes de programación, tales como Pascal concurrente, Pascal-plus, Modula-2, Modula-3 y Ada.

Un monitor es un módulo software que consta de:

- Un conjunto de variables que representan a los recursos que van a ser controlados.
- Un conjunto de procedimientos mediante los cuales se accede a esos recursos.
- Unas variables de condición para sincronizar los procesos que acceden al monitor.
- Una secuencia de inicialización.

La implementación del monitor debe garantizar las siguientes condiciones:

1. Sólo se debe acceder a las variables locales desde los procedimientos del monitor.
2. Un proceso entra en el monitor llamando a uno de sus procedimientos.
3. Los procedimientos del monitor son mutuamente exclusivos, es decir, sólo un proceso puede estar ejecutándose en el monitor en un instante dado; cualquier otro proceso que llame al monitor quedará bloqueado mientras espera a que esté disponible.

La última característica es la que proporciona exclusión mutua. Si los datos en un monitor representan recursos, entonces el monitor asegura la exclusión mutua en el acceso a esos recursos.

El monitor debe proporcionar además herramientas de sincronización. Supongamos que un proceso llama a un monitor y, mientras está en él, tiene que bloquearse hasta que se cumpla alguna condición. Se necesita algún tipo de herramienta que permita al proceso bloquearse y liberar al monitor para permitir que otro proceso pueda utilizarlo. Más tarde, cuando se haya cumplido la condición y el monitor esté libre de nuevo, el proceso se reanudará y podrá entrar en el monitor.

Para solucionar estos problemas de sincronización, el monitor proporciona las denominadas **variables de condición** que están contenidas en el monitor y sólo son accesibles desde dentro de él. Sobre estas variables operan dos funciones:

- cwait(C) Bloquea la ejecución del proceso en la condición C, insertándolo en una cola asociada a dicha condición.
- csignal(C) Despierta un proceso bloqueado anteriormente en esa condición. Si no hay procesos bloqueados, no hace nada. Si hay varios se elige uno de ellos (normalmente el primero, al tratarse de una cola FIFO).

La estructura general de un monitor se representa en la figura 5.2. Un proceso puede entrar en un monitor llamando a cualquiera de sus procedimientos, pero sólo puede haber un proceso en un monitor en un instante dado. Para garantizarlo, los procesos que intenten entrar en el monitor se bloquean en la cola de entrada al monitor hasta que esté disponible. Una vez que un proceso está en el monitor se puede bloquear temporalmente en la variable de condición x emitiendo la llamada cwait(x); en este caso, se coloca en la cola asociada a dicha condición, a la espera de que ésta cambie.

Si un proceso que se está ejecutando en el monitor detecta un cambio en la variable de condición x, emite una señal mediante csignal(x), que avisa a la correspondiente cola de condición de que ésta ha cambiado. Si hay procesos en esta cola se activa uno de ellos, por tanto, el proceso que emite la señal debe salir del monitor o debe bloquearse. Si se bloquea pasará a la cola denominada Urgente en la figura 5.2. Cuando el monitor quede disponible para ser utilizado por otro proceso, los que se encuentran en la cola Urgente tendrán prioridad sobre los que estén en la cola de entrada inicial, ya que los primeros ya han realizado algún trabajo con el monitor.

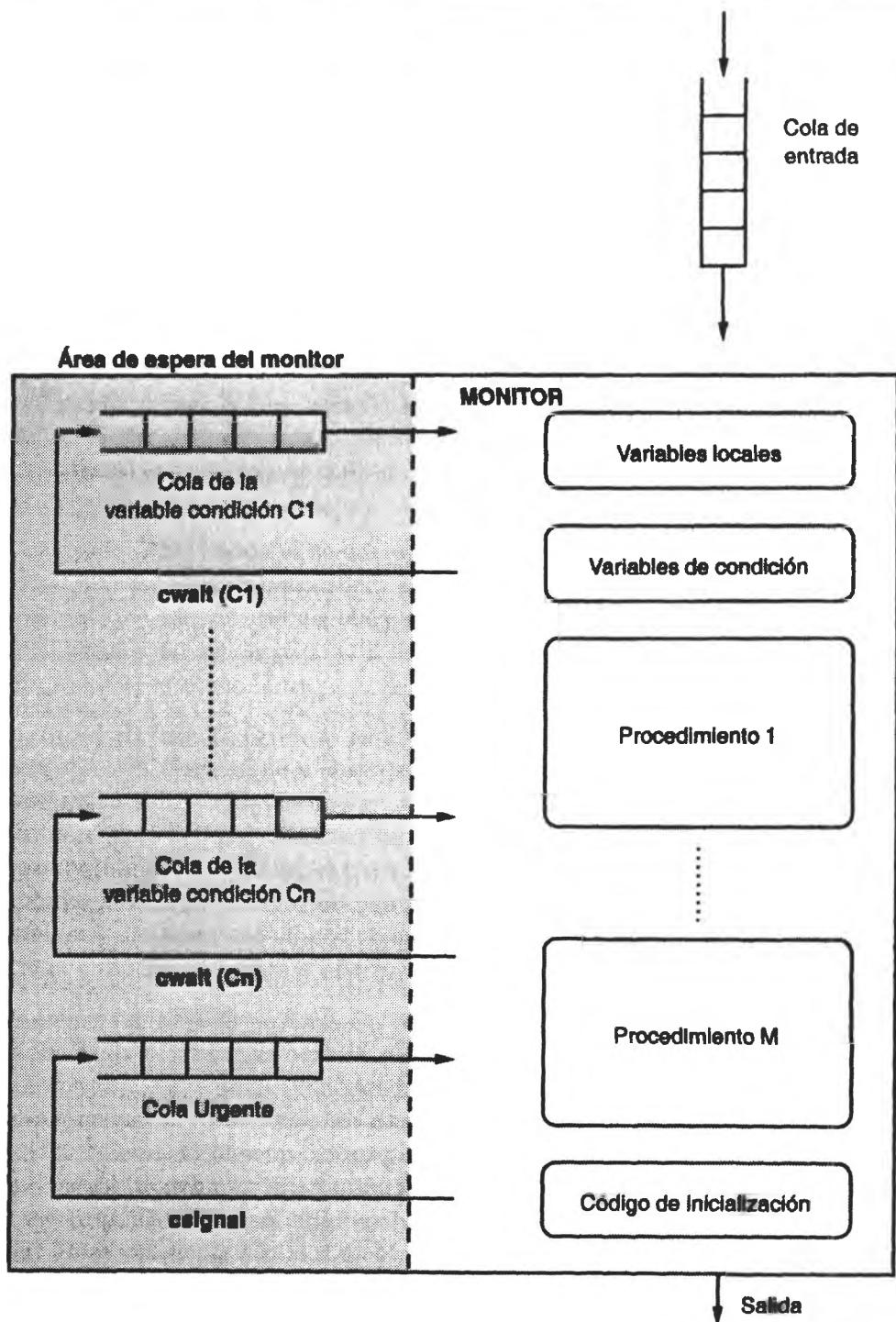


Figura 5.2: Estructura de un monitor

### 5.6.1 El problema del productor/consumidor

Como ejemplo de uso de los monitores, vamos a retomar el problema del productor/consumidor con *buffer* finito. El programa 5.20 muestra la solución utilizando un pseudocódigo, ya que el lenguaje C empleado hasta ahora no proporciona monitores.

El monitor **buffer\_limitado** controla el *buffer* que se utiliza para almacenar y retirar caracteres. Éste incluye dos variables de condición: **hueco** es cierta cuando hay sitio para añadir al menos un carácter al *buffer*, y **elemento**, que es cierta cuando hay al menos un carácter en él.

Un productor puede añadir caracteres al *buffer* sólo por medio del procedimiento **poner** del monitor y no tiene acceso directo al *buffer*. El procedimiento comprueba primero la condición **hueco** para determinar si hay espacio disponible en el *buffer*. Si no lo hay, el productor queda bloqueado en dicha condición. Cualquier otro proceso (productor o consumidor) puede entrar ahora en el monitor.

Más tarde, cuando el *buffer* deje de estar lleno, el productor podrá ser sacado de la cola, y continuará su procesamiento. Después de colocar un carácter en el *buffer*, el proceso señala la condición **elemento**. Se puede hacer una descripción similar del consumidor, siendo la condición **hueco** la utilizada para indicar la existencia de espacio libre en el *buffer*.

Este ejemplo permite ver la división de responsabilidad en los monitores en comparación con los semáforos. En el caso de los monitores, éstos garantizan la exclusión mutua. Sin embargo, el programador debe colocar las primitivas **cwait** y **csignal** adecuadas dentro de éste para prevenir que los procesos pongan datos en un *buffer* lleno o intenten sacarlos de un *buffer* vacío. En el caso de los semáforos, tanto la exclusión mutua como la sincronización son responsabilidad del programador.

Como en el caso de los semáforos, es posible cometer errores en la sincronización de los monitores. Por ejemplo, si se omite cualquiera de las funciones **csignal** en el monitor, los procesos que estén en la cola de la condición correspondiente se quedarán bloqueados indefinidamente.

La ventaja que tiene los monitores sobre los semáforos es que todas las funciones de sincronización están confinadas dentro del monitor. De este modo, es sencillo verificar si la sincronización se ha realizado correctamente y detectar los fallos. Es más, una vez que un monitor está correctamente programado, el acceso al recurso protegido es correcto para todos los procesos. Con los semáforos, en cambio, el acceso al recurso es correcto sólo si todos los procesos que acceden a él están correctamente programados.

**Programa 5.20****Productor/Consumidor con monitor**

```

program productor_consumidor
    monitor buffer_limitado;
        buffer: array [0..N] of caracteres;
        contador: entero;
        hueco, elemento: condition;
    procedure poner(x:caracter)
        begin
            si contador=N entonces cwait(hueco);
            poner_elemento(buffer, x);
            contador := contador + 1;
            csignal(elemento);
        end
    procedure tomar(x:caracter)
        begin
            si contador=0 entonces cwait(elemento);
            x := coger_elemento(buffer);
            contador := contador - 1;
            csignal(hueco);
        end
    begin
        contador := 0;
    end;
    procedure productor;
    var x: caracter;
    begin
        repetir
            producir(x);
            poner(x);
        siempre
    end;
    procedure consumidor;
    var x: caracter;
    begin
        repetir
            tomar(x);
            consumir(x);
        siempre
    end;
    begin
        parbegin
            productor; consumidor;
        parend
    end

```

## 5.7 Señales

Son una utilidad para superar o manejar las condiciones excepcionales de *software* o *hardware* que se den durante la ejecución de un programa. Son como interrupciones *software* que pueden ser enviadas a un proceso para informarle de algún evento asíncrono o situación especial.

Las señales son un mecanismo similar al semáforo, de forma que un proceso se queda bloqueado a la espera de una señal que será enviada por otro. Sin embargo, existen ciertas diferencias, y es que un proceso que espere una señal se quedará bloqueado hasta que ésta ocurra. Por otro lado, cuando un proceso envía una señal despierta al proceso que la esperase, y si no hubiera ninguno no hace nada, perdiéndose la señal. En una operación `signal` sobre un semáforo si no hay procesos esperando, el semáforo se incrementa.

## 5.8 Paso de mensajes

Los mecanismos anteriores nos permiten resolver la exclusión mutua, y de este modo sincronizar los procesos para compartir los recursos críticos. Otros, como las señales, simplemente nos permiten sincronizar procesos. Sin embargo, a veces es necesario intercambiar información entre procesos que se ejecutan concurrentemente. Desde la aparición de los primeros sistemas operativos como el monitor simple (ver capítulo 1), se incluyen mecanismos de protección de la memoria, de forma que un proceso sólo puede acceder a la zona de memoria que se le ha asignado. Por lo tanto, se necesita algún mecanismo, que nos permita copiar información del espacio de direcciones de un proceso a otro. Una herramienta que nos puede proporcionar una solución es el *paso de mensajes*.

Los mensajes son una colección de objetos con una estructura bien definida:

- Una cabecera de tamaño fijo con información sobre la fuente, el destino, tipo de mensaje, longitud e información de control.
- Un cuerpo de tamaño variable con el contenido del mensaje, donde su estructura e interpretación se definen por la aplicación.

El conjunto mínimo de primitivas necesarias para que los procesos puedan comunicarse y sincronizarse son:

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

donde *destino* indica el proceso al que se le quiere enviar el *mensaje*, y *origen* indica de quién queremos recibir el *mensaje*.

La semántica de estas primitivas necesita ser bien definida. Así, es necesario determinar si la comunicación es directa o indirecta, y si se realiza con bloqueo o sin bloqueo.

### 5.8.1 Características del direccionamiento

Un aspecto a considerar en las primitivas de comunicación es cómo establecer la entidad con la que se comunica, es decir, el *origen* y el *destino*. En este caso, podemos distinguir entre dos tipos de direccionamiento:

**Direccionamiento directo** en el que se especifica el nombre del proceso con el que se quiere comunicar.

**Direccionamiento indirecto** utiliza como medio de comunicación una estructura global de datos compartidas tanto por los emisores como por los receptores. Esta estructura puede ser un **buzón** o un **puerto**. De este modo, la comunicación se establece con esta estructura y no con el proceso. Esto es especialmente útil cuando el proceso emisor no tiene que conocer la identidad del proceso que recibe el mensaje. Igualmente, el proceso receptor sólo está interesado en el mensaje, y no de quién lo recibe. Por ejemplo, múltiples clientes pueden requerir servicios de uno o múltiples servidores.

El buzón es una estructura de datos global compartida tanto por el emisor como por el receptor, que requiere un acceso exclusivo y una sincronización de los procesos, lo que dificulta la tarea. Un buzón es una abstracción de una cola FIFO finita, mantenida por el núcleo, donde los mensajes se añaden o eliminan mediante las operaciones *send* y *receive*, respectivamente.

Un puerto es un caso especial de buzón, donde el propietario es el proceso que lo crea, y proporciona una comunicación muchos a uno, mientras que los buzones son objetos compartidos que permiten una comunicación muchos a muchos. En la figura 5.3 se observa las diferencias entre la comunicación directa e indirecta con buzones y puertos.

### 5.8.2 Sincronización de las primitivas

Otro aspecto a tener en cuenta en el paso de mensajes es el carácter **bloqueante** o **no bloqueante** de las primitivas de comunicación. La mayoría de los sistemas permiten la elección de ambos tipos.

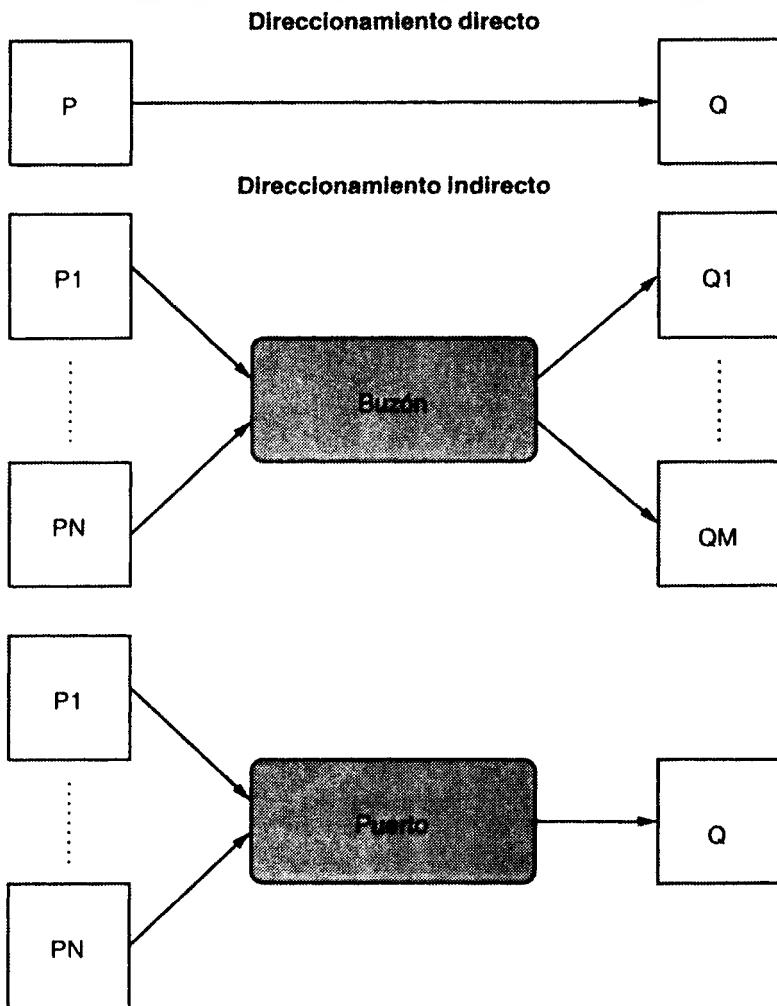


Figura 5.3: Comunicación directa e indirecta entre procesos

Una primitiva `send` bloqueante significa que el proceso emisor del mensaje se queda bloqueado hasta que éste llega al proceso receptor. Una primitiva `send` no bloqueante hace que el proceso emisor pueda continuar una vez que el mensaje ha sido entregado al núcleo. Por tanto, en este caso se necesita una estructura intermedia a la que enviar el mensaje, es decir, el direccionamiento debe ser indirecto.

Una primitiva `receive` bloqueante indica que el proceso receptor se queda bloqueado hasta recibir un mensaje. Un `receive` no bloqueante significa que el proceso

receptor comprueba la existencia de mensajes para él en una estructura intermedia y si no los hay continúa con su trabajo. En este caso, al igual que con el `send` bloqueante, se requiere un direccionamiento indirecto.

---

**Programa 5.21**
**Exclusión mutua con mensajes**

```
#include <mensaje.h>
#define SIEMPRE 1

extern buzon_t exclmut;

void inicial (void) {
    mensaje_t mens;
    crea_buzon(exclmut);
    send(exclmut, mens);
}

void Proceso_1 (void) {
    mensaje_t mens;
    while (SIEMPRE){
        Seccion_no_critica_1();
        receive(exclmut, mens);
        Seccion_critica_1();
        send(exclmut, mens);
    }
}

void Proceso_2 (void) {
    mensaje_t mens;
    while (SIEMPRE){
        Seccion_no_critica_2();
        receive(exclmut, mens);
        Seccion_critica_2();
        send(exclmut, mens);
    }
}
```

---

### 5.8.3 Exclusión mutua

En el programa 5.21 se muestra una posible solución al problema de la exclusión mutua para dos procesos utilizando mensajes con primitivas bloqueantes. El proceso que se ejecuta inicialmente se encargará de crear un buzón para la comunicación de los dos procesos, y enviará un mensaje. La presencia de éste en el buzón será indicador de que existen recursos disponibles. De este modo, cuando un proceso desea entrar en su sección crítica ejecuta un `receive`. Si existe el mensaje puede entrar en ella. Cuando sale de ésta, envía al buzón un nuevo

mensaje para liberar el recurso. Si el otro proceso intenta entrar en su sección crítica antes de que el primero haya salido de la suya, al no encontrar mensajes en el buzón, quedará bloqueado. Cuando finalmente libere el recurso, la operación `send` activará al otro proceso.

La generalización para  $N$  procesos es muy simple. Un proceso, antes de hacer uso de la sección crítica, realiza un `receive` para saber si está libre el recurso. Cuando deja de utilizar dicho recurso realiza un `send` para dejar paso a otro proceso.

## 5.9 Mecanismos de concurrencia en LINUX

LINUX proporciona diversos mecanismos para la sincronización y comunicación de procesos, como los semáforos, las señales, las interconexiones y las colas de mensajes.

### 5.9.1 Semáforos

Los semáforos son uno de los mecanismos que proporciona LINUX para la sincronización de procesos, con objeto de resolver problemas planteados en una relación de competencia y compartición de recursos.

En LINUX el semáforo no se trata individualmente, sino como un conjunto de semáforos, aunque éste puede constar de uno solo. Esta asociación en un conjunto permite identificar a todos los semáforos relacionados con el control de un recurso.

Las llamadas al sistema relacionadas con los semáforos son las que aparecen en la tabla 5.1. En ella podemos ver que sólo existe una primitiva para operar con ellos, `semop`. Cuando queremos hacer una operación tenemos que especificar el tipo de operación a realizar, indicándose el valor que será añadido o disminuido al contador actual del semáforo.

Llamada	Descripción
<code>semget</code>	Crea un conjunto de semáforos.
<code>semop</code>	Realiza una operación sobre un conjunto de semáforos.

Tabla 5.1: Llamadas al sistema relacionadas con semáforos

Además, es posible realizar un `wait` o un `signal` múltiple, es decir, podemos incrementar o decrementar el valor de un semáforo o un conjunto de ellos, en más de una unidad. En este caso, el sistema realizará esa operación múltiple como una

secuencia de operaciones individuales que incrementan o decrementan el valor de forma unitaria.

### 5.9.2 Señales

Otro mecanismo para sincronizar procesos concurrentes son las señales. La tabla 5.2 muestra las llamadas al sistema existentes en LINUX relacionadas con éstas. Los procesos pueden enviarse señales unos a otros a través de la llamada al sistema `kill`, siempre que tengan el mismo UID o GID, o procedan del mismo proceso padre. Es bastante frecuente que un proceso durante su ejecución reciba señales procedentes del núcleo.

Llamada	Descripción
<code>kill</code>	Envía una señal a un proceso.
<code>signal</code>	Instala un nuevo manejador para una señal.

**Tabla 5.2:** Llamadas al sistema relacionadas con señales

En LINUX las señales se pueden clasificar en los siguientes grupos:

- Relacionadas con la terminación de procesos.
- Relacionadas con las excepciones inducidas por los procesos. Por ejemplo, el intento de acceder fuera del espacio de direcciones de memoria del proceso, los errores producidos al manejar números en coma flotante, etc.
- Relacionadas con errores irrecuperables originados en el transcurso de una llamada al sistema.
- Originadas desde un proceso que se está ejecutando en modo usuario. Por ejemplo, cuando un proceso envía una señal a otro, cuando activa un temporizador y se queda a la espera de la señal de alarma, etc.
- Relacionadas con la interacción con el terminal. Por ejemplo, pulsar la tecla **break**.
- Señales para ejecutar un programa paso a paso. Son usadas por los depuradores.

Cuando un proceso recibe una señal puede hacer varias cosas:

- Realizar la acción por omisión asociada a la señal, es decir, se llama a la rutina de tratamiento predeterminada aportada por el núcleo.
- Realizar una acción definida por el propio proceso mediante la llamada `signal`; esto es posible salvo para las señales de parada y terminación.
- No realizar ninguna acción, excepto para las señales de parada y terminación en que debe ser realizada la acción por omisión.

### 5.9.3 Interconexiones en un sentido

La interconexión o tubería (*pipe*) es un método de comunicación entre procesos, que consiste en conectar la salida estándar de uno con la entrada estándar de otro, sin necesidad de crear por parte del usuario ficheros intermedios o temporales. Las tuberías son el método de comunicación entre procesos más antiguo en sistemas UNIX.

La comunicación entre procesos se realiza por medio de una zona de memoria compartida. Cuando creamos la interconexión, el núcleo reserva esta zona, asignándole dos descriptores, uno para escribir, y otro para leer. De este modo, la información que escribe un proceso en una interconexión puede ser leída inmediatamente por el otro, siendo el sistema el que realiza la sincronización de ambos. El único requisito para utilizar este mecanismo, es que los procesos deben ser hijos del mismo padre y residir en la misma máquina.

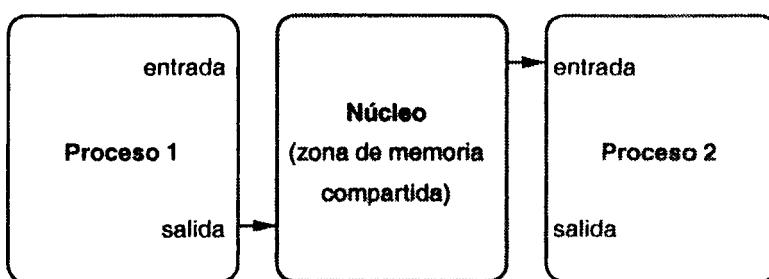


Figura 5.4: Representación gráfica de una interconexión

La llamada al sistema `pipe` (ver tabla 5.3) crea una interconexión entre dos procesos, de tal modo que nos devuelve dos descriptores, uno empleado para la lectura y otro para la escritura. Gráficamente podemos verlo en la figura 5.4, donde el núcleo actúa de intermediario. Cuando un proceso desea utilizar la

interconexión, emplea las mismas llamadas al sistema que para leer y escribir en un fichero (`read` y `write`, respectivamente), empleando como descriptores los obtenidos en la creación de la interconexión.

#### 5.9.4 Interconexiones FIFO

Una interconexión FIFO es similar a la anterior, diferenciándose en los siguientes aspectos:

- Existe como un fichero especial.
- Se pueden comunicar procesos de diferentes padres.
- Cuando se termina la comunicación entre los procesos, la interconexión permanece en el sistema para un uso posterior.

Esta interconexión se realiza con la llamada al sistema `mknod` (ver tabla 5.3). En este caso, para utilizar la interconexión, debido a que es un fichero especial que existe en el sistema, necesitamos abrirlo y cerrarlo como uno normal. Para ello utilizaremos las llamadas al sistema `open` y `close`, respectivamente. Una vez abierto el fichero, emplearemos las llamadas `read` y `write` para leer y escribir en él, respectivamente, al igual que en una interconexión en un sentido.

Llamada	Descripción
<code>pipe</code>	Crea una interconexión en un sentido.
<code>mknod</code>	Crea una interconexión FIFO.

**Tabla 5.3:** Llamadas al sistema relacionadas con interconexiones

Una de las características de esta comunicación es la necesidad de que existan dos procesos: uno para escribir y otro para leer. Si un proceso intenta leer de este fichero, quedará bloqueado hasta que no haya otro que quiera escribir en él, y viceversa.

#### 5.9.5 Colas de mensajes

Una **cola de mensajes** se puede describir como una lista enlazada dentro del espacio de direcciones del núcleo. Las llamadas al sistema relacionadas con ellas son las que aparecen en la tabla 5.4.

Llamada	Descripción
<code>msgget</code>	Crea una cola de mensajes.
<code>msgsnd</code>	Envía un mensaje a una cola.
<code>msgrcv</code>	Recibe un mensaje.

Tabla 5.4: Llamadas al sistema relacionadas con mensajes

Los mensajes se envían a la cola en orden, pero se pueden recuperar de distintas maneras. La más simple es mediante una estrategia FIFO. Sin embargo, es posible que un proceso recupere un mensaje determinado. Para ello, cada mensaje en la cola se identifica por una clave que conoce su receptor, que debe ser especificada cuando se intenta recuperar.

La comunicación mediante mensajes en el sistema LINUX se puede establecer bloqueante y no bloqueante, en función de los argumentos que se empleen cuando un proceso realice una llamada `msgsnd` o `msgrcv` para enviar o recibir un mensaje, respectivamente.

## 5.10 Resumen

La existencia de múltiples procesos concurrentes en un sistema ocasiona numerosos problemas. Uno de ellos es la exclusión mutua, es decir, el acceso exclusivo a un recurso crítico por parte de un proceso.

A lo largo de la historia de los sistemas operativos se han propuesto numerosas herramientas para solucionarlo. Los algoritmos como el de Dekker o el de Lamport permiten resolver la exclusión mutua, pero son de muy bajo nivel.

Algunos sistemas incorporan unas instrucciones hardware especiales para resolver este problema. Éstas presentan un mecanismo fácil de generalizar, pero con numerosos inconvenientes, tales como la espera activa y la inanición.

Los sistemas operativos aportan otro mecanismo, el semáforo. Éste es una estructura con dos componentes, una variable entera no negativa, y una cola para los procesos bloqueados en él. Sobre el semáforo se definen dos primitivas: `wait` y `signal`, que permiten decrementar e incrementar su valor, respectivamente. En función del valor de éste el proceso se bloqueará o despertará a otros procesos. Esto permite que el semáforo se convierta en una herramienta tanto para resolver el problema de la exclusión mutua, como los problemas de sincronización entre procesos.

Una herramienta de más alto nivel proporcionada por los lenguajes de programación son los monitores. En este caso, la exclusión mutua está resuelta intrínsecamente, liberando al programador de esta tarea, limitándose éste a la sincronización de los procesos dentro del monitor mediante las variables de condición.

El paso de mensajes, además de sincronizar procesos, nos permite comunicarlos. Los mensajes son enviados y recibidos a través de las primitivas `send` y `receive`, respectivamente, pudiendo ser enviados a un proceso o a una estructura intermedia de donde serán recogidos.

## 5.11 Ejercicios

1. Enumere los problemas que pueden aparecer entre procesos que compiten por el uso de los recursos y entre procesos que comparten recursos.
2. ¿Qué es una instrucción atómica?
3. ¿Qué es la espera activa? ¿Qué inconvenientes presenta?
4. Diferencias entre exclusión mutua, interbloqueo e inanición.
5. Se muestra un ejemplo de exclusión mutua mediante semáforos:

<b>proceso PROCESO-1</b>	<b>proceso PROCESO-2</b>
<b>    inicio</b>	<b>    inicio</b>
<b>repetir siempre</b>	<b>repetir siempre</b>
<b>wait(S)</b>	<b>wait(S)</b>
<b>seccion_critica</b>	<b>seccion_critica</b>
<b>signal(S)</b>	<b>signal(S)</b>
<b>seccion_no_critica</b>	<b>seccion_no_critica</b>
<b>fin-repetir</b>	<b>fin-repetir</b>
<b>fin</b>	<b>fin</b>

donde S es un semáforo con un valor inicial de 1. Explicar de forma razonada qué ocurriría si se producen los siguientes cambios:

- (a) El PROCESO-2 tras ejecutar la sección crítica realiza `wait(S)`, en lugar de `signal(S)`.
- (b) El semáforo se inicializa con un valor de 0.
- (c) El semáforo se inicializa con un valor de 2.
- (d) El PROCESO-1 realiza un `signal(S)` en lugar de un `wait(S)` antes de entrar en la sección crítica.
6. Indique si es posible que un semáforo tome los siguientes valores y en caso afirmativo diga qué ha ocurrido para que el semáforo tenga ese valor:

- (a)  $S = -10$   
(b)  $S = 10$   
(c)  $S = 0$   
(d)  $S = 3$  con un proceso bloqueado en él.  
(e)  $S = 0$  con tres procesos bloqueados en él.
7. Demuestre cómo puede producirse un interbloqueo si se cambia el orden de las operaciones `wait` en la solución a la exclusión mutua con semáforos vista en el programa 5.19.
8. Especifique qué método de resolución de exclusión mutua es más conveniente y por qué, en cada uno de los siguientes casos:
- (a) Sistema multiprocesador:
    - Inhabilitación de interrupciones.
    - Instrucciones especiales.
  - (b)  $N$  procesos y  $M$  recursos críticos:
    - Algoritmo.
    - Semáforos.
    - Monitores.
  - (c) 2 procesos y 1 recurso en una máquina desnuda (sin sistema operativo):
    - Algoritmo.
    - Semáforo binario.
  - (d) 2 procesos y 1 recurso en una máquina con un sistema operativo que implementa semáforos binarios:
    - Semáforos generales.
    - Algoritmo.
9. Dado el siguiente problema de exclusión mutua resuelto mediante semáforos:

```
program problema
const n=20;
var s:semáforo (:=1);
procedure P (i:entero)
inicio
    repetir siempre
        wait(S);
        sección_crítica
        signal(S);
        sección_no_crítica
    fin-repetir
```

```
fin
inicio /* programa principal */
parbegin
    P(1);
    P(2);
    ...
    P(20);
parend
fin
```

Resuélvalo mediante:

- (a) La instrucción comprobar\_y\_establecer.
  - (b) La instrucción intercambiar.
  - (c) Paso de mensajes.
  - (d) Monitores.
10. Establezca un mecanismo con semáforos para entrar en un aula de prácticas de una Universidad, donde se sabe que existen menos ordenadores que alumnos. Se ha de tener en cuenta que para poder entrar al aula los alumnos tienen que pasar por un pasillo, donde sólo cabe un alumno.
11. En un supermercado existen 20 cajeras y 1000 clientes. Los clientes cuando realizan compras se dirigen a una cajera libre. Una vez allí colocan sus productos en la cinta transportadora, donde la cajera irá realizando la suma de todos ellos. Una vez sumado todos los productos, se le comunicará el precio al cliente que pagará. Éste tendrá que esperar a que se le devuelva el ticket de compra. Puede ocurrir que la cajera se quede en ciertos momentos sin cambio suficiente por lo que tendrá que llamar al único encargado del supermercado. Resuelva este problema con semáforos, suponiendo que la cinta transportadora dispone de una capacidad para cinco productos.
12. ¿Qué diferencias hay entre los monitores y los semáforos a la hora de resolver los problemas de exclusión mutua y sincronización?
13. ¿Qué diferencia existe entre un buzón y un puerto?
14. ¿Qué características presentan los semáforos en LINUX?

# Capítulo 6

## Interbloqueos

---

La existencia de recursos que deben ser utilizados bajo exclusión mutua puede desembocar en una situación no deseada dentro del sistema, el interbloqueo. Éstos afectan al rendimiento, ya que dejan procesos bloqueados que tienen asignados recursos. En este capítulo veremos cuáles son las condiciones que deben darse para que se produzca este problema, así como las técnicas para abordarlo.

### 6.1 Introducción

Una de las funciones principales del sistema operativo es la gestión de los recursos del sistema, para lo cuál se encarga de asignar, desasignar y llevar el control de éstos. Cuando un proceso necesita un recurso, lo tiene que solicitar al sistema mediante los servicios que éste proporciona. Los recursos críticos pueden provocar la aparición de una serie de problemas en el sistema.

Cuando un proceso solicita un recurso pueden ocurrir dos cosas, una es que esté disponible, en este caso se le asignará y el proceso podrá continuar su ejecución; otra posibilidad es que el recurso esté asignado a otro proceso, por lo que el primero tendrá que pasar a estado bloqueado a la espera del recurso.

Puede darse una situación en la que los procesos que están bloqueados nunca cambien de estado, porque los recursos que han solicitado están asignados a

otros procesos que también están bloqueados. Esta situación se denomina interbloqueo<sup>1</sup>.

Podemos definir un interbloqueo como el bloqueo permanente de un conjunto de procesos que compiten por los recursos del sistema o que se comunican entre ellos. De esta forma, un conjunto de procesos se encuentra en estado de interbloqueo, cuando cada uno de ellos está esperando un suceso que sólo puede ser causado por otro proceso del mismo conjunto, y que nunca se producirá porque todos están bloqueados.

A diferencia de otros problemas que surgen entre los procesos concurrentes, no existe para éste una solución eficiente. En este capítulo, examinaremos la naturaleza del problema del interbloqueo, y la forma en que los sistemas operativos pueden tratarlo.

## 6.2 Condiciones necesarias

Coffman, Elphick y Shoshani establecen en 1971 [Coff71] que para que se produzca un interbloqueo deben darse simultáneamente las cuatro condiciones siguientes:

1. **Exclusión mutua**: Sólo un proceso puede usar un recurso a un tiempo.
2. **Retener y esperar**: Los procesos retienen los recursos ya asignados, mientras esperan la asignación de otros para continuar con su ejecución.
3. **No apropiación**: No se puede quitar a la fuerza los recursos asignados a un proceso, éste debe liberarlos voluntariamente cuando no los necesite.
4. **Espera circular**: Existe una cadena circular de procesos en la cuál cada proceso tiene uno o más recursos que son requeridos por el siguiente proceso en la cadena. Formalmente, hay un conjunto de procesos en espera  $\{p_0, \dots, p_n\}$ , tal que el proceso  $p_i$  está esperando un recurso retenido por  $p_{i+1}$ , para todo  $i = \{0, \dots, n\}$ ; y,  $p_n$  está esperando un recurso retenido por  $p_0$ .

Las tres primeras condiciones son necesarias, pero no suficientes, para que exista interbloqueo. La cuarta condición es en realidad, una consecuencia potencial de las tres primeras. Es decir, dado que se producen las tres primeras condiciones, puede ocurrir una secuencia de eventos que desembocue en una espera circular irresoluble. Esta espera circular es de hecho la definición de un interbloqueo. A

<sup>1</sup>También se conoce como abrazo mortal, bloqueo mutuo o *deadlock*.

pesar de ser la definición, para que no se pueda resolver la espera circular han de cumplirse las condiciones anteriores.

Existe una diferencia fundamental entre las tres primeras condiciones y la última. Las tres primeras son decisiones de política del sistema operativo, mientras que la cuarta es una circunstancia que podría darse o no en función de la secuencia de peticiones y liberaciones de recursos por parte de los procesos.

### 6.3 Modelado del interbloqueo

En 1972, Holt [Holt72] demostró cómo pueden ser modeladas las cuatro condiciones mediante los denominados **grafos de asignación de recursos**, en los que se indican las asignaciones y peticiones de recursos en un sistema.

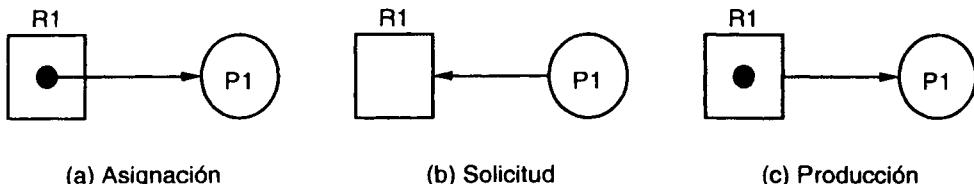
Estos grafos usan la siguiente notación gráfica:

- Los círculos representan procesos.
- Los cuadrados grandes representan clases o tipos de recursos críticos.
- Los círculos pequeños dentro de los anteriores indican las unidades que existen de cada tipo de recurso.
- Los arcos orientados representan:

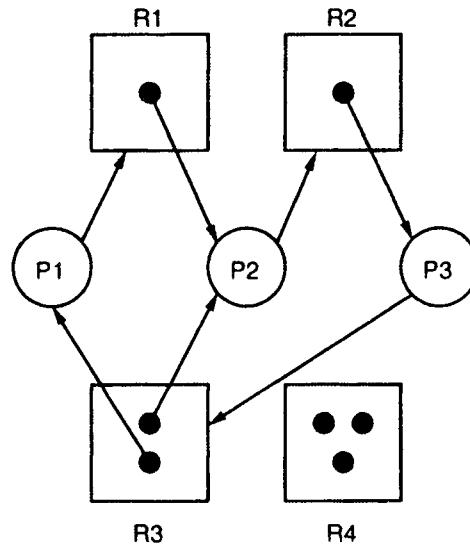
**Asignación** Si el arco va del vértice que representa una instancia de un tipo de recurso al que representa un proceso, indica que el recurso está asignado al proceso (figura 6.1(a)).

**Solicitud** Si el arco va de un vértice que representa un proceso a uno que representa un tipo de recurso, indica una petición del recurso (figura 6.1(b)).

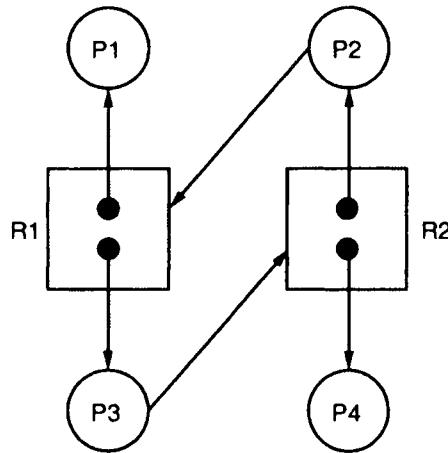
**Producción** Si el arco va de un vértice que representa un tipo de recurso a un proceso, indica que éste es productor del recurso (figura 6.1(c)).



**Figura 6.1:** Notación para el modelado de interbloqueos



**Figura 6.2:** Grafo de asignación de recursos con interbloqueo



**Figura 6.3:** Grafo de asignación de recursos con ciclo pero sin interbloqueo

Este grafo de asignación de recursos nos permite determinar de una forma fácil si en un sistema existe un interbloqueo o no. Así, si el grafo no tiene un ciclo, entonces el sistema no se encuentra en un estado de interbloqueo. Por otro lado, si está interbloqueado el grafo presenta al menos un ciclo, donde se encuentran implicados los procesos y recursos que lo forman, como se puede ver en la figura 6.2. Sin embargo, la presencia de un ciclo no implica la existencia de

un interbloqueo. Así, la figura 6.3 presenta un ciclo, pero los procesos P1 y P4 pueden finalizar y liberar sus recursos, de forma que P2 y P3 pueden continuar con su trabajo.

## 6.4 Estrategias para tratar los interbloqueos

Dado que las peticiones y liberaciones de recursos se realizan en un orden impredecible, es muy difícil diseñar una política infalible con un coste aceptable, al contrario que con la exclusión mutua.

Los sistemas operativos pueden afrontar los interbloqueos siguiendo una de las tres estrategias siguientes:

- Utilizar un protocolo que asegure que no van a aparecer interbloqueos en el sistema. Esto se puede conseguir de dos formas diferentes, mediante lo que se conoce como **prevención** de interbloqueos, que intenta evitar que éstos aparezcan eliminando una de las cuatro condiciones necesarias; o bien, mediante los métodos de **predicción**<sup>2</sup>, éstos hacen un reparto cuidadoso de los recursos, intentando predecir antes de asignar uno, si esta asignación puede conducir a una situación de interbloqueo.
- Dejar que aparezcan interbloqueos y utilizar métodos para detectarlos y, si existen, hacer que desaparezcan. Éstos son los denominados métodos de **detección y recuperación**.
- Otros sistemas no se preocupan del problema en absoluto. Si el interbloqueo aparece en el sistema habrá que hacerlo desaparecer manualmente. Esta estrategia es empleada por muchos sistemas operativos, incluyendo a **UNIX** y **LINUX**. Es aceptable en el caso de que los interbloqueos ocurran con muy poca frecuencia.

### 6.4.1 Prevención

Esta técnica fue propuesta por Havender en 1968 [Have68]. Consiste en diseñar un sistema de tal forma que se elimine toda posibilidad de que ocurra un interbloqueo. Para ello, hemos de asegurarnos de que una de las cuatro condiciones (véase el apartado 6.2) no se dé en el sistema. En la práctica sólo se proporcionan métodos

<sup>2</sup>La expresión inglesa *avoidance* que en la mayor parte de la bibliografía en español sobre interbloqueos se traduce como evitación, se ha traducido como predicción, ya que prevenir y evitar se pueden considerar sinónimos en español.

para eliminar las tres últimas, dado que un sistema operativo debe proporcionar exclusión mutua para el acceso a los recursos críticos.

Los métodos para prevenir el interbloqueo pueden ser de dos tipos. Los indirectos que consisten en impedir la aparición de la condición de retener y esperar o la de no apropiación, y los directos que evitan la aparición de la espera circular.

#### 6.4.1.1 Negación de la condición de retener y esperar

Para asegurarnos de que en un sistema nunca se cumple la condición de retener y esperar, tenemos que garantizar que siempre que un proceso solicita un recurso no retenga otros. Esto se puede conseguir mediante distintas estrategias.

##### **Todo o nada**

Consiste en forzar a los procesos a pedir todos los recursos que van a necesitar para su terminación al principio, y bloquearlos mientras esto no pueda ser garantizado. De esa forma, si está disponible el conjunto de recursos que necesita un proceso, entonces el sistema puede asignárselos y éste puede seguir su ejecución. Si algún recurso no está disponible, el proceso deberá esperar. Mientras el proceso espera no puede tener ningún recurso. Con esto se evita la condición de retener y esperar, y por tanto, no puede aparecer el interbloqueo.

La forma de implementarla es hacer que las peticiones de recursos se hagan al comienzo, independientemente de si los va a utilizar inmediatamente o no. Para ello se debe disponer de una llamada al sistema que permita solicitar un conjunto de recursos simultáneamente; asimismo, la liberación de éstos se hará en bloque al finalizar la ejecución del proceso. Esta solución es ineficiente debido a dos causas:

- Baja utilización de los recursos debida a que éstos pueden estar asignados, pero inactivos durante un largo período de tiempo, durante el cuál se niega el acceso a otros procesos. Por ejemplo, supongamos un proceso que tras leer unos datos de un fichero, realiza cálculos y por último, al final de su ejecución imprime los resultados. Mientras realiza las operaciones de cálculo tiene asignados el fichero y la impresora sin que ningún otro proceso pueda usarlos, puesto que no puede liberar ningún recurso hasta que no termine.
- Posibilidad de inanición de un proceso que requiera recursos muy utilizados, ya que puede esperar mucho tiempo para conseguirlos, aunque podría empezar a ejecutarse sólo con algunos de ellos. En el ejemplo anterior, el proceso

podría haber empezado su ejecución sólo con el fichero, sin necesidad de esperar la impresora.

## División de peticiones

Para disminuir los inconvenientes anteriores se puede dividir el proceso en fases que se ejecuten de manera relativamente independiente, y aplicar la estrategia anterior a cada una de ellas. El proceso pide todos los recursos que va a necesitar durante la ejecución de cada fase. Cuando termina una, libera todos los recursos asignados y solicita los de la siguiente. De esta forma, se reduce el desperdicio de recursos, pero hace más complicado el trabajo del programador, al requerir un trabajo extra en el diseño de las aplicaciones. En el caso descrito en el apartado anterior se podían definir tres fases, una de lectura de datos en la que sólo necesitará el fichero, otra de cálculo en la que no necesita ningún recurso y, por último, una de impresión de resultados en la que utilizará la impresora.

### Petición incremental de recursos y liberación de éstos

El proceso va pidiendo los recursos a medida que los necesita; si un recurso no se le puede conceder porque no está disponible, liberará voluntariamente todos los que tenía asignados hasta el momento. El proceso se ocupa de dejar los recursos en el estado adecuado.

El problema que se presenta con esta estrategia es que algunos recursos no pueden ser liberados y readquiridos posteriormente de una forma fácil. Por ejemplo, si se han realizado cambios irreversibles en la memoria o en ficheros, el simple hecho de devolver el recurso puede corromper el sistema.

Por tanto, la liberación de un recurso sólo tiene sentido si la integridad del sistema no se ve afectada, y cuando el gasto debido a las operaciones de guardar y restaurar los estados de los procesos y los recursos sea aceptablemente pequeño.

#### 6.4.1.2 Negación de la condición de no apropiación

La condición de no apropiación puede ser negada permitiendo la apropiación, es decir, autorizando al sistema a retirar el control de ciertos recursos a un proceso bloqueado. De este modo, si un proceso solicita un recurso que no está disponible porque está asignado a otro proceso, entonces el sistema puede apropiarlo si el proceso al que está asignado se encuentra bloqueado. El empleo de este tipo de estrategia podría seguir el algoritmo 6.1 cada vez que un proceso solicita un recurso.

**Algoritmo 6.1****Negación de la no apropiación**

```

si está disponible
entonces
    se asigna el recurso al proceso
si no si asignado a procesos en espera de recursos
entonces
    el sistema se apropiá del recurso
    el sistema asigna el recurso al proceso solicitante
si no
    el proceso pasa a estado de espera del recurso
fin si

```

Esta apropiación es involuntaria desde el punto de vista del proceso afectado, por lo que el sistema operativo debe encargarse de guardar el estado del recurso apropiado y añadir éste a la lista de los solicitados por el proceso. Éste volverá a ejecutarse solamente cuando pueda obtener de nuevo sus recursos anteriores, así como los nuevos que está solicitando.

La apropiación sólo es posible para ciertos tipos de recursos, que permitan guardar de forma fácil su estado, un ejemplo es la memoria principal cuyo contenido puede ser almacenado en memoria secundaria. Sin embargo, otro tipo de recursos, tales como ficheros parcialmente actualizados, no pueden ser apropiados sin corromper el sistema. Esto hace que la apropiación de recursos sea aún más difícil que la liberación voluntaria de la que se habló antes.

#### **6.4.1.3 Negación de la condición de espera circular**

Una forma de evitar la espera circular es mediante la siguiente estrategia:

1. Definir una ordenación de los tipos de recursos. Para ello, asignamos a cada tipo de recurso un número entero único, que nos permite compararlos y determinar si uno precede al otro en nuestro orden. Más formalmente, sea  $R = \{r_1, \dots, r_n\}$  el conjunto de tipos de recursos, en los que establecemos el siguiente orden:  $r_1 < r_2 < \dots < r_n$ .
2. Exigir que los procesos pidan los recursos en orden creciente de enumeración. Esto es posible mediante dos estrategias:
  - Si un proceso solicita un recurso de tipo  $r_i$ , después sólo podrá solicitar recursos de tipo  $r_j$ , si y sólo si  $r_j > r_i$ . Cuando se necesiten varios ejemplares del mismo tipo de recurso, sólo se debe hacer una petición para todos.

- Siempre que un proceso solicite un recurso de tipo  $r_i$  debe liberar todos los recursos  $r_k$  que tengan un índice mayor, es decir,  $r_k > r_i$ .

Esta estrategia presenta ciertas dificultades a la hora de implementarla:

- La ordenación de los tipos de recursos debe ser lógica para evitar esperas innecesarias, de tal modo, que siga el orden en el que se usan normalmente los recursos. Los procesos que los necesiten en orden diferente al previsto, deberán adquirirlos y conservarlos durante bastante tiempo antes de utilizarlos, lo cuál implica un bajo rendimiento o utilización de los recursos. Por tanto, hemos de buscar la ordenación óptima de éstos.
- Si se agregan nuevos tipos de recursos al sistema hemos de establecer una nueva ordenación. Además, en algunos casos, puede ser necesario modificar las diferentes aplicaciones para adaptarse al nuevo orden.
- La ordenación afecta a la capacidad del programador para escribir libre y fácilmente el código de sus aplicaciones, pues ha de solicitar los distintos recursos en función de ésta.

## 6.4.2 Predicción

Los métodos de prevención se basan en negar alguna de las condiciones necesarias para que se produzcan interbloqueos, restringiendo de cierta manera las peticiones de recursos. Esto conduce a un uso ineficiente de éstos.

Otro enfoque para resolver el problema de los interbloqueos es la predicción. Este método permite que se den las tres primeras condiciones, pero cuando un proceso pide un recurso el sistema estudia si su asignación puede producir un interbloqueo o no.

Para aplicar una estrategia de este tipo, se necesita disponer de información sobre las peticiones de recursos que van a realizar todos los procesos del sistema.

### 6.4.2.1 El algoritmo del banquero

Existen diversos algoritmos de predicción caracterizados por la mayor o menor cantidad de información previa necesaria. El más conocido es el **algoritmo del banquero**, propuesto por Dijkstra en 1965 [Dijk65], llamado así porque introduce a un banquero que hace préstamos y recibe pagos de una fuente dada de capital. Éste necesita conocer el número de recursos disponibles y asignados, así como

la demanda máxima de los procesos. Antes de ver este algoritmo, empezaremos definiendo una serie de conceptos relacionados con él.

Consideremos un sistema de  $n$  procesos y  $m$  tipos de recursos diferentes. Definiremos los siguientes vectores y matrices:

$$\text{Recursos} = [R_i] = (R_1, R_2, \dots, R_m)$$

$$\text{Disponible} = [D_i] = (D_1, D_2, \dots, D_m)$$

$$\text{Demanda} = [Dm_{ij}] = \begin{pmatrix} Dm_{11} & Dm_{12} & \dots & Dm_{1m} \\ Dm_{21} & Dm_{22} & \dots & Dm_{2m} \\ \dots & \dots & \dots & \dots \\ Dm_{n1} & Dm_{n2} & \dots & Dm_{nm} \end{pmatrix}$$

$$\text{Asignación} = [A_{ij}] = \begin{pmatrix} A_{11} & A_{12} & \dots & A_{1m} \\ A_{21} & A_{22} & \dots & A_{2m} \\ \dots & \dots & \dots & \dots \\ A_{n1} & A_{n2} & \dots & A_{nm} \end{pmatrix}$$

El vector *Recursos* señala la cantidad total de cada tipo de recurso que hay en el sistema. El vector *Disponible* da la cantidad de cada recurso no asignada actualmente. La matriz *Demanda* indica las necesidades máximas de cada proceso para cada recurso. Es decir,  $Dm_{ij}$  da la demanda del proceso  $p_i$  para el recurso  $r_j$ . Esta información debe ser declarada por el proceso de antemano para que se pueda utilizar el algoritmo del banquero. Del mismo modo,  $A_{ij}$  es el número de unidades del recurso  $r_j$  asignadas al proceso  $p_i$ . Con estas estructuras se pueden establecer las siguientes relaciones:

1. Todos los recursos o están asignados o están disponibles.

$$\forall i, \sum_{k=1}^n A_{ki} + D_i = R_i$$

2. Un proceso no puede solicitar más unidades de cada tipo de recurso de las disponibles en el sistema.

$$\forall k, i, Dm_{ki} \leq R_i$$

3. A los procesos no se les puede asignar mayor cantidad de recursos que los declarados inicialmente.

$$\forall k, i, A_{ki} \leq Dm_{ki}$$

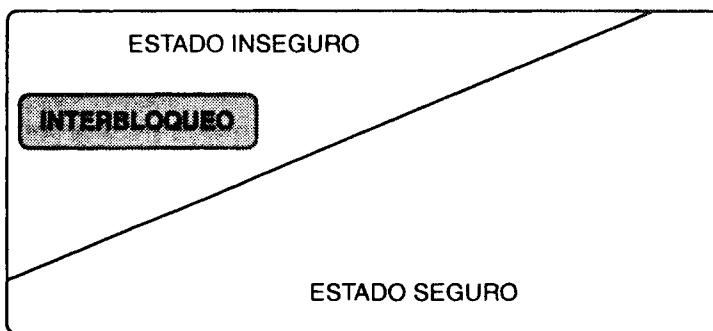
Las estructuras definidas anteriormente nos permiten definir el **estado del sistema**, donde se refleja qué recursos tiene asignados cada proceso. A partir de éste podemos obtener la matriz *Necesidad* que representará el número de instancias de cada recurso que podría necesitar un proceso. Más formalmente:

$$\forall i, j, N_{ij} = Dm_{ij} - A_{ij}$$

Se dice que un sistema está en **estado seguro**, si existe un orden en el cuál todos los procesos pueden finalizar su ejecución, sin que se produzca un interbloqueo. Si este orden no existe nos encontramos en un **estado inseguro**.

Formalmente, un sistema se encuentra en estado seguro si existe una secuencia de procesos  $\langle p_1, \dots, p_n \rangle$ , donde cada proceso  $p_i$  puede finalizar con los recursos que tiene asignados más los disponibles más los recursos asignados a todos los procesos  $p_j$ , para todo  $j < i$ .

Un estado seguro es un estado sin interbloqueo. Del mismo modo, un estado de abrazo mortal es un estado inseguro. Sin embargo, un estado inseguro no implica necesariamente la existencia del interbloqueo, simplemente indica que una secuencia desafortunada de eventos puede ocasionar un interbloqueo. En la figura 6.4 podemos ver la relación existente entre estos conceptos.



**Figura 6.4:** Relaciones entre estado seguro, inseguro e interbloqueo

Esto nos sugiere una estrategia de predicción de interbloqueo, que consiste en asegurar que el sistema esté siempre en un estado seguro. El algoritmo del banquero es una estrategia de este tipo.

El estado inicial en el que no se ha asignado ningún recurso y todos están disponibles, es siempre seguro. Partiendo de éste, cuando un proceso realiza una petición, se simula su concesión, se actualiza el estado del sistema, y se determina si nos encontramos en un estado seguro. Si es así, se concede la solicitud y, si no, se bloquea al proceso hasta que sea seguro.

**Algoritmo 6.2****Algoritmo del banquero**

```

Sea  $P_i$ , el proceso que hace una petición
si  $\exists$  un recurso  $R_k$ , tal que  $Solicitud[i][k] > Necesidad[i][k]$ 
entonces
    Error, el proceso  $P_i$  supera su demanda máxima inicial
fin si
si  $\exists$  un recurso  $R_k$ , tal que  $Solicitud[i][k] > Disponible[i][k]$ 
entonces
    bloquear al proceso  $P_i$ , porque no hay recursos disponibles
fin si
para todo recurso,  $R_j$ ,
hacer
     $Disponible[j] = Disponible[j] - Solicitud[i][j]$ 
     $Asignación[i][j] = Asignación[i][j] + Solicitud[i][j]$ 
     $Necesidad[i][j] = Necesidad[i][j] - Solicitud[i][j]$ 
fin para
ejecutar algoritmo de seguridad
si Sistema == SEGURO
entonces
    asignar los recursos al proceso  $P_i$ 
si no
    para todo recurso,  $R_j$ ,
    hacer
         $Disponible[j] = Disponible[j] + Solicitud[i][j]$ 
         $Asignación[i][j] = Asignación[i][j] - Solicitud[i][j]$ 
         $Necesidad[i][j] = Necesidad[i][j] + Solicitud[i][j]$ 
    fin para
    bloquear al proceso  $P_i$ 
fin si

```

Para implementar este algoritmo es necesario disponer de cierta información, que vendrá dada por las estructuras anteriores más la matriz *Solicitud*. Ésta contiene las solicitudes de recursos que realizan los distintos procesos. Así,  $Solicitud_{ij}$  indica la solicitud del proceso  $p_i$  del recurso  $r_j$ . El algoritmo 6.2 muestra una posible implementación. Siempre que un proceso pide recursos, se comprueba, en primer lugar, si esta petición está dentro de las necesidades máximas declaradas inicialmente, si no fuera así se produciría un error. Posteriormente se determina si existen recursos suficientes para atenderla, en caso afirmativo se simula la asignación comprobándose si el sistema queda en estado seguro. Si no había recursos

o el sistema llega a estado inseguro se bloquea al proceso.

Para determinar si el sistema se encuentra o no en un estado seguro podemos utilizar las estructuras vistas anteriormente. El algoritmo 6.3 muestra el mecanismo que podría emplearse para esta comprobación.

#### Algoritmo 6.3

#### Algoritmo de seguridad

```

para todo recurso,  $R_j$ ,
hacer
    Trabajo[j] = Disponible[j];
fin para
para todo proceso,  $P_i$ ,
hacer
    Acabar[i] = Falso;
fin para
mientras  $\exists$  un proceso  $P_i$ , tal que Acabar[i] == Falso y
     $\forall$  recurso  $R_j$ , Necesidad[i][j]  $\leq$  Trabajo[j]
hacer
    Acabar[i] = Verdadero;
    para todo recurso,  $R_j$ ,
    hacer
        Trabajo[j] = Trabajo[j] + Asignación[i][j];
    fin para
fin mientras
Sistema = SEGURO;
para todo proceso,  $P_i$ ,
hacer
    si Acabar[i] == Falso
    entonces
        Sistema = INSEGURO;
    fin si
fin para

```

El algoritmo del banquero garantiza que se terminarán los procesos avanzando de estado seguro en estado seguro para evitar el interbloqueo, permitiendo las condiciones de exclusión mutua, retener y esperar, y no apropiación.

#### Inconvenientes del algoritmo del banquero

El algoritmo del banquero ha sido utilizado para predecir interbloqueos en sistemas con pocos recursos, y es una política menos restrictiva que la prevención. Sin embargo, es poco práctico para la mayoría de los sistemas por varias razones:

- Los procesos deben declarar por anticipado el número máximo de recursos que van a necesitar. Esto es factible en un sistema por lotes, pero en un sistema interactivo es muy difícil de conocer.

- El número de recursos de cada clase debe permanecer constante. Esto es difícil de conseguir ya que, bien por averías, bien por mantenimiento preventivo, no podemos contar con un número de recursos siempre constante.
- Los procesos deben liberar explícitamente los recursos antes de terminar.
- El algoritmo tiene un alto coste en tiempo cuando el número de procesos y de peticiones es muy elevado, ya que se necesitan  $m \times n^2$  operaciones, donde  $n$  es el número de procesos y  $m$  el de recursos.
- El algoritmo garantiza que todas las peticiones serán concedidas dentro de un intervalo de tiempo finito, pero no especificado. Está claro que, en sistemas reales, se necesitan garantías mucho mayores que ésta.
- De forma similar, el algoritmo supone que los procesos devolverán los recursos dentro de un intervalo de tiempo finito. También en este caso se necesitan garantías mucho mayores que ésta para sistemas reales.
- El algoritmo asume el peor de los casos y, por lo tanto, realiza una pobre utilización de recursos manteniendo sin asignar recursos vitales, con tal de permanecer en estado seguro.
- Castiga a los procesos que requieren muchos recursos, de tal modo que los que necesiten pocos pueden provocar la inanición de los primeros.
- Por otro lado, los procesos a considerar deben ser independientes; esto es, el orden en que se ejecuten no debe estar forzado por condiciones de sincronización. Por este motivo, la ejecución estrictamente secuencial de los procesos, es la peor condición a imponer.

#### 6.4.3 Detección y recuperación

Las estrategias de prevención y predicción son muy poco flexibles a la hora de asignar los recursos porque imponen muchas restricciones, dando lugar a una baja utilización de éstos. Sin embargo, las estrategias de detección no imponen restricciones al acceso a los recursos por parte de los procesos, permitiéndose la aparición de los interbloqueos. En estos sistemas es necesario disponer de un algoritmo que compruebe la existencia de interbloqueo, y otro que nos permita hacerlo desaparecer en caso de que exista.

La comprobación de la existencia del interbloqueo implica una cierta sobrecarga en el sistema. Por un lado, requiere un gasto extra en tiempo de procesamiento

para ejecutar el algoritmo y, por otro, necesita un coste para mantener la información sobre la situación actual del sistema.

Antes de estudiar las distintas estrategias de detección y recuperación, el diseñador del sistema debe plantearse la siguiente cuestión, ¿cuándo debe realizarse la comprobación o control de la existencia del interbloqueo? Existen varias posibilidades:

- Cada vez que se deniega una solicitud. Esto permite detectar rápidamente la existencia de interbloqueo pero consume mucho tiempo de CPU.
- A intervalos regulares. En este caso habría que ajustar el tamaño del intervalo, teniendo en cuenta que si es demasiado pequeño también se consumirá mucho tiempo de CPU, y si es muy grande, se pueden haber producido muchos interbloqueos, siendo más difícil la recuperación.
- Cuando desciende el rendimiento del sistema. Se evita ejecutar innecesariamente el algoritmo, pero hay que estar controlando continuamente el rendimiento del sistema.

Una vez detectado un interbloqueo en un sistema, se debe seguir alguna estrategia para su recuperación. Ésta implica la pérdida de parte o de la totalidad del trabajo realizado por algunos procesos, pero este es el coste de este método.

#### 6.4.3.1 Detección por grafos

En el apartado 6.3 vimos cómo se pueden representar gráficamente los procesos y recursos del sistema. Esta representación nos sirve para determinar si existe o no un interbloqueo. Para ello se aplica una técnica de **reducción de grafos**. Un grafo puede ser reducido por el proceso  $P_i$ , si:

1. No espera ningún recurso.
2. Está esperando un recurso del que existen unidades disponibles.

Cuando un grafo es reducido por el proceso  $P_i$  libera todas las unidades de los recursos asignados. Se dice que un grafo es **irreducible** si no puede ser reducido por algún proceso. Un grafo es **completamente reducible** si existe una serie de transformaciones que reducen el grafo a una situación donde no existen aristas de solicitudes ni de asignaciones.

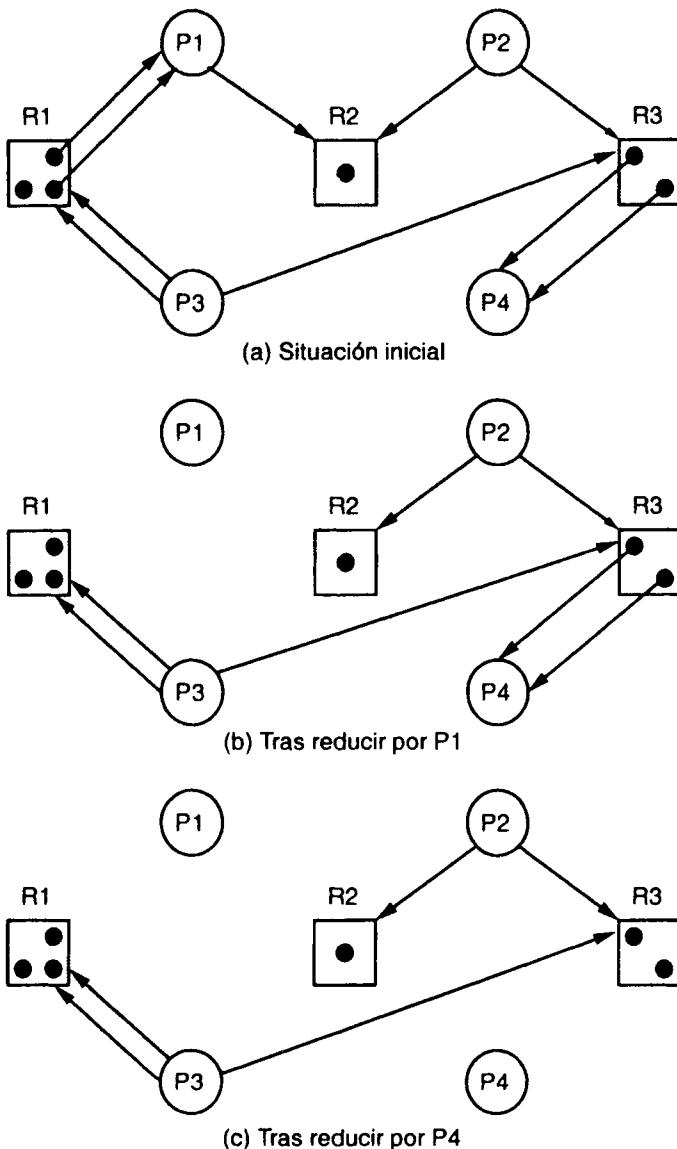


Figura 6.5: Ejemplo de reducción de un grafo

En la figura 6.5(a) tenemos un ejemplo donde existen tres recursos y cuatro procesos. En ella, podemos observar que P1 tiene dos recursos de tipo R1 asignados, y existe un recurso de tipo R2 disponible para atender la única solicitud que está haciendo. Por tanto, es posible reducir el grafo por este proceso. La nueva situación aparece en la figura 6.5(b). En ella, P4 tiene todos sus recursos asignados, por lo que se puede reducir el grafo. La figura 6.5(c) muestra el grafo tras reducir por P4. Finalmente, aunque no se muestra, el grafo se puede reducir por P2 y P3, al existir recursos para ambos.

#### 6.4.3.2 Algoritmo de detección

El método gráfico descrito anteriormente es difícil de implementar en un sistema operativo. A continuación vamos a ver un algoritmo más simple, que consiste en investigar toda posible secuencia de asignación para los procesos que aún tengan que completarse.

##### Algoritmo 6.4

##### Algoritmo de detección

```

para todo proceso,  $P_i$ ,
hacer
    Bloqueado[i] = Verdadero
fin para
para todo recurso,  $R_j$ ,
hacer
    Trabajo[j] = Disponible[j]
fin para
mientras  $\exists$  un proceso  $P_i$ , tal que Bloqueado[i] == Verdadero y
             $\forall$  recurso,  $R_j$ ,  $Solicitud[i][j] \leq Trabajo[j]$ 
hacer
    para todo recurso,  $R_j$ ,
    hacer
        Bloqueado[i] = Falso
        Trabajo[j] = Trabajo[j] + Asignación[i][j]
    fin para
fin mientras
Interbloqueo = Falso
para todo proceso,  $P_i$ ,
hacer
    si Bloqueado[i] == Verdadero
    entonces
        Interbloqueo = Verdadero
    fin si
fin para
```

El algoritmo 6.4 muestra una posible implementación. Como se puede apreciar

es muy parecido al algoritmo 6.3 empleado para determinar si un sistema está en estado seguro, empleando las mismas estructuras.

#### 6.4.3.3 Recuperación de un interbloqueo

Una vez detectado el interbloqueo tenemos dos opciones para hacerlo desaparecer. Una consiste en terminar uno o más procesos para romper la espera circular, la otra en apropiarse de recursos de uno o más procesos implicados en el interbloqueo.

##### Terminación de procesos

Con esta estrategia eliminamos procesos, recuperándose los recursos que tenían asignados con objeto de que el sistema pueda continuar con normalidad. Para ello tenemos varias opciones:

1. Eliminar todos los procesos interbloqueados. Es una de las soluciones más adoptadas en los sistemas operativos, pero presenta un coste muy elevado.
2. Ir abortando sucesivamente todos los procesos interbloqueados hasta que desaparezca el abrazo mortal, es decir, hasta que haya suficientes recursos libres como para continuar sin interbloqueo la ejecución de los restantes procesos. Es necesario establecer un orden a la hora de seleccionar los procesos, normalmente se seguirá un criterio de coste mínimo. Tras la eliminación de cada uno habrá que ejecutar un algoritmo de detección para determinar si aún existe el interbloqueo.

En la segunda estrategia la decisión de eliminar un proceso no es fácil y pueden seguirse diversos criterios con objeto de que la pérdida que sufra el sistema sea mínima. Algunos de éstos pueden ser:

- Cuántos procesos se verán implicados en la reanudación.
- Cantidad y calidad de los recursos utilizados (qué tipo, si son fáciles de apropiar, etc.).
- La prioridad del proceso.
- Tiempo de ejecución transcurrido.
- Cuántos recursos más necesita para finalizar.

### Apropiación de recursos

Consiste en retirar o apropiarse de los recursos de uno de los procesos interbloqueados para dárselos a otro de los implicados, permitiéndole que termine y salga del interbloqueo. En este caso hay que afrontar las siguientes cuestiones:

**Selección de la víctima** Determinar qué procesos y qué recursos son apropiados, siempre siguiendo el criterio del mínimo coste, al igual que cuando se abortan procesos. También se ha de evitar la inanición, para que no siempre se apropie al mismo proceso.

**Apropiación del recurso** Hemos de tener en cuenta que el proceso al que le quitamos el recurso, en un futuro tendrá que continuar y habrá que asignarle de nuevo dicho recurso como si no hubiera pasado nada. Es por este motivo por lo que sólo se pueden apropiar ciertos recursos, al igual que en la prevención cuando se niega la condición de no apropiación (ver apartado 6.4.1.2).

## 6.5 Resumen

La utilización de recursos críticos por parte de los procesos puede ocasionar la aparición de una situación conocida como interbloqueo. Ésta se puede definir como el bloqueo permanente de un conjunto de procesos que compiten por los recursos del sistema o que se comunican entre ellos.

Las técnicas clásicas para abordar este problema se clasifican en tres grandes grupos: prevención, predicción y detección. La elección de una u otra dependerá del sistema y de la información que poseamos sobre él. No existe una solución óptima al problema, ya que todas ellas implican o una pobre utilización de los recursos o bien, una sobrecarga en el sistema. De este modo, en muchos sistemas donde se supone que su probabilidad de aparición es pequeña no se aplica ninguna estrategia, y en caso de aparecer se opta por reiniciar el sistema.

Los métodos de prevención se caracterizan por negar una de las condiciones necesarias para que se produzca el interbloqueo. Son muy restrictivos en la asignación de recursos a los procesos, proporcionando un uso ineficiente de éstos.

Las estrategias de predicción se basan en el conocimiento previo de cierta información, con objeto de determinar si existe una secuencia de eventos que pueda conducir o no a un interbloqueo. De esta forma, cuando un proceso solicita un recurso, se determina si su asignación puede conducir o no a un interbloqueo.

En el primer caso se le deniega el acceso al recurso. El algoritmo de predicción más conocido es el del banquero.

Por el contrario, las estrategias de detección son más liberales en el sentido de no imponer ninguna restricción ni tener conocimiento alguno de los procesos. Estos algoritmos permiten la aparición del interbloqueo en el sistema, proporcionan un método para su detección, y si lo detectan, intentan hacer que desaparezca con el menor coste posible.

## 6.6 Ejercicios

1. ¿Qué diferencias existen entre las estrategias de prevención, predicción y detección?
2. ¿Cuál es la diferencia principal entre el interbloqueo y la inanición? ¿En qué se parecen?
3. ¿Es posible que aparezca un interbloqueo en el que esté implicado un único proceso? ¿Y con una única clase o tipo de recurso?
4. Diga si las siguientes afirmaciones son verdaderas o falsas, razonando su respuesta:
  - (a) Un sistema que en un momento dado presenta espera circular está en interbloqueo.
  - (b) El algoritmo del banquero está limitado porque el número de recursos ha de ser fijo.
  - (c) La exclusión mutua es una condición que no se puede eliminar.
  - (d) Se puede prevenir el interbloqueo mediante la asignación de prioridades, de tal modo que un recurso se asignará al proceso de mayor prioridad.
5. En un grafo de asignación de recursos, ¿es posible la existencia de un ciclo sin que exista un interbloqueo? Razone la respuesta. En caso afirmativo explique por qué y dibuje una gráfica de ejemplo.
6. Las estrategias de predicción necesitan cierta información tal como la demanda máxima de recursos que van a necesitar los procesos. Pero, ¿cómo podríamos determinar esta demanda máxima en un sistema por lotes y en un sistema interactivo?

7. Considere la siguiente instantánea de un sistema que emplea un método de predicción basado en el algoritmo del banquero:

	Asignación			
	R1	R2	R3	R4
P1	0	0	1	2
P2	1	0	0	0
P3	2	3	5	4
P4	0	6	3	1
P5	0	0	1	4

	Demanda			
	R1	R2	R3	R4
P1	0	0	1	2
P2	1	7	5	0
P3	3	3	5	6
P4	0	6	5	1
P5	10	6	5	6

Disponible	
R1	1
R2	5
R3	2
R4	0

Conteste de forma razonada a las siguientes preguntas:

- (a) ¿Cuál es el contenido de la matriz *Necesidad*?
  - (b) ¿Está el sistema en estado seguro?
  - (c) Si llega la solicitud del proceso P2 = (0, 4, 2, 0), ¿puede ser satisfecha sin problema?
  - (d) Si a continuación llega la solicitud P5 = (1, 0, 0, 0), ¿podría ser satisfecha sin problema?
  - (e) ¿Y si a continuación llega la solicitud P4 = (1, 0, 0, 0)?
8. Suponga un sistema que aplica una estrategia de detección y recuperación de interbloqueos. La situación inicial viene representada por las siguientes estructuras:

	Asignación			
	R1	R2	R3	R4
P1	0	0	1	2
P2	1	0	0	0
P3	1	2	1	1
P4	0	1	2	2
P5	0	0	1	3

	Solicitud			
	R1	R2	R3	R4
P1	0	0	1	2
P2	1	2	1	0
P3	2	1	2	0
P4	0	3	1	2
P5	0	3	2	3

Disponible	
R1	1
R2	2
R3	2
R4	0

Se pide:

- (a) Modele mediante un grafo de asignación de recursos la situación actual del sistema.
- (b) ¿Existe interbloqueo? En caso afirmativo indique los procesos implicados.



---

Parte 4

# MEMORIA

---



## Administración de la memoria

---

La gestión de la memoria es un aspecto fundamental de los sistemas operativos, ya que para que los procesos puedan ejecutarse deben residir en ella. En este capítulo se estudiarán los requisitos de gestión de memoria, así como diversos esquemas de asignación que exigen que la imagen del proceso esté cargada en ella completamente. Finalmente, se introducirán los aspectos básicos de los esquemas de paginación y segmentación, que se utilizan en combinación con los sistemas de memoria virtual, que se tratarán en el capítulo siguiente.

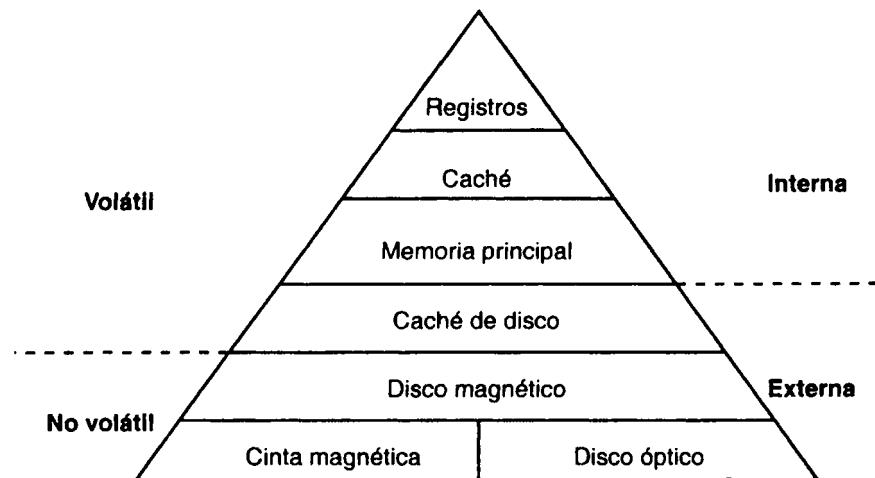
### 7.1 Jerarquía del almacenamiento

Un sistema de computación necesita disponer de un sistema de almacenamiento eficiente. El sistema ideal sería el que tuviera una gran capacidad para poder ejecutar la mayor cantidad de programas posible, un tiempo de acceso bajo para obtener un mejor rendimiento, y un coste razonable. Sin embargo, no existe ningún sistema de almacenamiento de información que cumpla estos tres requisitos simultáneamente. Normalmente se establecen las siguientes relaciones:

- A menor tiempo de acceso, mayor coste por bit.
- A mayor capacidad, menor coste por bit.

- A mayor capacidad, mayor tiempo de acceso.

La solución al problema consiste en no emplear un único componente de memoria, sino una **jerarquía de almacenamiento**, como la que aparece en la figura 7.1.



**Figura 7.1:** Jerarquía del almacenamiento

Los distintos tipos de memoria que se muestran pueden ser volátiles o no volátiles según si la información almacenada se mantiene de forma permanente o no. Del mismo modo, se pueden clasificar en sistemas de almacenamiento externos o internos, dependiendo de si encuentran o no en dispositivos externos al sistema.

En el nivel superior se encuentra la memoria más rápida, de menor capacidad y mayor coste, los registros del procesador; en contraposición con el nivel más bajo en el que aparece la memoria más lenta, de mayor capacidad y más económica. La **memoria principal** es el elemento principal de almacenamiento interno en el sistema de computación. Con objeto de mejorar el rendimiento se suele contar con un nivel de almacenamiento más rápido y de menor capacidad, que se conoce como **memoria caché**. Ésta se utiliza como almacenamiento intermedio de los datos usados recientemente, para que un posterior uso no implique acceder a memoria principal.

Los dispositivos de almacenamiento masivo externos, tales como los discos, cintas magnéticas y los discos ópticos constituyen un sistema de almacenamiento no volátil de mucha mayor capacidad que la memoria principal. Esta memoria externa no volátil se conoce como **memoria auxiliar o secundaria**. Los discos

magnéticos también se suelen emplear para proporcionar una extensión a la memoria principal, conocida como **memoria virtual**, que se discutirá en el capítulo siguiente.

En la figura 7.1 aparece también otro nivel de almacenamiento que es la **caché de disco**, cuyo funcionamiento es similar a la situada entre los registros y la memoria principal.

Podemos decir que a medida que bajamos en la jerarquía se cumplen las siguientes condiciones:

1. Disminuye el coste por bit.
2. Aumenta la capacidad.
3. Aumenta el tiempo de acceso.
4. Disminuye la frecuencia de acceso por parte del procesador.

Este último punto, la disminución de la frecuencia de acceso, es la clave para el éxito de esta jerarquía. La base para la validez de esta afirmación viene dada por el principio conocido como **cercanía de referencias** o **principio de localidad** [Denn68]. En él se establece que durante la ejecución de un proceso, las referencias a memoria que hace el procesador tienden a estar agrupadas. Es decir, en períodos largos de tiempo los grupos de localizaciones referenciados cambian, pero en períodos cortos, el procesador suele trabajar con un conjunto de referencias a memoria que no cambia. Teniendo esto en cuenta, es posible organizar la información a través de la jerarquía, de tal forma que el porcentaje de accesos sea menor a medida que descendemos en ella.

## 7.2 Traducción de direcciones

En el apartado 3.1 hicimos referencia a que un programa se convierte en proceso cuando se carga en memoria. Para ello, el programa debe pasar previamente por una serie de fases como son la compilación, el enlazado y por último su carga en la memoria principal.

A lo largo de estas fases las direcciones de memoria se pueden expresar de distintas formas. Así, podemos distinguir entre **direcciones lógicas, relativas y físicas**. Una dirección lógica, también denominada **simbólica**, es una referencia a una localización de memoria independiente de su ubicación en memoria principal;

por tanto, antes de acceder a la memoria habrá que realizar una traducción de ésta a una dirección física.

Una dirección relativa es un caso particular de dirección lógica, que se expresa como una localización relativa a algún punto conocido, usualmente el principio del programa. Una dirección física o **absoluta**, es una localización en memoria principal. La figura 7.2 muestra los posibles direccionamientos.

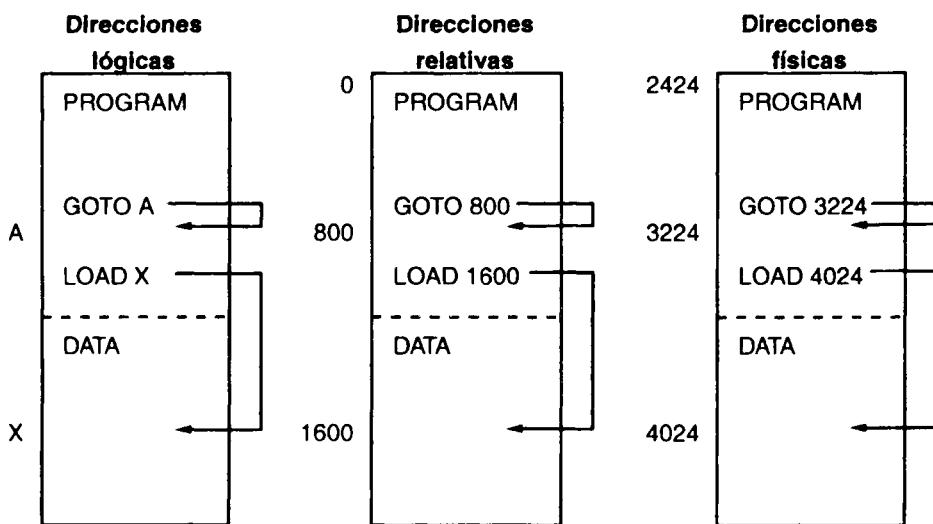


Figura 7.2: Tipos de direccionamientos

Como hemos dicho anteriormente, en las diferentes etapas por las que pasa el programa antes de poder ser ejecutado, las direcciones de memoria se pueden representar de distintas formas. Sin embargo, hemos de tener en cuenta que en el momento de la ejecución siempre se refieren direcciones físicas, por lo que si inicialmente se utilizan direcciones lógicas éstas tendrán que ser traducidas en algún momento. Esta **traducción o ligadura de direcciones** se puede producir:

- En tiempo de programación o compilación.
- En tiempo de carga.
- En tiempo de ejecución.

Si en el momento de hacer el programa, el programador proporciona direcciones absolutas o si el compilador genera este tipo de direcciones, el módulo de carga que se le pasa al cargador contendrá direcciones absolutas, por lo que tendrá que ser colocado siempre en la misma zona de memoria. Esto presenta dos

inconvenientes. Por un lado, no permite cambiar el proceso de ubicación durante su ejecución, y por otro, es necesario conocer de antemano en qué zona de la memoria se va a situar. Esto requerirá tener conocimiento de la técnica de gestión de memoria que se emplea, con objeto de saber qué huecos libres existen en ella.

La desventaja de generar direcciones absolutas antes de la carga es que el módulo de carga resultante sólo puede ser colocado en una región específica de la memoria principal. Cuando ésta es compartida por muchos procesos, esta forma de trabajo es poco flexible, por tanto, es mejor tener un módulo de carga que pueda ser colocado en cualquier parte de la memoria.

Para conseguir esto, el compilador no debe producir direcciones absolutas sino direcciones que sean relativas a algún punto conocido, tal como el comienzo del programa. Al principio del módulo de carga se le asigna la dirección relativa 0, y el resto de referencias dentro del módulo se expresan en relación a ésta. De este modo, teniendo todas las referencias a memoria expresadas de forma relativa, es fácil para el cargador colocar el módulo en la dirección deseada. Si éste va a ser colocado a partir la dirección  $x$ , sólo tendrá que añadir esta dirección a cada referencia a medida que vaya cargando el módulo en memoria.

Con este tipo de traducción, el problema que se presenta es que si se saca el proceso de memoria principal, cuando vuelva a ella deberá ser colocado en la misma posición donde estaba inicialmente. Otro inconveniente es que no se podrá cambiar de ubicación al proceso durante su ejecución.

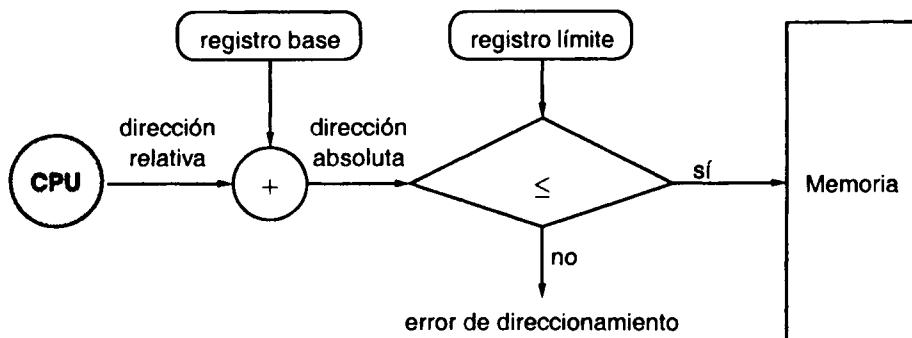
La última posibilidad es retrasar el cálculo de las direcciones absolutas hasta que se necesiten, es decir, en tiempo de ejecución. En este caso, el módulo de carga se sitúa en memoria principal con todas las referencias a memoria en forma relativa, y a medida que se van ejecutando las instrucciones se calculan las direcciones absolutas. Para que esto no degrade el rendimiento del sistema, la traducción se debe realizar mediante un mecanismo hardware en vez de por software.

Este cálculo dinámico de direcciones es el más flexible ya que permite que un programa pueda cargarse en cualquier zona de la memoria, y durante la ejecución de un proceso, éste podrá intercambiarse sin problemas y, si es necesario, cambiar de ubicación en memoria principal.

En este esquema de traducción, cuando un proceso pasa al estado de ejecución, un registro especial del procesador, llamado registro base, se carga con la dirección de comienzo del proceso en memoria principal. También suele existir un registro límite que indica la última dirección correspondiente al proceso. Durante su ejecución nos encontraremos con direcciones relativas que tienen que ser traducidas a direcciones absolutas. La figura 7.3 muestra la forma en que se hace

normalmente esta traducción. El procesador realiza los siguientes pasos:

1. Añade el valor del registro base a la dirección relativa, obteniendo así la absoluta.
2. Compara la dirección resultante con el valor del registro límite. Si la dirección está dentro de los límites, se puede proceder a la ejecución de la instrucción. Si no es así, se genera una excepción, para que el sistema operativo trate el error. De este modo se proporciona un método de protección donde la imagen de cada proceso se encuentra aislada por el contenido de los registros base y límite, estando a salvo del acceso por parte de otros procesos.



**Figura 7.3:** Soporte hardware para la traducción de direcciones relativas

### 7.3 Funciones del administrador de la memoria

Históricamente la memoria principal ha sido un recurso costoso, por lo que los diseñadores de sistemas operativos han intentado obtener el máximo rendimiento de ella. En la actualidad la memoria principal es más barata pero sigue siendo importante administrarla adecuadamente porque esto va a influir en gran medida en el rendimiento global del sistema.

La parte del sistema operativo que se encarga del manejo de la memoria se denomina administrador de la memoria. Entre las labores que debe realizar están:

- Controlar qué partes de la memoria están en uso y cuáles están disponibles.

- Decidir dónde situar los procesos en memoria, y controlar en qué zona de la memoria residen en cada momento.
- Asignar memoria a los procesos cuando la necesiten y retirársela cuando terminen.
- Controlar el intercambio de procesos entre memoria principal y secundaria.

Para llevar a cabo de forma adecuada estas funciones, el administrador de la memoria debe proporcionar **reubicación**, protección y compartición.

Por reubicación entendemos la posibilidad de que un proceso a lo largo de su vida pueda cambiar de localización dentro de la memoria. En sistemas de multiprogramación esto suele ser necesario por ejemplo, si necesitamos intercambiar el proceso, ya que cuando éste vuelva a la memoria principal puede no ser posible cargarlo en el mismo lugar donde estaba inicialmente; aún en el caso de que no necesitemos intercambiar el proceso, muchos sistemas de administración de la memoria necesitan cambiar los procesos de localización para hacer así un mejor uso de la memoria.

Tanto en sistemas de monoprogramación como en los de multiprogramación, es necesario disponer de un sistema de protección adecuado, ya que un proceso de usuario no debe acceder a la zona de memoria donde reside el sistema operativo, ni tampoco a la ocupada por otros procesos.

En los sistemas de multiprogramación nos podemos encontrar con varios procesos ejecutando el mismo código; en estos casos, para ahorrar memoria es conveniente que puedan compartirlo. Para ello, el mecanismo de protección que proporciona el sistema debe ser lo suficientemente flexible como para permitir la compartición de ciertos bloques de memoria por parte de distintos procesos.

## 7.4 Esquemas de asignación de la memoria

En este capítulo vamos a analizar distintos esquemas de administración de la memoria. La elección de uno de ellos para un sistema concreto depende de muchos factores, principalmente del diseño hardware de éste. Una clasificación de estos esquemas es la siguiente:

**Asignación contigua** Exigen que la imagen del proceso se cargue en memoria ocupando un espacio de direcciones contiguo. Ejemplos de esquemas de asignación de este tipo son:

*Máquina desnuda* No existe sistema operativo, por tanto el proceso de usuario puede ocupar toda la memoria.

*Sistemas de monoprogramación* En memoria principal residen el sistema operativo y un programa de usuario.

*Sistemas de multiprogramación con particiones* Se caracterizan por dividir la memoria en particiones, residiendo en cada una de ellas un proceso.

**Asignación no contigua** En este caso no se exige que la imagen del proceso ocupe un espacio de direcciones contiguo. Se consideran dos esquemas de este tipo, la **paginación** y la **segmentación**. Debemos tener en cuenta que estos esquemas se utilizan en los sistemas de memoria virtual que se estudiarán en el capítulo 8, dejando para éste la descripción de su funcionamiento.

## 7.5 Sistemas de monoprogramación

Desde el punto de vista del manejo de la memoria, los sistemas de monoprogramación son mucho más simples que los de multiprogramación, ya que la memoria se divide en dos áreas contiguas. Una de ellas está asignada permanentemente a la parte residente del sistema operativo, mientras que la otra se asigna a los procesos transitorios. En ésta se pueden cargar tanto procesos de usuario como las partes no residentes del sistema operativo. Esta forma de gestión de la memoria la utilizan habitualmente sistemas tales como MS-DOS.

Con el fin de proporcionar un área contigua de memoria libre para los procesos transitorios, el sistema operativo se sitúa normalmente en un extremo, cuya elección suele venir determinada por la ubicación del vector de interrupciones. La figura 7.4 muestra todas las posibilidades.

Los sistemas de monoprogramación no suelen proporcionar protección de la zona de memoria donde residen los procesos de usuario, ya que en cada momento sólo permite que haya un único proceso residente en memoria. Sin embargo, es deseable proteger el código del sistema operativo. Esta protección se puede implementar mediante un registro límite, que contiene la dirección de memoria a partir de la cual puede acceder el proceso de usuario. Cada vez que éste hace referencia a una dirección de memoria, se comprueba si le está permitido o no, haciendo uso de este registro, como se puede ver en la figura 7.5.

Algunos ejemplos de sistemas operativos de monoprogramación son el IBM 1130 y el HP 2116B.

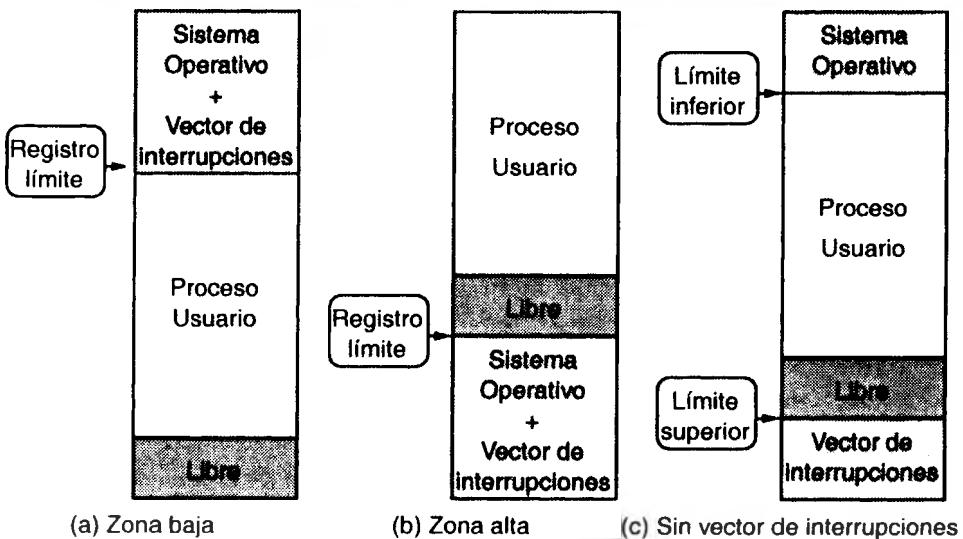


Figura 7.4: Organización de la memoria en monoprogramación

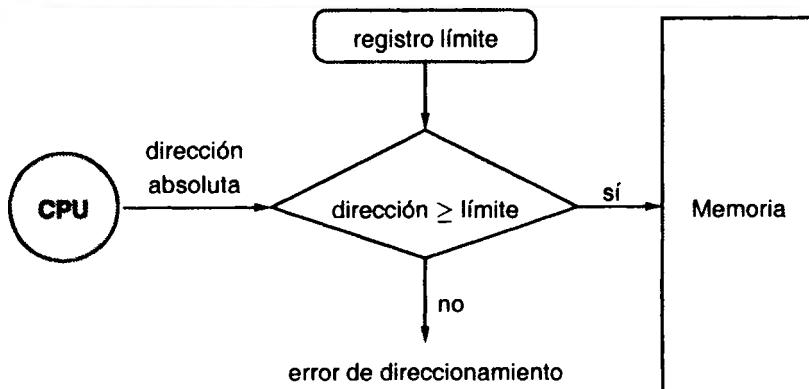
## 7.6 Multiprogramación con particiones fijas

En sistemas de multiprogramación la zona de memoria disponible para los procesos transitorios debe ser compartida por varios de éstos, por tanto, se necesita un esquema de manejo de la memoria más complejo.

El esquema más simple para estos sistemas consiste en dividir la memoria disponible para los procesos transitorios en un número fijo de particiones, cada una de las cuales puede ser asignada a un proceso. Este esquema se conoce como **particiones fijas**. Un ejemplo de un sistema operativo que usó esta técnica fue el sistema de IBM OS/MFT (*Multiprogramming with a Fixed Number of Tasks*).

### 7.6.1 Selección del tamaño de las particiones

La división de la memoria en particiones se realiza durante el proceso de generación del sistema, por lo que el número y tamaño de éstas permanece fijo a partir de este momento. Hay que tener en cuenta que el número de particiones va a determinar el grado de multiprogramación máximo que podemos alcanzar, ya que en una partición sólo se puede cargar un proceso. El tamaño de las particiones también es importante porque limita el tamaño máximo de los procesos que podemos cargar en ellas. Hay que tener en cuenta también que si el tamaño de un proceso es menor



**Figura 7.5:** Protección de la memoria en sistemas de monoprogramación

que el de la partición, toda la memoria que sobra se desperdicia ya que no puede ser utilizada por otro proceso, esto es lo que se conoce como **fragmentación interna**. Por tanto, tendremos que hacer estimaciones acerca de cuál puede ser el tamaño máximo de un proceso y del tamaño más usual de éstos.

La figura 7.6 muestra las dos alternativas posibles con respecto al tamaño de las particiones. Una posibilidad es hacer uso de particiones de igual tamaño y la otra tener particiones de tamaños diferentes. El uso de particiones de tamaños diferentes proporciona una mayor flexibilidad al esquema de particiones fijas.

### 7.6.2 Algoritmos de colocación

Cuando se utilizan particiones del mismo tamaño, cualquier partición puede ser empleada para alojar un proceso, siempre que el tamaño de éste sea menor o igual al de la partición. Si todas las particiones estuvieran ocupadas, habría que recurrir al intercambio de procesos. El planificador a medio plazo tendría que seleccionar el proceso a sacar de la memoria, preferentemente uno que no esté en estado listo.

Cuando tenemos particiones de tamaños diferentes, hay distintas posibilidades a la hora de asignar los procesos a éstas. Podríamos tener una cola de procesos asignada a cada partición (figura 7.7(a)), de modo que cada proceso se situase en la cola asociada a la partición de menor tamaño donde éste cupiera. Cuando una partición queda libre se elige un proceso de la cola asociada para ocuparla. La ventaja que presenta es que se minimiza la cantidad de memoria que se desperdicia dentro de cada partición, es decir, se reduce la fragmentación interna. Sin embargo, desde el punto de vista del sistema tiene el inconveniente de que

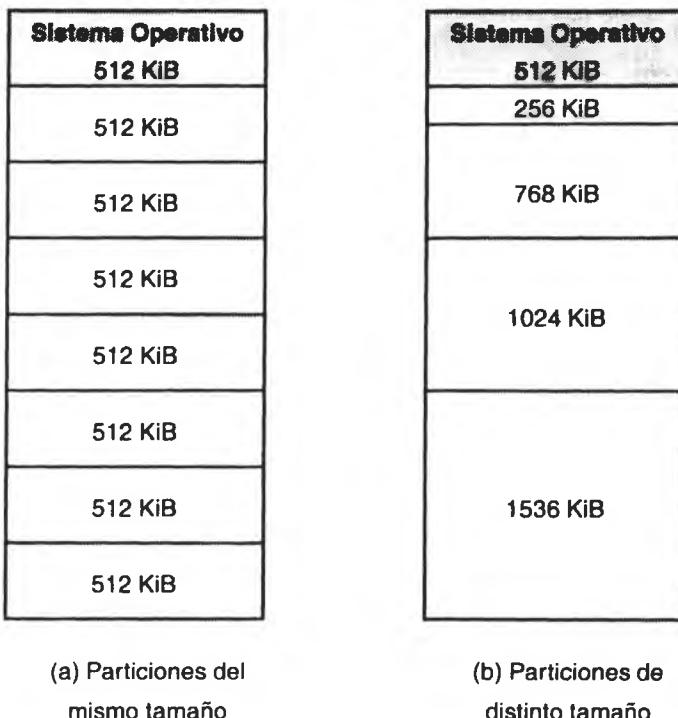
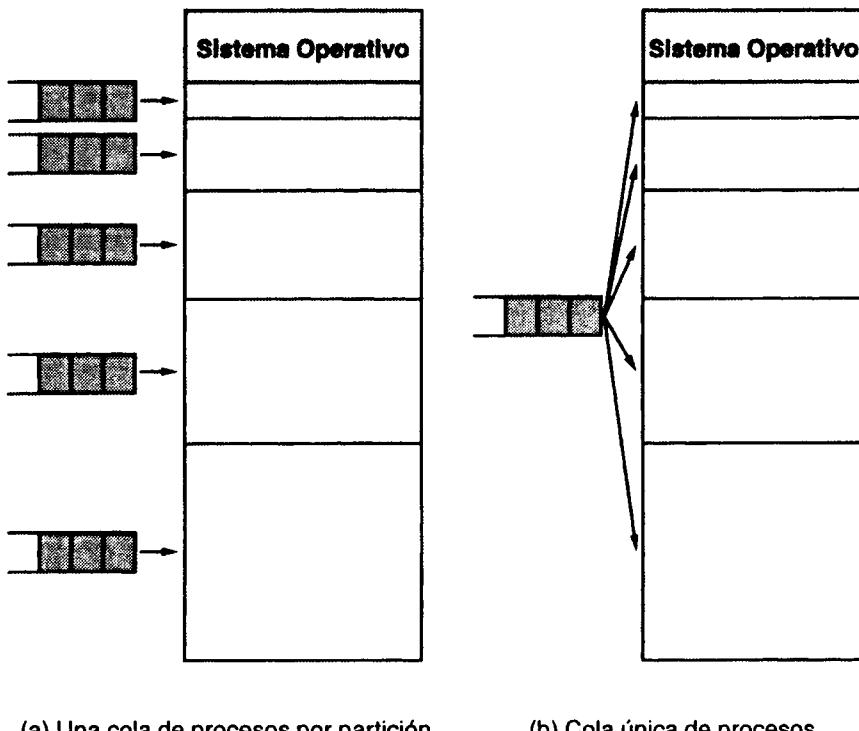


Figura 7.6: Diseño de particiones fijas en una memoria de 4 MiB

puede haber particiones grandes vacías mientras que tenemos procesos esperando en colas asociadas a particiones pequeñas.

Otra alternativa consiste en tener una única cola de procesos como en la figura 7.7(b). En este caso, cuando una partición queda libre, es necesario disponer de una estrategia que determine qué proceso de la cola se va a cargar en ella. Una posibilidad es emplear el algoritmo del **primer ajuste**, que selecciona el primer proceso de la cola que quepa en la partición. Como no es deseable desperdiciar una partición grande con un proceso pequeño, una estrategia diferente consiste en buscar en toda la cola y tomar el proceso más grande que quepa en ella. Este segundo algoritmo discrimina a los procesos pequeños frente a los grandes, y se conoce con el nombre de **mejor ajuste**.

La labor de asignar los procesos a las particiones siguiendo una determinada estrategia la realizará el planificador a largo plazo del que se habló en el apartado 4.2.1.



**Figura 7.7:** Alternativas de planificación en multiprogramación con particiones fijas

### 7.6.3 Elementos de control

En un sistema de este tipo se necesita saber en todo momento qué particiones están ocupadas y cuáles están libres, para ello se utiliza una **tabla de particiones**. Ésta posee una entrada por cada partición, donde se indica su dirección inicial, tamaño y estado, es decir, si la partición está o no asignada a un proceso actualmente. Un ejemplo de tabla de particiones es la que aparece en la figura 7.8.

### 7.6.4 Protección

En los sistemas de multiprogramación debemos proteger el código y los datos de cada proceso frente al acceso de los demás. La protección se puede implementar de varias formas, haciendo uso de dos registros. Una posibilidad es que uno de ellos mantenga la dirección base de la partición y el otro su longitud. La otra mantiene en cada registro los extremos superior e inferior de la partición. La

Partición	Inicio	Tamaño	Estado
0	0 KiB	512 KiB	ASIGNADA
1	512 KiB	256 KiB	ASIGNADA
2	768 KiB	768 KiB	LIBRE
3	1536 KiB	1024 KiB	ASIGNADA
4	2560 KiB	1536 KiB	LIBRE

Figura 7.8: Tabla de particiones para un sistema de particiones fijas

figura 7.9 muestra las dos posibilidades. Como se puede apreciar, el uso de un registro base y la longitud proporciona, además del esquema de protección, del soporte necesario para la traducción de direcciones.

Cada vez que un proceso desea acceder a una dirección de memoria se comprueba si está dentro de los límites permitidos para éste. Si es así, se le dejará acceder a ella, si no lo está se producirá un error.

### 7.6.5 Inconvenientes

El esquema de particiones fijas es muy simple, pero presenta algunos inconvenientes:

- El grado de multiprogramación del sistema es fijo y se establece en el momento de la generación del sistema cuando se crean las particiones de la memoria.
- Si un proceso es pequeño y no ocupa una partición completa se produce fragmentación interna.
- Dado que el tamaño de las particiones se establece en el momento de la generación del sistema, si el tamaño de un proceso es mayor al de cualquiera de las particiones definidas no se podrá ejecutar en ese sistema, a menos que se vuelva a redefinir el tamaño de éstas.

## 7.7 Multiprogramación con particiones variables

Los inconvenientes que presentaban los esquemas de particiones fijas hicieron que se desarrollara un sistema alternativo que se conoce como **particiones variables**.

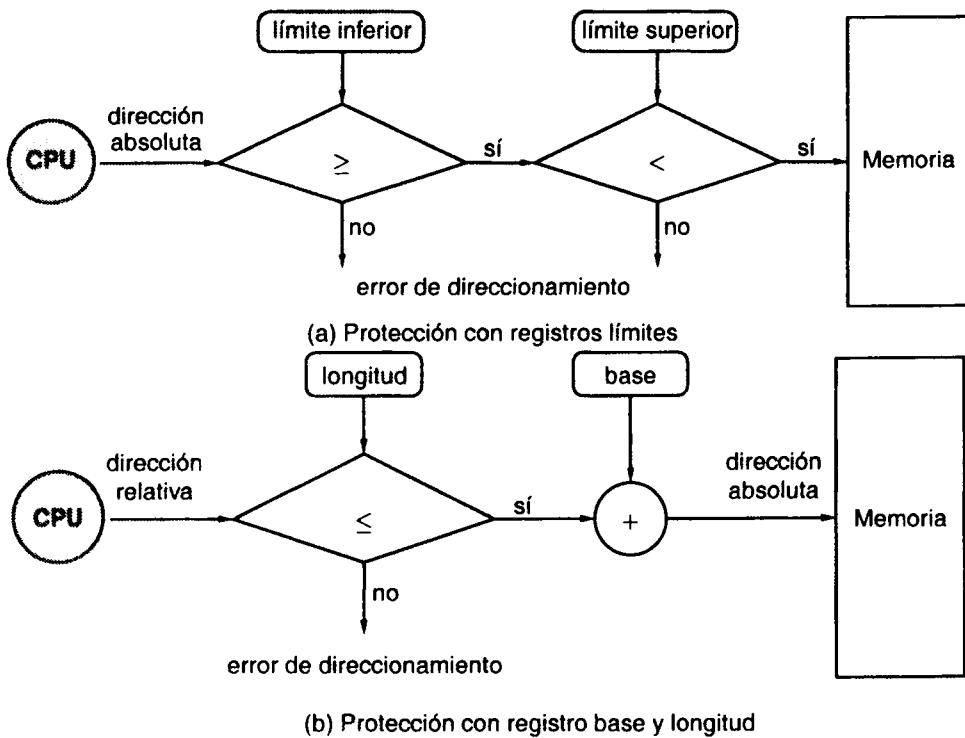


Figura 7.9: Esquemas de protección en sistemas con particiones fijas

**o dinámicas.** Un sistema operativo que usó esta técnica fue el sistema de IBM OS/MVT (*Multiprogramming with a Variable Number of Tasks*).

Cuando se emplean particiones variables, el número y tamaño de éstas varían a lo largo del tiempo, ya que cada proceso ocupa la cantidad de memoria que necesita. En la figura 7.10 puede verse cómo funciona. Inicialmente, sólo tenemos en memoria el sistema operativo y el proceso 1 (figura 7.10(a)). Posteriormente se van cargando los procesos 2, 3 y 4 (figuras 7.10(b), (c) y (d)). Esto deja un hueco que no es lo suficientemente grande como para albergar al proceso 5. En un instante dado, el sistema intercambia el proceso 1 (figura 7.10(e)), dejando así espacio suficiente para el proceso 5 (figura 7.10(f)). Al ser éste más pequeño que el hueco disponible, se genera otro hueco. Termina la ejecución del proceso 2 y abandona la memoria, fusionándose el espacio libre que deja con el hueco adyacente (figura 7.10(g)). Posteriormente se vuelve a cargar en memoria el proceso 1 que había sido intercambiado (figura 7.10(h)).

Como se puede observar en el ejemplo anterior el esquema de particiones va-

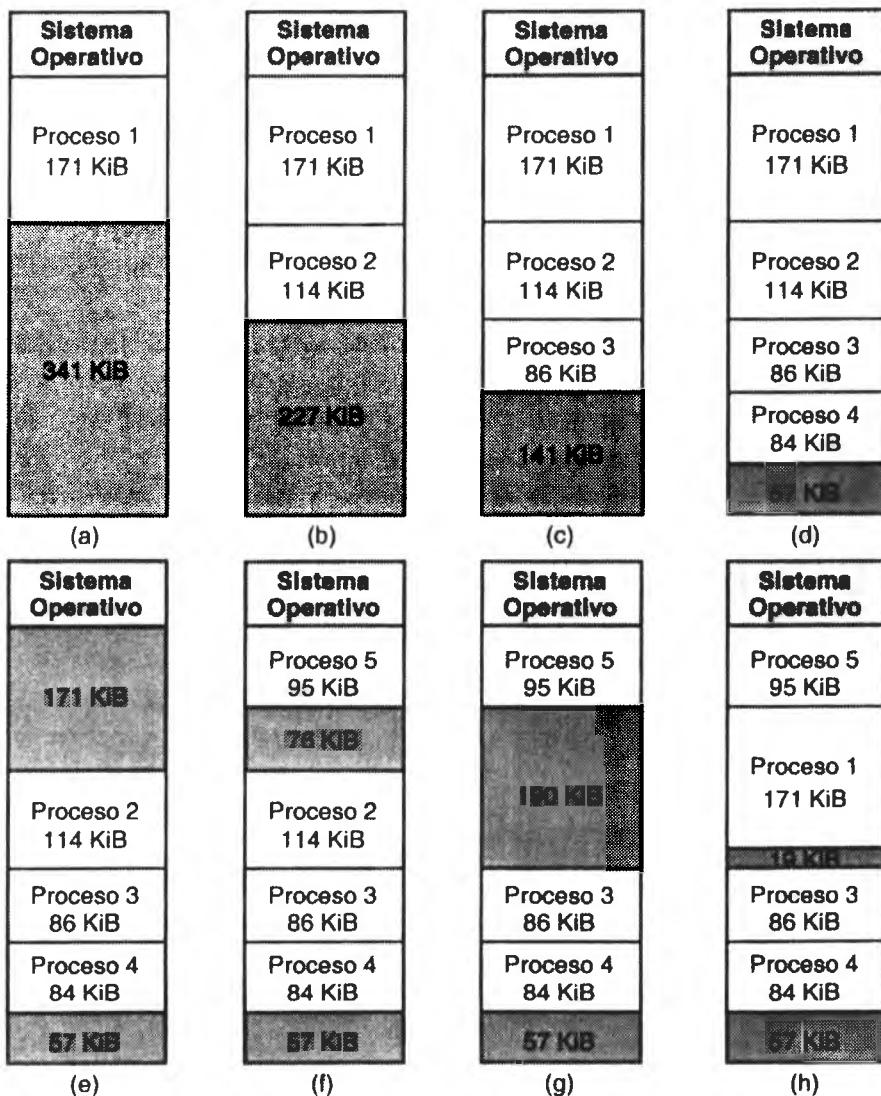


Figura 7.10: Asignación de la memoria con particiones variables

riables es más flexible que el de particiones fijas, puesto que el número, tamaño y posición de las particiones varían a lo largo del tiempo, adaptándose a las necesidades de los procesos. Esto permite mejorar la utilización de la memoria, al no existir fragmentación interna. Sin embargo, la gestión y control de la memoria son más complejas.

Aunque este esquema no presenta fragmentación interna, a lo largo del tiempo se van creando en la memoria huecos pequeños que no pueden alojar nuevos procesos. Esta situación se conoce como **fragmentación externa**.

### 7.7.1 Compactación

Una técnica que permite solucionar la fragmentación externa es la **compactación** o **recolección de basura**. Consiste en desplazar los procesos cargados en memoria de forma que todo el espacio libre quede formando un solo hueco. Si en la figura 7.10(h) hacemos compactación nos quedaría un hueco de 76 KiB. El problema que presenta la compactación es que consume tiempo de CPU. La figura 7.11 muestra de forma esquemática la memoria antes y después de realizarla.

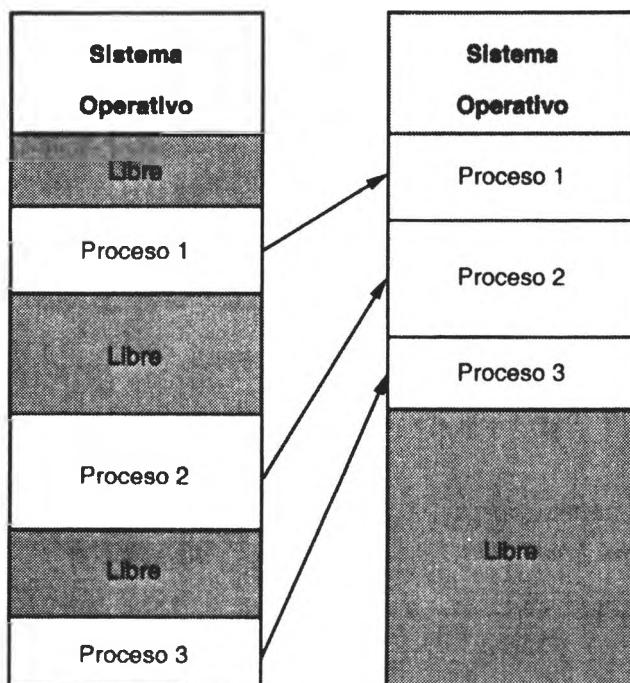


Figura 7.11: Compactación en un sistema con particiones variables

¿En qué momentos debería realizar la compactación el sistema operativo? Tenemos tres posibilidades. Podemos realizarla cuando se alcance un determinado nivel de ocupación de la memoria, por ejemplo, el 75%. El inconveniente que tiene es que podemos obligar al sistema a realizar un trabajo extra sin ser necesario porque no haya procesos esperando a entrar en memoria. Otra posibilidad es realizarla sólo cuando haya procesos esperando a entrar y los huecos no tengan el tamaño adecuado. En este caso, habrá que estar comprobando continuamente si hay trabajos en espera. La tercera posibilidad es hacerla cada cierto tiempo; la dificultad de ésta consiste en determinar el intervalo de tiempo adecuado. Si éste es pequeño se gastará mucho tiempo de CPU en hacer compactación; si se elige demasiado grande, podrán agruparse muchos trabajos a la espera y pueden perderse sus ventajas.

También hay que tener en cuenta que para poder realizar compactación se requiere que los procesos se puedan ejecutar en su nueva ubicación, por tanto, la traducción de direcciones deberá efectuarse en tiempo de ejecución para que los procesos sean reubicables (apartado 7.2).

### 7.7.2 Algoritmos de colocación

La decisión de dónde colocar un proceso en memoria es importante, puesto que como hemos dicho anteriormente la compactación de la memoria consume tiempo. Por tanto, habrá que utilizar alguna estrategia que determine en qué hueco hay que situar un proceso cuando se va a cargar en la memoria. Se podrían utilizar los mismos algoritmos que en el esquema de particiones fijas, el primer y el mejor ajuste (apartado 7.6.2), y otros tales como el **siguiente** y el **peor ajuste**.

El algoritmo del mejor ajuste elige el bloque de memoria que tiene un tamaño más parecido al del proceso. El primer ajuste empieza a rastrear la memoria desde el principio y elige el primer bloque disponible que sea lo suficientemente grande para alojarlo. El siguiente ajuste comienza la búsqueda desde la localización de la última asignación y elige el primer bloque disponible que tenga un tamaño adecuado. El peor ajuste hace lo contrario del primero, es decir, elige para cada proceso el hueco más grande posible.

La figura 7.12(a) muestra una instantánea del estado de la memoria en un sistema. Las zonas sombreadas son los huecos existentes en los que se indica su tamaño y las restantes están ocupadas por procesos. Asimismo se resalta el último proceso que ha sido cargado en memoria. La figura 7.12(b) muestra qué huecos se asignarían a un proceso de 14 KiB por parte de los diferentes algoritmos estudiados. El algoritmo del primer ajuste le asigna un hueco de 18 KiB generando un fragmento de 4 KiB. El mejor ajuste selecciona el hueco más parecido a su

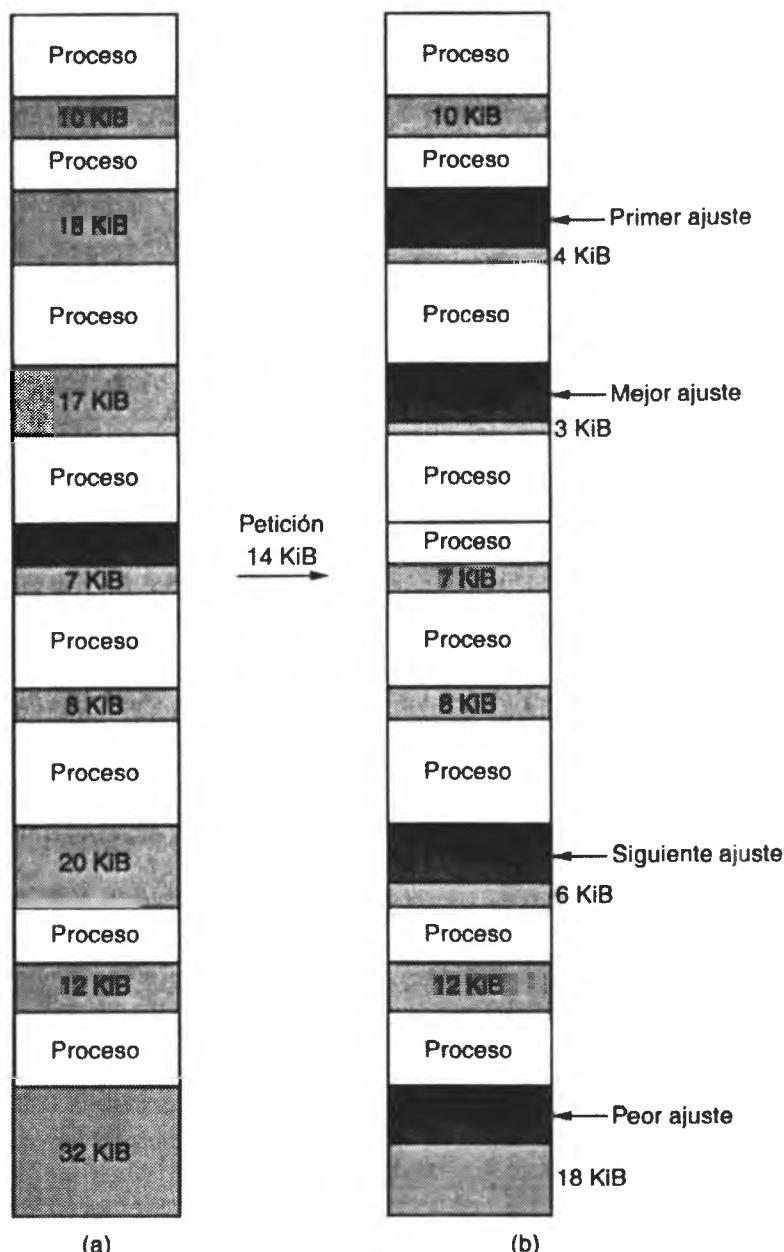


Figura 7.12: Ejemplo de uso de diversos algoritmos de colocación

tamaño (17 KiB), produciéndose el fragmento más pequeño (3 KiB). El peor ajuste, en contraposición al anterior, le asigna el hueco más grande (32 KiB), lo que origina la aparición de un hueco de 18 KiB. Finalmente, el siguiente ajuste escoge el primer hueco donde quepa el proceso partiendo de la posición de la última asignación, en este caso selecciona un hueco de 20 KiB, dando lugar a un fragmento de 6 KiB.

Estudios comparativos de los algoritmos [Wils95] han conducido a las siguientes conclusiones. El algoritmo del mejor ajuste es el que suele dar peores resultados. Esto es debido a que elige para cada petición el hueco que más se adapta, por lo que, si sobra memoria, se origina el hueco más pequeño posible. Éstos, por su pequeño tamaño, son difíciles de asignar, provocando una fragmentación rápida de la memoria y, por tanto, la necesidad de realizar compactación aumenta. A la vista de esto, podríamos pensar que el algoritmo del peor ajuste nos proporcionaría el mejor rendimiento al dejar el hueco más grande posible. Sin embargo, se ha comprobado que este algoritmo tiende a fragmentar huecos grandes, por lo que peticiones de cantidades grandes de memoria no pueden ser satisfechas sin recurrir a la compactación.

El algoritmo del primer ajuste es el más simple y el que mejor rendimiento proporciona. El del siguiente ajuste, aunque tiene un comportamiento similar al anterior, suele producir peores resultados, ya que con frecuencia, conduce a la asignación de un hueco del final de la memoria, que suele ser el más grande. Esto provoca una situación similar a la del peor ajuste.

### 7.7.3 Elementos de control

Vamos a considerar dos métodos que pueden utilizarse para llevar el control del uso de la memoria en sistemas de particiones variables.

#### 7.7.3.1 Mapas de bits

Consiste en disponer de un bit por cada unidad de asignación en la que se divide la memoria. De este modo, si el bit está a 0 indica que esa unidad está libre, y si se encuentra a 1 significa que está ocupada. La figura 7.13 representa una zona de memoria con cuatro procesos y su correspondiente mapa de bits.

Una cuestión importante que tiene que resolverse en el diseño de este esquema es precisamente el tamaño de la unidad de asignación. Ésta puede variar desde unos cuantos bytes hasta varios KiB. Para el establecimiento del tamaño adecuado se han de tener en cuenta las siguientes relaciones:

- El tamaño del mapa de bits es inversamente proporcional al tamaño de la unidad de asignación.
- La fragmentación interna producida cuando el tamaño del proceso no es un múltiplo entero de la unidad de asignación es proporcional al tamaño de ésta.

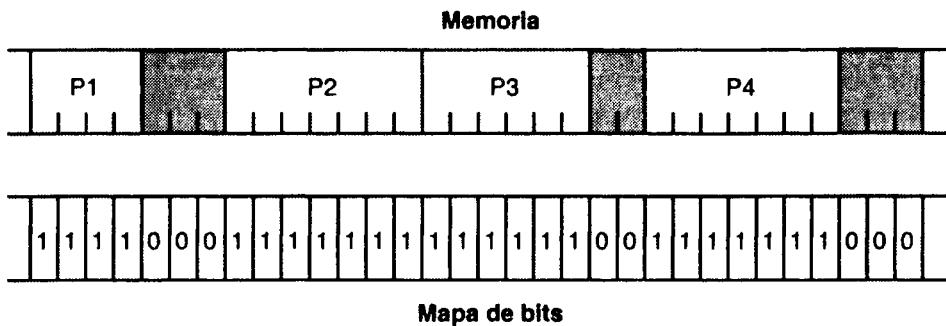


Figura 7.13: Mapa de bits de una zona de memoria con 4 procesos

Esta técnica es simple. Sin embargo, su principal inconveniente es la búsqueda de un hueco de memoria de una longitud determinada, adaptándose al criterio del algoritmo de colocación empleado.

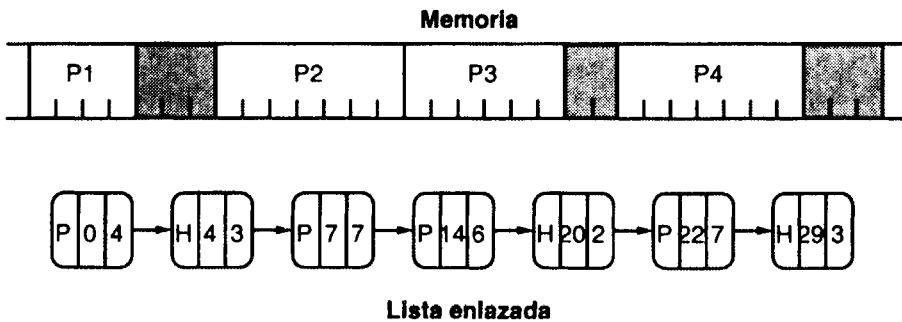


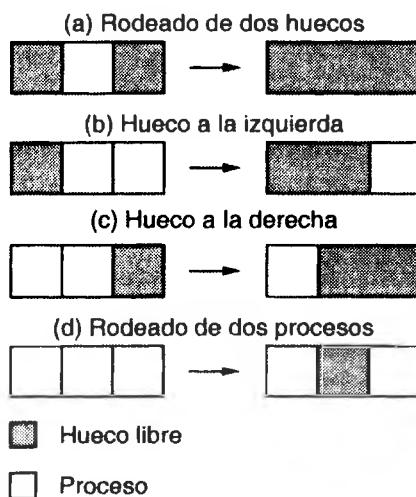
Figura 7.14: Lista enlazada de una zona de memoria con 4 procesos

### 7.7.3.2 Listas enlazadas

Otro método de control de la memoria es a través de una lista enlazada, donde cada elemento de la misma indica un proceso o un hueco existente en memoria. La

figura 7.14 muestra una zona de memoria con cuatro procesos y su correspondiente lista enlazada, donde cada elemento recoge la siguiente información: tipo de la zona de memoria (proceso o hueco), dirección de inicio, longitud y puntero al siguiente elemento.

Cuando un proceso finaliza su ejecución o se intercambia, el espacio de memoria que ocupa se libera, lo que implica una actualización de la lista. Esta operación no es tan simple como en el mapa de bits (cambiar 1 por 0), sino que depende de la situación del proceso en memoria. Así, cuando está rodeado de otros procesos, la liberación consiste únicamente en sustituir el tipo de zona de proceso a hueco. Sin embargo, cuando dispone de huecos adyacentes, es necesario fusionarlos, con objeto de formar uno de mayor tamaño. Para facilitar la realización de esta operación de fusiónado de huecos, la lista enlazada se suele implementar como una lista doblemente enlazada ordenada por direcciones. La figura 7.15 muestra las distintas posibilidades que se pueden plantear.



**Figura 7.15:** Combinaciones posibles cuando un proceso sale de la memoria

La ventaja que tienen las listas enlazadas es que su implementación se puede adaptar al algoritmo de colocación empleado. Una posibilidad consiste en utilizar listas separadas para llevar el control de los procesos en memoria y de los huecos. De este modo, cada lista se puede ordenar según el criterio más adecuado consiguiendo optimizar la operación de búsqueda de un hueco. Así, si se emplea el algoritmo del mejor ajuste, la lista de huecos podría mantenerse ordenada por tamaños crecientes; si se emplea el peor ajuste el orden más adecuado de los huecos sería de mayor a menor tamaño. En los casos del primer y siguiente ajuste las

listas se ordenarían por direcciones.

Cuando los huecos se conservan en una lista separada es posible añadir una optimización adicional. En vez de tener un conjunto aparte de estructuras de datos para conservar la lista de huecos, se pueden utilizar éstos para mantenerla. Así sólo se necesitaría un puntero al primer hueco.

## 7.8 El sistema compañero

Los sistemas de multiprogramación con particiones presentan varios inconvenientes. Por un lado, en un esquema de particiones fijas se limita el grado de multiprogramación y se desperdicia espacio mediante la fragmentación interna. Por otro, el esquema de particiones variables produce sobrecarga en el sistema con la realización de la compactación para resolver la fragmentación externa, y su mantenimiento es más complejo. Un compromiso entre ambos esquemas es el **sistema compañero**.

En este sistema, no se establece un número fijo de particiones, pero se limita el tamaño de los bloques de memoria que pueden solicitar los procesos. Sólo es posible asignar bloques de tamaño  $2^k$ , donde  $I \leq k \leq S$ , siendo  $2^I$  el más pequeño que se puede asignar, y  $2^S$  el más grande, que coincide con la memoria del sistema.

Inicialmente, la memoria se considera como un bloque libre único de tamaño  $2^S$ . Para cada potencia de 2 entre  $2^I$  y  $2^S$  se mantiene una lista de bloques libres de dicho tamaño, que inicialmente están vacías, excepto la lista de los bloques de tamaño  $2^S$ . La lista de bloques libres de tamaño  $2^i$  recibe el nombre de **lista *i***.

Cuando se produce una petición de tamaño  $b$ , se busca un bloque de tamaño  $2^i$ , donde  $i$  es el entero más pequeño tal que  $2^{i-1} < b \leq 2^i$ . Si no existe un bloque de tamaño  $2^i$  con esas características, se escoge uno de tamaño  $2^{i+1}$  para dividirlo en dos bloques iguales de tamaño  $2^i$ , uno de los cuales se asigna y el otro permanece libre situándose en la lista  $i$ . En el caso de no existir bloques de tamaño  $2^{i+1}$  libres, se divide uno de tamaño  $2^{i+2}$  en dos de  $2^{i+1}$ , uno queda libre, y el otro se divide en dos bloques de tamaño  $2^i$ . En caso de no existir bloques libres de tamaño  $2^{i+2}$ , el proceso continúa hasta encontrar un bloque disponible. Si no existe bloque alguno, no se puede asignar la memoria.

En este esquema, si un bloque de tamaño  $2^{i+1}$  empieza en la posición  $p$ , los dos bloques resultantes de su división empiezan en las posiciones  $p$  y  $p + 2^i$ , respectivamente. Se dice que ambos bloques son **compañeros**. Cuando se libera un bloque de tamaño  $2^i$  y está libre su compañero, ambos se combinan en uno de tamaño  $2^{i+1}$ .

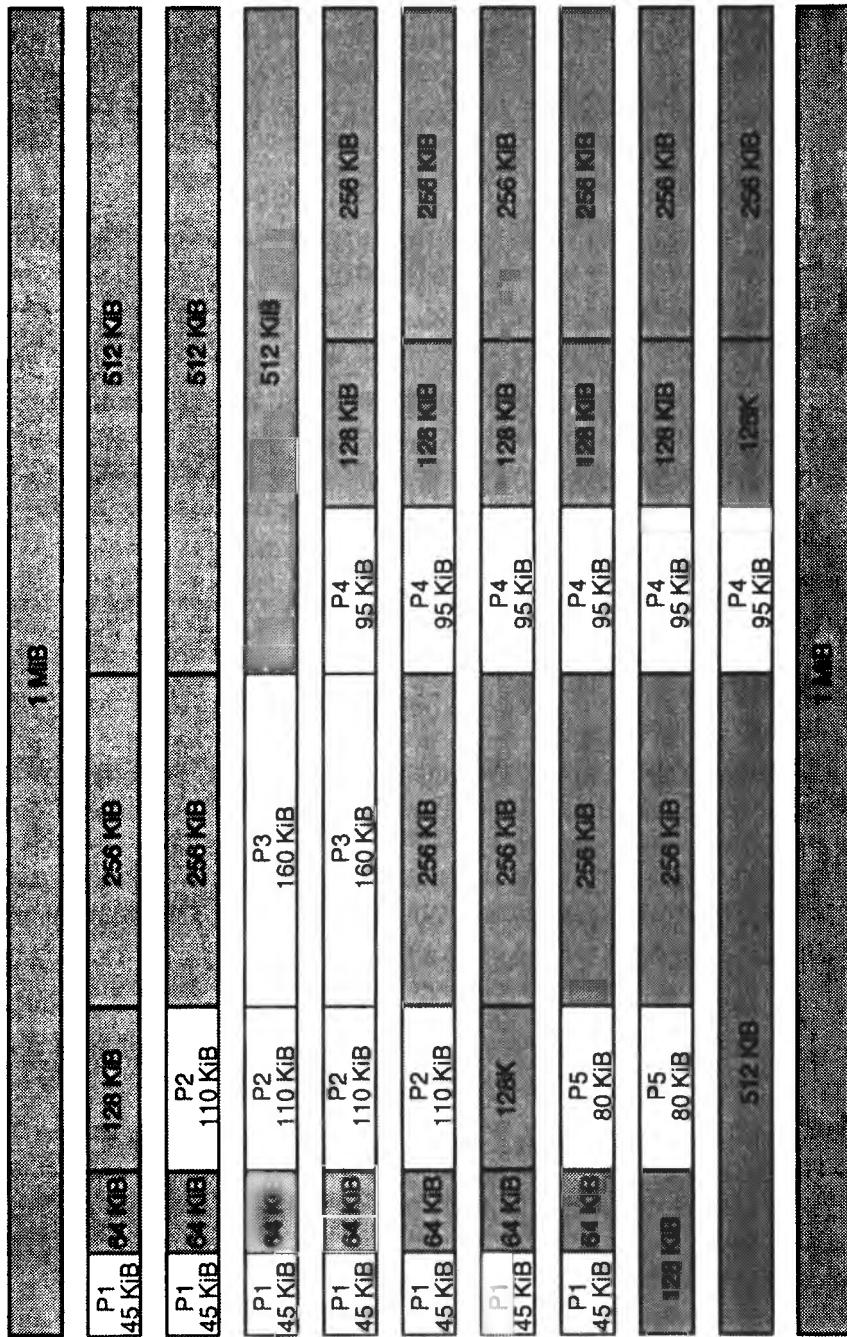


Figura 7.16: Sistema compañero

La figura 7.16 muestra un ejemplo del funcionamiento de este esquema en un sistema con una memoria de 1 MiB. Inicialmente se produce una petición para un proceso P1 de 45 KiB. El bloque inicial de la memoria se divide en dos compañeros de 512 KiB, el primero se divide a su vez en dos bloques de 256 KiB, que se subdividen en dos de 128 KiB, para finalmente dividirlos en otros dos de 64 KiB, uno de los cuales se asigna al proceso. A continuación se produce una petición de 110 KiB, asignándose el bloque de 128 KiB que se creó anteriormente. Este proceso de división de bloques continúa mientras sea necesario.

Cuando empiezan a producirse liberaciones de memoria, se van fusionando los huecos. Así, cuando el proceso P5 finaliza libera un bloque de 128 KiB, que tiene a su compañero libre, fusionándose ambos en un bloque de 256 KiB. Éste tiene su compañero libre, por lo que pueden fusionarse en un único bloque de 512 KiB.

Este sistema presenta un esquema de asignación de memoria dinámico con un grado de multiprogramación flexible, al igual que el esquema de particiones variables. Con el fin de simplificar la gestión del espacio libre y asignado, la memoria se asigna en bloques de ciertos tamaños, asemejándose al esquema de particiones fijas, presentando también fragmentación interna.

## 7.9 Paginación

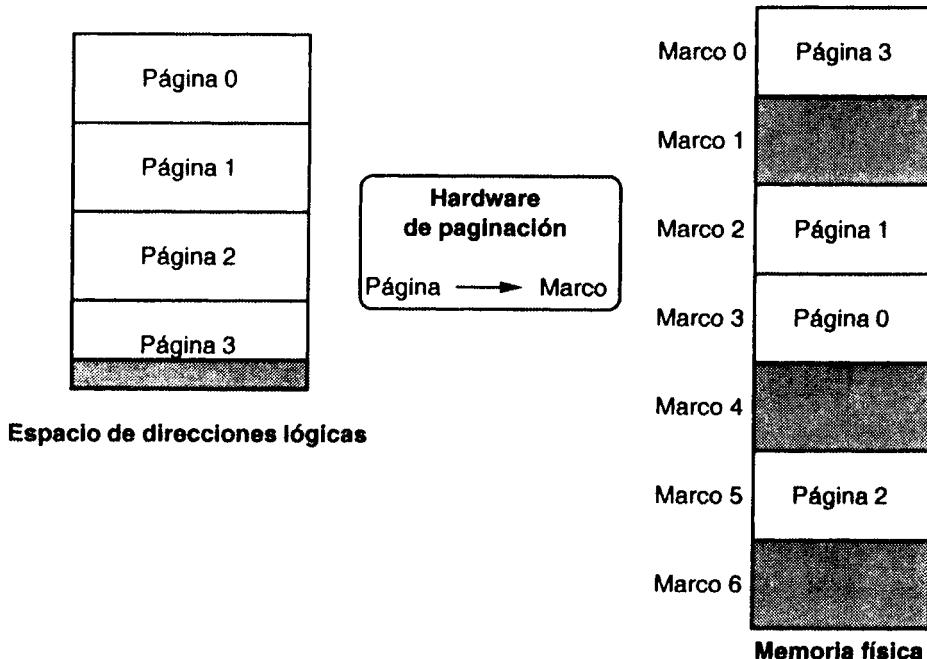
Los esquemas de multiprogramación vistos hasta ahora presentan problemas de fragmentación externa e interna. La primera se produce cuando la memoria disponible no está contigua, sino fragmentada en muchos bloques dispersos. Éstos no pueden ser aprovechados para cargar un proceso, puesto que la memoria que se asigna debe ser contigua. La fragmentación interna se origina cuando el tamaño del bloque de memoria asignado no se ajusta al del proceso.

Una estrategia alternativa que soluciona el problema de la fragmentación externa y minimiza la interna consiste en permitir que la memoria asignada a un proceso no sea contigua. Una forma de llevar a cabo esta solución es la paginación.

La paginación considera la memoria física dividida en bloques llamados **marcos**, todos del mismo tamaño. La memoria lógica también se considera dividida en bloques del mismo tamaño que los marcos, denominados **páginas**. Al cargar un proceso en memoria, sus páginas pueden ser cargadas en cualquier marco disponible, no siendo necesario que éstos estén contiguos.

La paginación, a diferencia de los esquemas anteriores, introduce una divergencia entre la visión que los usuarios tienen de los procesos en memoria y su situación real. Los usuarios consideran que sus procesos ocupan zonas contiguas

de memoria, pero si se usa un esquema de paginación, los procesos estarán dispersos en ella. La traducción de direcciones lógicas a físicas es la que hace concordar estas dos visiones. Este proceso se lleva a cabo en tiempo de ejecución y para no degradar el rendimiento del sistema se realiza con ayuda de un hardware adecuado. La figura 7.17 representa la memoria lógica y física en el esquema de paginación.



**Figura 7.17:** Memoria lógica y física en el esquema de paginación

A continuación veremos los pasos que debe dar el administrador de la memoria para cargar un proceso en un sistema de paginación (figura 7.18):

1. Se calcula el número de páginas que ocupará el proceso. Para ello se divide el espacio de direcciones lógicas de éste por el tamaño de una página. Si el número obtenido no es entero se redondea por exceso, ya que siempre hay que asignar un número entero de marcos a un proceso. Esto puede provocar que el último marco presente fragmentación interna.
2. Como cada página necesita un marco de memoria física, habrá que comprobar si están disponibles. Dado que las páginas no necesitan estar contiguas, los marcos pueden estar en cualquier lugar. El sistema operativo mantiene

ne la información sobre qué marcos están libres y cuáles ocupados en una estructura de datos llamada **tabla de marcos** (apartado 7.9.1).

3. Si hay memoria libre suficiente, cada página se va cargando en un marco. Para saber posteriormente donde está cargada cada página de un proceso, el sistema mantiene para cada uno de ellos una estructura conocida como **tabla de páginas** (apartado 7.9.2).

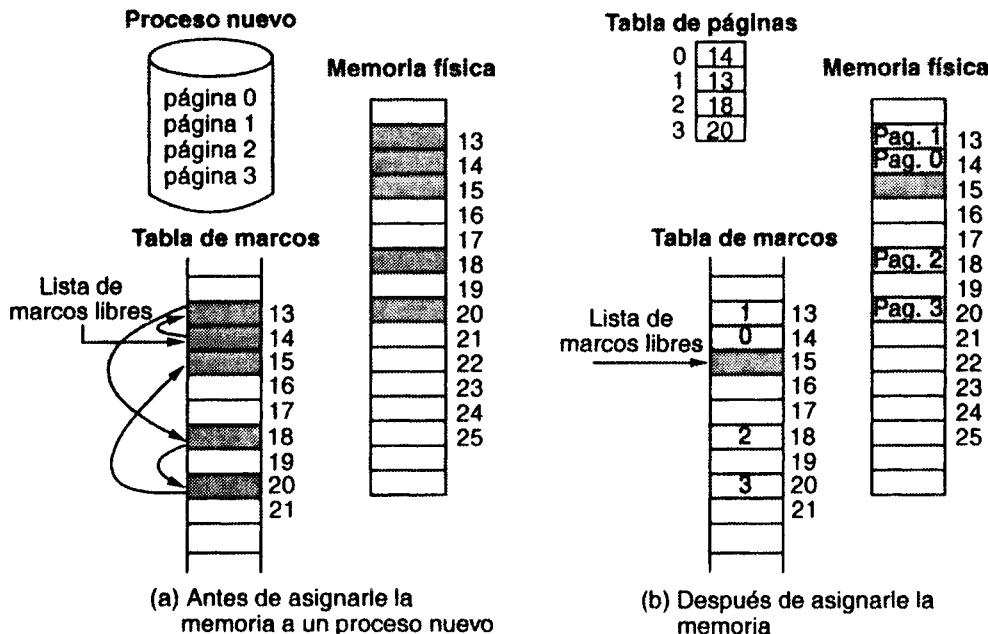


Figura 7.18: Carga de un proceso en un sistema de paginación

### 7.9.1 Tabla de marcos

Para asignar memoria a un proceso, el sistema operativo debe tener información sobre el número total de marcos y su estado (libre u ocupado). Ésta se mantiene en una estructura denominada **tabla de marcos**, que posee una entrada por cada marco de memoria física. El tamaño de la tabla de marcos es fijo y será igual al tamaño de la memoria principal dividido por el tamaño de un marco. Así,

$$TM = \frac{MF}{M}$$

donde  $TM$  es el número de entradas de la tabla de marcos,  $MF$  es el tamaño de la memoria física, y  $M$  es el tamaño del marco, que es igual al de la página.

Cada entrada de la tabla de marcos mantiene información del estado del marco, indicando si está libre o asignado y, si está asignado, a qué proceso y qué página de éste lo ocupa.

### 7.9.2 Tabla de páginas

En los sistemas de paginación existe una tabla de páginas por cada proceso activo, creándose durante la carga de éste en memoria. La tabla de páginas tiene tantas entradas como páginas pueda ocupar el proceso; cada una indica el número de marco donde está alojada.

La tabla de páginas es un elemento vital para la traducción de direcciones en un sistema de paginación ya que establece la correspondencia entre la dirección lógica y la física. Este proceso se realiza en tiempo de ejecución, por lo que cada acceso a memoria supone acceder primero a la tabla de páginas y posteriormente a la dirección pedida.

Para disminuir el tiempo medio de acceso a memoria la tabla de página se ha implementado de diversas formas, en registros, o utilizando *buffers de traducción adelantada* (TLB, *Translation Lookaside Buffer*).

El uso de registros para mantener la tabla de páginas hace que el acceso a ésta sea muy rápido, pero tiene el inconveniente de que sólo puede usarse cuando el número de entradas de ésta es pequeño, de ahí que no se utilice en la actualidad.

Otra forma posible de acelerar el acceso a la tabla de páginas es mantenerla en memoria física y disponer de algunas de sus entradas en una TLB. Se trata de una memoria caché de traducciones que contiene aquellas entradas de la tabla de páginas usadas más recientemente. Este esquema se utiliza en los sistemas actuales.

### 7.9.3 Traducción de direcciones

En un sistema paginado las direcciones lógicas tienen dos componentes, el número de página,  $p$ , y el desplazamiento dentro de ésta,  $d$ . El número de página se usa como índice para acceder a la tabla de páginas, que contiene el número de marco en el que reside la página. Este número de marco se combina con el desplazamiento para obtener la dirección física. La figura 7.19 representa este proceso de traducción. En ella aparece un registro hardware que apunta a la base

de la tabla de páginas del proceso (PTBR, *Page-Table Base Register*) que está en ejecución. Cuando se produce un cambio de proceso es necesario cambiar el contenido de este registro para disponer de la nueva tabla de páginas.

El tamaño de las páginas y los marcos se establece en función del hardware disponible. Con objeto de facilitar el proceso de traducción se suele seleccionar una potencia de 2 para éste. Así, si el tamaño de una página es  $2^s$ , el número de bits necesarios para direccionar una unidad de almacenamiento dentro de ésta es  $s$ . Por otro lado, si en nuestra máquina las direcciones lógicas son de  $t$  bits (el tamaño del espacio de direcciones lógicas es  $2^t$ ) y el tamaño de la página es  $2^s$ , entonces los primeros  $t - s$  bits designan el número de página, y los restantes  $s$  bits el desplazamiento dentro de ésta.

De forma análoga, la dirección física tiene la forma  $(m, d)$ , donde  $m$  es el número de marco y  $d$ , el desplazamiento dentro de él. El número de bits de cada componente se determina a partir del tamaño del marco y del número de bits de la dirección física en nuestra máquina.

En el ejemplo que se muestra en la figura 7.19 partimos de una dirección lógica de 16 bits y un tamaño de página de 1 KiB ( $2^{10}$ ). Por tanto, los 6 primeros bits constituyen el número de página y los 10 bits restantes, el desplazamiento. La dirección física del sistema es de 16 bits, por lo que presenta la misma estructura que la lógica. Supongamos la dirección lógica 0000110101111010, que corresponde a la página número 3 (000011) y a un desplazamiento de 378 (0101111010). Si ésta reside en el marco 2, la dirección física 0000100101111010 corresponde al marco número 2 (000010) y desplazamiento 378 (0101111010).

Si se dispone de TLB el proceso de traducción de direcciones se modifica tal como aparece en la figura 7.20. Cuando se va a traducir una dirección lógica a física, se comprueba si la entrada de la tabla de páginas correspondiente se encuentra en la TLB. Si es así, no es necesario acceder a la tabla de páginas; en caso contrario habrá que acceder a ella.

Puesto que la TLB sólo contiene algunas de las entradas de la tabla de páginas, no se puede indexar por el número de página, sino que cada entrada deberá contener el número de página, además de la entrada de la tabla de páginas completa. Cuando se accede a la TLB se consultan todas sus entradas simultáneamente, acelerándose el proceso de búsqueda.

#### 7.9.4 Protección

En los sistemas paginados la protección comprende dos aspectos, la comprobación de la validez de las direcciones y el modo de acceso a las páginas.

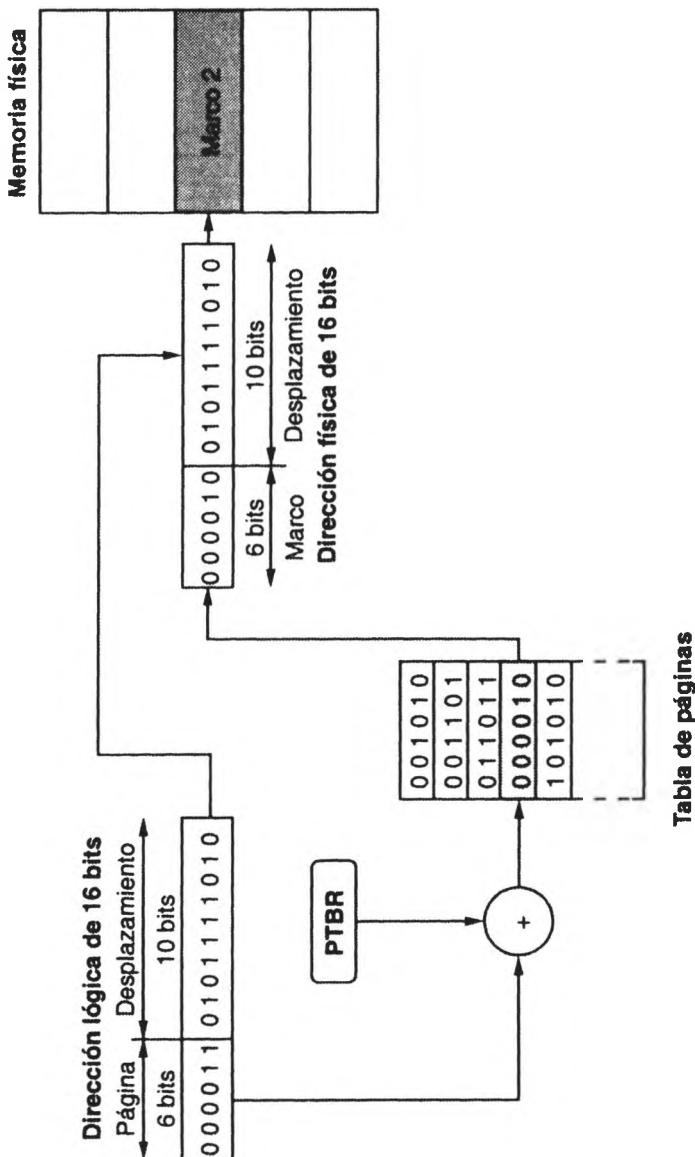
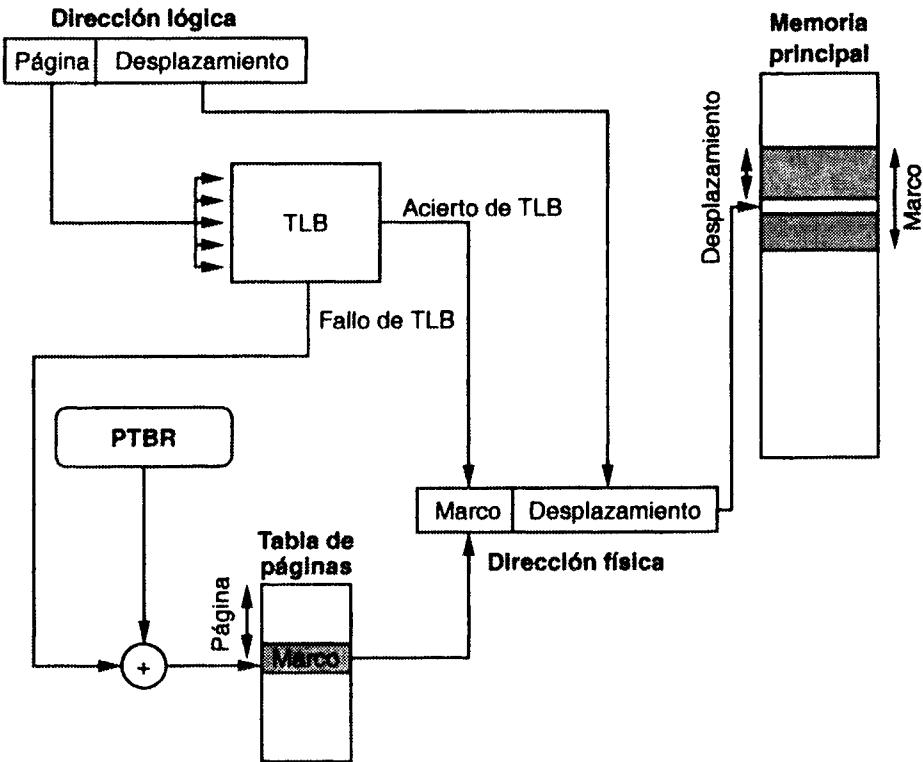


Figura 7.19: Traducción de direcciones en un sistema de paginación



**Figura 7.20:** Traducción de direcciones en paginación con TLB

El primer aspecto hace referencia a la comprobación que hace el sistema para evitar que un proceso acceda al espacio de direcciones de otro. Para ello, se añade un bit de validez a cada entrada de la tabla de páginas que indicará si la página pertenece o no al proceso. Otra posibilidad es utilizar un registro de longitud de la tabla de páginas que contiene el número de páginas del proceso. Cualquier dirección lógica generada se comprueba con este valor para saber si está dentro del rango permitido.

Dependiendo del contenido de las páginas se podrá acceder a ellas de un modo u otro. Para controlar esto se asocian unos bits de acceso a cada entrada de la tabla de páginas, sólo lectura o lectura-escritura. De este modo, el sistema podrá comprobar fácilmente si el proceso está intentando acceder a la página de forma adecuada.

El programa 7.1 muestra el proceso de traducción de direcciones junto con las comprobaciones de validez y modo de acceso. En primer lugar se extrae el número de página a partir de los  $t - s$  bits más a la izquierda de la dirección

lógica. Esto no es más que una división de la dirección lógica por el tamaño de la página. A continuación se comprueba si la página es válida y se poseen los permisos adecuados para realizar la operación deseada. Para ello, el número de página se emplea como índice en la tabla de páginas del proceso (apuntada por el registro PTBR). Finalmente, si todo es correcto se genera la dirección física, que se obtiene de multiplicar el número de marco por el tamaño de la página ( $m \times 2^s$ ), y añadirle el desplazamiento. Esta dirección no necesita ser calculada, se construye fácilmente concatenando el número de marco al desplazamiento.

**Programa 7.1****Traducción de direcciones en un sistema paginado**

```
#include "memoria.h"

direccion_f traducir (direccion_l dir_logica, acceso_t modo)
{
    int pagina, desplazamiento;
    div_t division;
    direccion_f dir_fisica;

    division = div (dir_logica, TAM_PAGINA);

    pagina = division.quot;
    desplazamiento = division.rem;

    if ( ! PTBR[pagina].valida )
        fallo_validez(pagina)
    else if ( comprueba_permisos(PTBR[pagina].permisos, modo) )
        fallo_proteccion(pagina);
    else
        dir_fisica = PTBR[pagina].marco * TAM_PAGINA + desplazamiento;

    return dir_fisica;
}
```

### 7.9.5 Páginas compartidas

Si varios procesos ejecutan el mismo código, sería conveniente que todos ellos pudieran compartir los marcos de memoria donde reside éste, ya que esto supondría un ahorro considerable de memoria. Los sistemas de paginación permiten hacer esto siempre, claro está, que el código sea reentrante, es decir, no automodificable (el código no cambia durante su ejecución).

Así, si varios procesos comparten el mismo código, éste estará cargado en una serie de marcos compartidos, mientras que los datos de cada proceso estarán en

sus propios marcos. Si esto es así, habrá varias entradas de distintas tablas de páginas que apuntarán a los mismos marcos. La tabla de marcos debe contemplar también esta compartición añadiendo un nuevo campo a cada entrada que sea un contador del número de procesos que están compartiéndolo. Esto es necesario porque cuando uno de los procesos que está compartiendo ese marco termine, se debe comprobar el contador para saber si todavía hay procesos que lo estén usando. De este modo, cada proceso al terminar decrementa el contador en una unidad, y cuando llegue a 0 es cuando el marco queda libre.

### 7.9.6 Fragmentación

Cuando se usa un esquema de paginación no aparece fragmentación externa, ya que cualquier marco puede ser asignado a un proceso que lo necesite. Sin embargo, sí se puede dar fragmentación interna. Si los requisitos de memoria de un proceso no son un múltiplo entero del tamaño de una página, dado que los marcos son asignados como unidades, el último marco asignado no se aprovechará completo.

Si el tamaño del proceso es independiente del tamaño de la página, podemos esperar una fragmentación interna media de  $2^{s-1}$ , es decir, de media página por proceso.

### 7.9.7 Tamaño de las páginas

El tamaño de las páginas viene determinado por la arquitectura del sistema de computación. En algunos casos, el diseñador del sistema operativo puede decidir entre un conjunto de valores. En estos casos, la decisión a tomar debe tener en cuenta los siguientes factores, la fragmentación interna y el tamaño de la tabla de páginas.

Dado que la fragmentación interna media por proceso es la mitad de una página, sería deseable tener páginas pequeñas. Por otro lado, si se reduce el tamaño de la página, crece el número de éstas, y, por tanto, el de la tabla de páginas. En función de este factor, sería deseable tener páginas grandes.

Se puede establecer una relación para determinar el valor óptimo del tamaño de la página. Sea  $S$  el tamaño promedio de los procesos,  $P$  el tamaño de la página y  $E$  el número de bytes por entrada en la tabla de páginas. El número de páginas medio por proceso viene dado por la relación  $\frac{S}{P}$ , y su tabla de páginas requiere un espacio de  $E \times \frac{S}{P}$ . Al existir fragmentación interna en la última página, el coste global del sistema paginado con una página de tamaño  $P$ , viene dado por la siguiente ecuación:

$$\left(\frac{S}{P}\right) \times E + \frac{P}{2}$$

donde el óptimo se alcanza en

$$P = \sqrt{2 \times S \times E}$$

## 7.10 Segmentación

Otro esquema de administración de la memoria en el que no se exige que el proceso esté cargado en una zona contigua es la segmentación. En este caso el programa y los datos se dividen en **segmentos**, que pueden tener tamaños diferentes. Esta división de los procesos en segmentos se corresponde mejor que la anterior con la visión que tienen los usuarios de ellos. En la figura 7.21 se puede observar la visión que tiene el usuario de la imagen de un proceso. En ella los ficheros fuente que constituyen el programa se transforman en un espacio de direcciones lógicas constituido por diversos segmentos. En un sistema de segmentación, éstos se cargan en memoria en los huecos disponibles del tamaño adecuado y se crea una **tabla de segmentos** para su gestión.

Al igual que en el esquema de paginación, una dirección lógica consta de dos partes, un número de segmento y un desplazamiento. Las direcciones físicas se obtienen en tiempo de ejecución, para ello se hace uso de la tabla de segmentos.

Además de esta estructura, se necesita llevar el control de qué zonas de la memoria están libres y cuáles ocupadas. Esto se puede hacer mediante listas enlazadas o mapas de bits, al igual que en los sistemas de particiones variables (ver apartado 7.7.3).

### 7.10.1 Tabla de segmentos

La segmentación emplea para la traducción de direcciones la tabla de segmentos. Cada proceso tiene su propia tabla, que contiene una entrada por cada segmento. Éstas almacenan la dirección de comienzo del segmento en memoria física, así como su longitud, que se emplea para verificar que el proceso utilice direcciones válidas.

La tabla de segmentos puede contener además los bits de protección correspondientes, para controlar el modo de acceso al segmento.

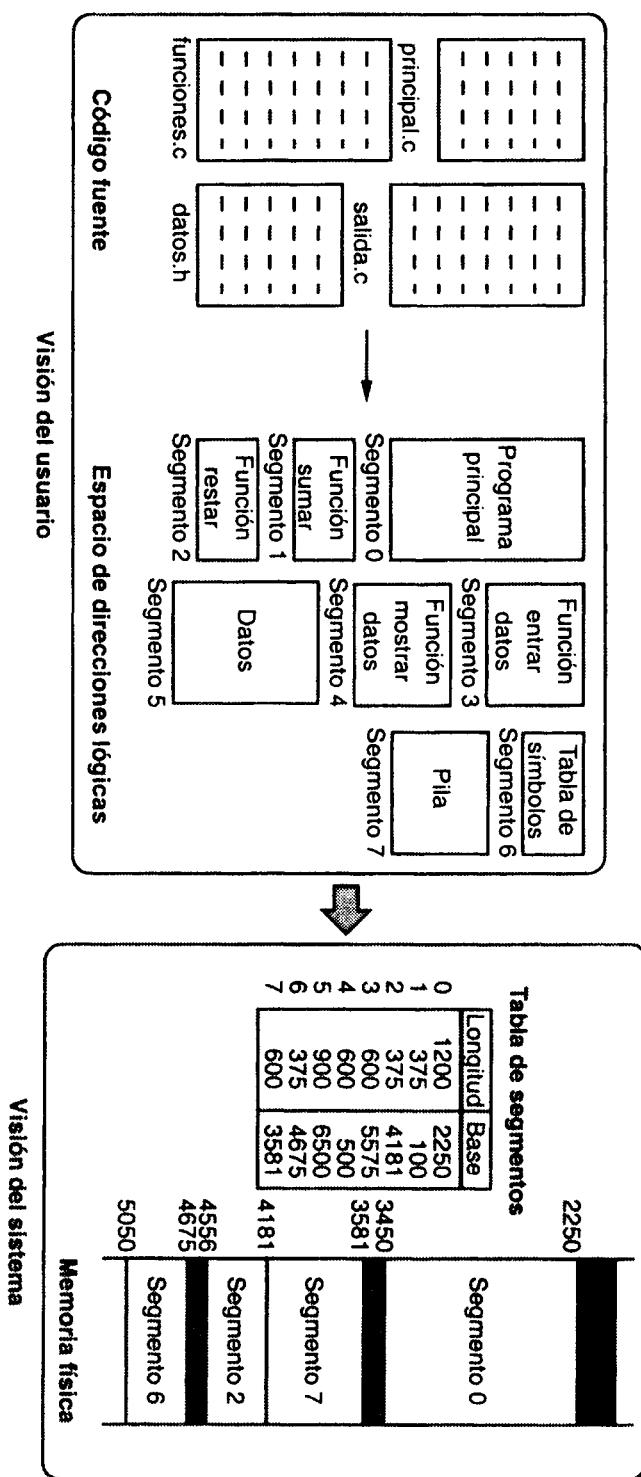


Figura 7.21: Esquema del sistema segmentado

Como ocurre con la paginación, si la tabla de segmentos se mantiene en memoria, se requieren dos accesos a ésta para acceder a cada dirección, uno a la tabla y otro a la dirección deseada. Con objeto de reducir este doble acceso se suele emplear un conjunto de registros asociativos para guardar las entradas de la tabla de segmentos que se han usado más recientemente, de forma similar a la TLB.

### 7.10.2 Traducción de direcciones

Las direcciones lógicas en un sistema de segmentación vienen dadas por dos componentes: el número de segmento,  $s$ , y el desplazamiento,  $d$ . El primero se utiliza como índice para consultar la tabla de segmentos que almacena la dirección base de éste, que se combina con el desplazamiento para obtener la dirección física. Al generarse las direcciones físicas en tiempo de ejecución, los procesos son reubicables. La figura 7.22 representa este proceso de traducción. En ella aparece el registro base de la tabla de segmentos (STBR, *Segment-Table Base Register*), y el registro de longitud de la tabla de segmentos (STLR, *Segment-Table Length Register*). El primero apunta a la tabla de segmentos del proceso en ejecución, y el segundo mantiene el número de segmentos de éste. Cuando se produce un cambio de proceso es necesario actualizarlos.

En los sistemas segmentados, como el tamaño de los segmentos es variable, se establece un tamaño máximo de éstos que determina el formato de la dirección lógica. En el ejemplo de la figura 7.22, la dirección lógica es de 16 bits, y se considera un tamaño máximo de segmento de 4 KiB; de este modo, los doce últimos bits indican el desplazamiento y los cuatro primeros el número de segmento. Así, la dirección lógica 0010001011100000 corresponde al segmento 2 y desplazamiento 736. Mediante la tabla de segmentos obtenemos la dirección de comienzo de éste en memoria física (0001000000010000), siendo la dirección física requerida la suma de ésta y el desplazamiento, 000100101110000.

### 7.10.3 Compartición y protección

Una ventaja de la segmentación es la posibilidad de compartir segmentos. Cuando dos o más procesos comparten segmentos, varias entradas de las tablas de segmentos de los procesos apuntarán a la misma dirección física. Para conseguir esta compartición, la tabla de segmentos debe tener información de cuántos procesos comparten cada segmento mediante un contador. Cuando los procesos van finalizando y liberando sus segmentos, los contadores correspondientes disminuyen en una unidad, y cuando llegan a cero se libera el segmento.

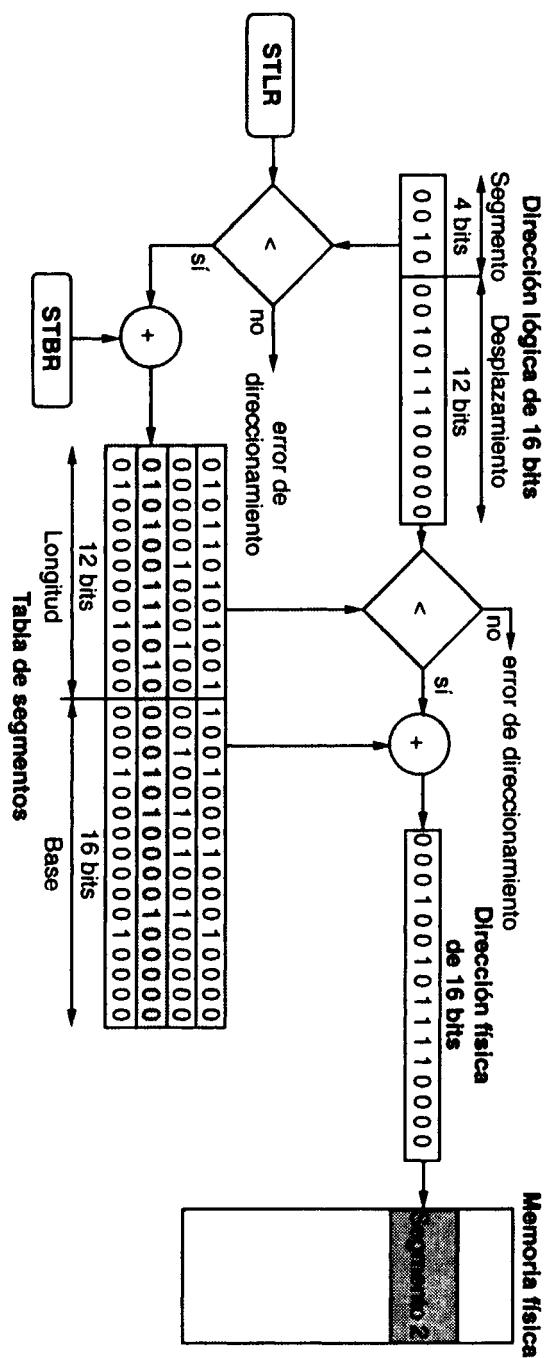


Figura 7.22: Traducción de direcciones en un sistema de segmentación

Aunque la compartición parece sencilla, implica una serie de consideraciones. Los segmentos de código suelen contener referencias a sí mismos (por ejemplo, las funciones recursivas), que consisten en un número de segmento y un desplazamiento. Por tanto, todos los procesos que comparten el segmento deberán asignarle el mismo número. Los segmentos de datos de sólo lectura, así como los de código que no se refieran a sí mismos pueden tener distintos números.

La segmentación permite realizar la protección de los segmentos fácilmente. Dado que los segmentos son porciones bien definidas de los programas, es normal encontrarse con segmentos de datos y segmentos de código. Puesto que las instrucciones no se pueden modificar a sí mismas, los segmentos de código suelen ser de sólo lectura. Por contra, los segmentos de datos pueden ser de lectura-escritura. Para conseguir esta protección se incorporan los bits correspondientes en la tabla de segmentos.

---

**Programa 7.2****Traducción de direcciones en un sistema segmentado**

```
#include "memoria.h"

direccion_f traducir (direccion_l dir_logica, acceso_t modo)
{
    int segmento, desplazamiento;
    div_t division;
    direccion_f dir_fisica;

    division = div (dir_logica, TAM_SEGMENTO_MAXIMO);

    segmento = division.quot;
    desplazamiento = division.rem;

    if ( segmento > STLR)
        fallo_validez(segmento)
    else if ( desplazamiento > STBR[segmento].longitud )
        fallo_direccionamiento(segmento, desplazamiento)
    else if ( comprueba_permisos(STBR[segmento].permisos, modo) )
        fallo_proteccion(segmento);
    else
        dir_fisica = STBR[segmento].base + desplazamiento;

    return dir_fisica;
}
```

---

El programa 7.2 muestra la traducción de direcciones en un sistema de segmentación junto con las comprobaciones de validez y modo de acceso. En primer lugar, se obtiene el número de segmento dividiendo la dirección lógica por el tamaño de segmento máximo posible. A continuación se comprueba si la dirección

generada es válida. Para ello se compara el número de segmento con el registro STLR y el desplazamiento con la longitud del segmento. Si la dirección es válida se pasa a comprobar la corrección del modo de acceso, mediante los bits de protección que se mantienen en la tabla de segmentos. Si todo es correcto, la dirección física de la unidad de direccionamiento referenciada es la suma de la dirección de comienzo del segmento más el desplazamiento.

#### 7.10.4 Fragmentación y algoritmos de colocación

Un sistema de segmentación no presenta fragmentación interna, ya que cada segmento se aloja en una zona de memoria de su misma longitud. Sin embargo, sí puede presentar fragmentación externa si todos los bloques de memoria libre son demasiado pequeños para dar entrada a un nuevo segmento. Para reducir esta fragmentación se puede realizar compactación, al igual que en el sistema de particiones variables, pero con más facilidad al disponer de procesos reubicables.

Un aspecto relacionado con la fragmentación externa es la colocación de los segmentos en memoria. Dado que podemos encontrarnos con varios huecos en memoria y varios segmentos por ubicar, es necesario emplear un algoritmo de colocación de segmentos, como el mejor ajuste o el primer ajuste. Esta labor la realiza el planificador a largo plazo.

### 7.11 Segmentación paginada

Aparte de los sistemas de paginación y segmentación puros, existen sistemas que combinan ambas técnicas. Ejemplos de éstos son el sistema operativo MULTICS con la arquitectura GE 645 y el sistema OS/2 para Intel 386.

Éstos se caracterizan porque ofrecen al usuario una memoria lógica que partitionan en segmentos, pero cada uno se almacena en un conjunto de páginas. De esta forma, se elimina la fragmentación externa y se resuelve el problema de la colocación. La figura 7.23<sup>1</sup> muestra el esquema de uno de estos sistemas. En ella, la dirección lógica consta de tres componentes, número de segmento, número de página y desplazamiento. El primero se usa como índice para acceder a la tabla de segmentos, cuya dirección base viene dada por el registro STBR, que proporciona la dirección base de la tabla de páginas. El número de página se usa como índice en esta tabla para obtener la dirección del marco. Finalmente, se concatena con el desplazamiento dando como resultado la dirección física deseada.

<sup>1</sup>Para una mayor claridad, se ha omitido la comparación del número de segmento con la longitud de la tabla de segmentos, al igual que el desplazamiento con la longitud de éste.

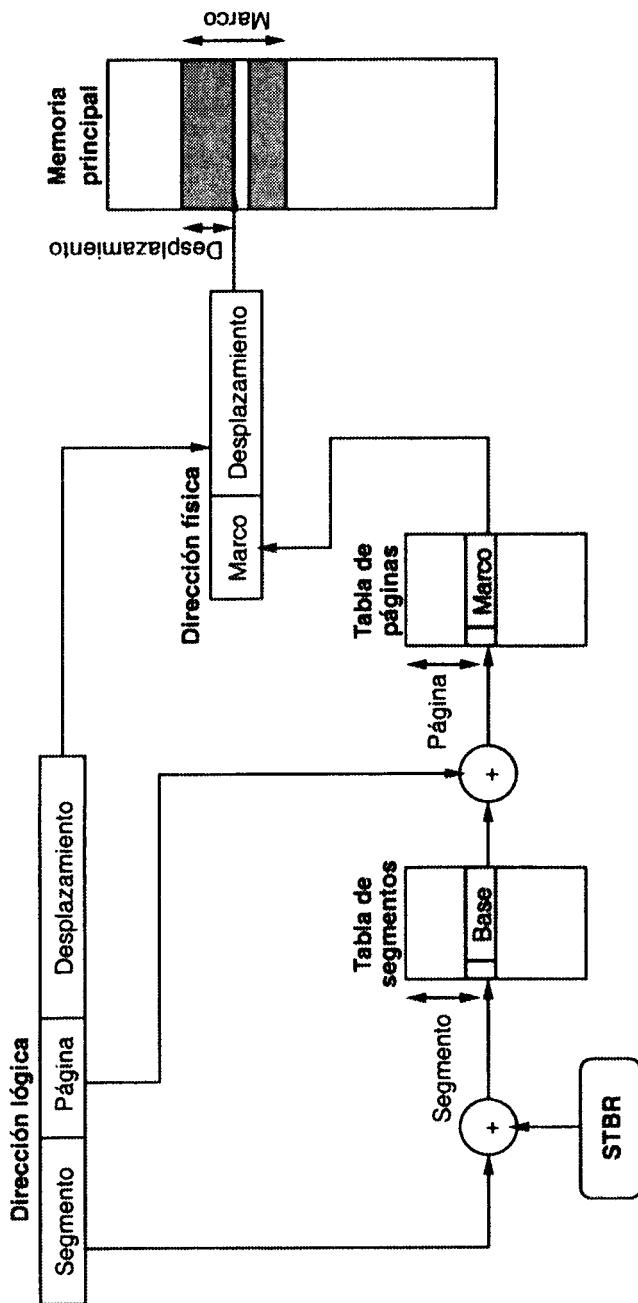


Figura 7.23: Traducción de direcciones en un sistema segmentado paginado

## 7.12 Resumen

El administrador de la memoria es la parte del sistema operativo que se encarga del manejo de ésta. Entre las labores que debe realizar están: controlar las zonas de memoria ocupadas y libres, asignar memoria a los procesos, retirársela, controlar en qué zona reside cada uno, etc.

Los sistemas pueden adoptar distintos esquemas de asignación de la memoria principal. Éstos se pueden clasificar en dos grandes grupos, los de asignación contigua, que exigen que la imagen del proceso se cargue en una zona contigua de memoria; y los de asignación no contigua, que no presentan esta exigencia. Dentro del primer grupo se encuentran los sistemas de monoprogramación y los de multiprogramación con particiones. La paginación y la segmentación son esquemas de asignación no contigua que se usan en los sistemas de memoria virtual que se estudiarán en el capítulo 8.

En los sistemas de monoprogramación sólo residen en memoria principal el sistema operativo y un proceso de usuario; sin embargo, los sistemas de multiprogramación son más complejos porque varios procesos de usuario comparten la memoria. En este caso, existen diversas formas de asignarla. Se pueden crear una serie de particiones de tamaño fijo, o bien se pueden crear las particiones a medida que se necesitan adaptándolas al tamaño de los procesos.

La paginación es un esquema de asignación no contigua que divide los procesos en bloques de tamaño fijo llamados páginas. La memoria principal también se considera dividida en bloques del mismo tamaño que las páginas, que se denominan marcos. Una página puede ser cargada en cualquier marco.

Otro esquema de asignación no contigua es la segmentación, que divide el proceso en fragmentos, denominados segmentos, que pueden ser de diferentes tamaños. Estos segmentos pueden cargarse en cualquier zona contigua de memoria.

Existen esquemas más complejos que combinan la paginación con la segmentación para aprovechar las ventajas de ambos.

## 7.13 Ejercicios

1. ¿Qué diferencias existen entre las direcciones lógicas y físicas?
2. Diga qué diferencias hay entre la fragmentación externa e interna.
3. ¿Qué tipo de fragmentación puede aparecer en los siguientes esquemas de gestión de memoria? Razoné las respuestas.

- (a) Máquina desnuda.
  - (b) Asignación contigua de un sólo proceso (monoprogramación).
  - (c) Particiones fijas.
  - (d) Particiones variables.
  - (e) Paginación.
  - (f) Segmentación.
4. Para cada uno de los esquemas anteriores indique de forma razonada en qué etapa se debe realizar la traducción de direcciones (compilación, carga o ejecución).
  5. En un esquema de protección de memoria con registro base y longitud, es necesario un esquema de reubicación dinámica, donde primero comparamos la longitud y luego se procede a obtener su dirección absoluta. Ventajas e inconvenientes de emplear este esquema frente a realizar primero la traducción de la dirección y luego comparar el límite físico (no la longitud).
  6. ¿Qué es un proceso reubicable?
  7. ¿Qué hay que hacer para reubicar un proceso cuya traducción de direcciones se ha realizado en tiempo de compilación? ¿Habría alguna diferencia si se hubiese realizado en tiempo de carga? Razoné las respuestas.
  8. ¿Es necesaria la compactación en un sistema paginado? ¿Y en un sistema segmentado? Razoné las respuestas.
  9. ¿En qué momento (compilación, carga o ejecución) se ha de realizar la traducción de las direcciones en un sistema segmentado? Razoné la respuesta.
  10. ¿Qué función realiza el planificador a largo plazo en un esquema de gestión de memoria de particiones fijas con una única cola?
  11. En un sistema de particiones fijas, ¿cómo se determina el grado de multi-programación? ¿Y en uno de particiones variables?
  12. Disponemos de un sistema paginado con direcciones físicas y lógicas de 16 bits y páginas de 1024 bytes. En un instante dado, la tabla de páginas presenta el siguiente aspecto:

Página	Marco
0	3
1	10
2	4
3	NULL
...	...

- (a) ¿Cuántas páginas puede tener un proceso? ¿Y cuántos marcos tiene la memoria?
- (b) ¿Cuál es el tamaño del marco de memoria?
- (c) Exprese las siguientes direcciones relativas en lógicas (página y desplazamiento), y en físicas (marco, desplazamiento).
- 1120
  - 2398
  - 3597
13. Un sistema tiene 32 marcos, cada uno de ellos de 1024 bytes. Las direcciones lógicas tienen el mismo tamaño que las direcciones físicas. Conteste las siguientes preguntas de forma razonada:
- ¿Cuántas entradas tendremos en la tabla de marcos?
  - Si cada entrada ocupa 4 bytes, ¿qué memoria se necesita para almacenar esta tabla?
  - ¿Cuántos bits tiene la dirección física?
  - ¿Cuántas entradas habrá en la tabla de páginas de cada proceso?
14. Considere un sistema de particiones variables. En un momento dado la memoria presenta los siguientes huecos (en orden de direcciones físicas): 10 KiB, 4 KiB, 20 KiB, 18 KiB, 7 KiB, 9 KiB, 12 KiB, 15 KiB. A continuación se realizan las siguientes solicitudes de forma sucesiva: 12 KiB, 10 KiB y 9 KiB. ¿Qué hueco le corresponde a cada solicitud si empleamos las siguientes estrategias de colocación?
- Primer ajuste.
  - Siguiente ajuste, donde el último proceso que entró se situó entre los huecos de 20 KiB y 18 KiB.
  - Mejor ajuste.
  - Peor ajuste.
15. Considere un sistema de 4200 palabras de memoria principal que utiliza un esquema de particiones variables. Éste permite realizar compactación cuando existe un proceso que no tiene un hueco para su tamaño. En un instante determinado disponemos de tres procesos en memoria con los siguientes datos:

Dirección de inicio	Longitud
1000	1000
2900	500
3400	800

- (a) Mostrar la lista de huecos y procesos que lleva el control de la memoria.
  - (b) Si se crean tres nuevos procesos cuyos tamaños son 500, 1200 y 200, determine cómo quedará la lista de huecos y procesos si se utiliza la estrategia de colocación del mejor ajuste.
16. Suponga un sistema con direcciones lógicas de 16 bits, con un tamaño máximo de segmento de 4 KiB. ¿Cuál es el formato de la dirección lógica? ¿Cuántas entradas tiene la tabla de segmentos?
17. Considere un sistema con 32 MiB de memoria principal, donde el esquema de gestión de memoria es de particiones variables. La unidad de asignación mínima es de 1 byte y las direcciones y los enteros son de 32 bits. En un momento dado, disponemos de diez procesos cargados en memoria y 20 huecos. ¿Qué espacio ocuparían un mapa de bits y una lista enlazada para la gestión de la memoria libre? Razona las respuestas.



# Capítulo 8

## Memoria virtual

---

En este capítulo se introduce el concepto de memoria virtual. Este tipo de sistemas, a diferencia de los estudiados en el capítulo anterior, permite que un proceso se ejecute sin que su imagen esté cargada completamente en la memoria física. Este modo de funcionamiento necesita un soporte hardware, así como una serie de políticas del sistema operativo. Ambos aspectos serán tratados en este capítulo.

### 8.1 Introducción

Los sistemas clásicos de administración de la memoria exigen que un proceso esté cargado completo en ella para poder ejecutarse; esto tiene el inconveniente de que no podemos tener procesos más grandes que la memoria física disponible. Una forma de vencer esta limitación consiste en dividir los programas en fragmentos independientes que puedan ejecutarse de forma separada, que se conocen con el nombre de **superposiciones**. Este sistema tenía el inconveniente de que era el programador el que debía realizar la división.

El concepto de **memoria virtual** supone un avance importante en este campo ya que permite la ejecución de procesos que no están cargados en memoria física en su totalidad. Además tiene la ventaja de liberar al programador de especificar qué partes deben estar cargadas y cuáles no, ya que todo este trabajo lo realiza

el sistema operativo.

Podemos definir los sistemas de memoria virtual como el conjunto de técnicas de gestión de memoria que nos permite ejecutar un proceso sin que su espacio lógico de direcciones se encuentre en memoria física en su totalidad. Éstos permiten abstraer al usuario de las características de la memoria física, tales como su tamaño o el hecho de que esté compartida por otros procesos.

En estos sistemas sólo se mantienen en memoria física aquellas partes del proceso que se necesitan para su ejecución, el resto se guarda en memoria secundaria. Si durante la ejecución del proceso se hace referencia a un dato o instrucción que no resida en memoria física será necesario transferirlo desde la memoria secundaria. Dado que esto puede producir una disminución de la velocidad de ejecución de los procesos y, por tanto, del rendimiento del sistema, en su momento se mantuvieron muchos debates sobre su efectividad. Sin embargo, la experiencia ha demostrado que este tipo de sistemas proporciona un rendimiento adecuado, y de hecho es el sistema de gestión de memoria que se suele emplear actualmente.

En los sistemas de memoria virtual se suele emplear el término **memoria real** para designar a la memoria física, en contraposición al concepto de memoria virtual. Ésta es una memoria mucho más grande, ya que abarca tanto la memoria física como el almacenamiento secundario. La parte de un proceso que está actualmente en memoria principal se denomina **conjunto residente**.

Las ventajas principales que proporciona la memoria virtual es que el tamaño de la imagen de un proceso puede ser mayor al de la memoria física disponible. Como consecuencia de esto, una ventaja aún más importante es que la suma de los espacios direccionables de los procesos activos, puede exceder la capacidad de la memoria física. Además, libera al programador de conocer los límites impuestos por la capacidad de almacenamiento real del sistema.

## 8.2 El principio de localidad

El buen funcionamiento de los sistemas de memoria virtual está basado en el **principio de localidad**. Éste establece que las referencias al código y a los datos dentro de un proceso tienden a estar agrupadas durante períodos cortos de tiempo. Durante períodos largos, estos grupos van cambiando.

El análisis del funcionamiento de los programas reales nos puede ayudar a comprender el principio de localidad:

**Ejecución secuencial** El flujo de ejecución de un programa suele ser secuencial,

salvo en las instrucciones de salto y de llamada, que suponen una pequeña parte del total. Por tanto, la próxima instrucción a ejecutar será normalmente la siguiente a la actual.

**Bucles** Los bucles suelen constar de un número de instrucciones relativamente pequeño que se repiten muchas veces. Durante la ejecución de una iteración, sólo se están empleando un número pequeño de instrucciones, que se repiten mientras dure la ejecución del bucle.

**Código de uso poco frecuente** En la mayor parte de los programas suele haber secciones de código que se ejecutan con poca frecuencia. Por ejemplo, las destinadas a la gestión de errores, los correctores sintácticos de los editores, etc.

**Procesamiento de estructuras** La utilización de estructuras de datos grandes, tales como matrices, emplean normalmente un procesamiento secuencial referenciándose elementos contiguos en la estructura<sup>1</sup>.

**Espacio reservado para estructuras** Normalmente, ciertas estructuras de datos, como matrices, listas y tablas, se les suele asignar más memoria de la que realmente necesitan.

A la vista de estas consideraciones se puede apreciar como, en muchas ocasiones, no es necesario disponer de toda la imagen del proceso en memoria. Incluso en aquellos casos en los que se utilice por completo, no se necesitará en su totalidad durante todo el tiempo que dure la ejecución.

## 8.3 Principios de operación

Los sistemas de memoria virtual se pueden implementar basándose en la paginación, en la segmentación, o en una combinación de ambas; en este último caso el tratamiento es muy similar al de la paginación puesto que se utilizan segmentos divididos en páginas. El estudio que se realiza en este capítulo se va a centrar fundamentalmente en los sistemas de memoria virtual paginados.

La paginación fue utilizada por primera vez por los diseñadores del computador Atlas para proporcionar solamente una memoria virtual grande, ya que el sistema

---

<sup>1</sup> Hay que hacer notar, que la forma en que las estructuras de datos están almacenadas en memoria juega un papel importante en el comportamiento de la localidad. Por ejemplo, una matriz bidimensional almacenada por columnas pero procesada por filas podría requerir acceder a datos ubicados físicamente dispersos.

no contemplaba inicialmente multiprogramación. Este computador fue construido en la Universidad de Manchester en 1960 [Kilb62].

En un sistema de memoria virtual paginado, cada proceso sólo mantiene en memoria principal unas cuantas páginas del total que componen su imagen, esto es lo que se conoce como conjunto residente. La imagen completa del proceso se sitúa en memoria secundaria, y sólo se traen a memoria principal las páginas que se necesitan en cada instante. Las decisiones de qué páginas traer, cuándo traerlas, y dónde ubicarlas son las que afectan al gestor de memoria virtual.

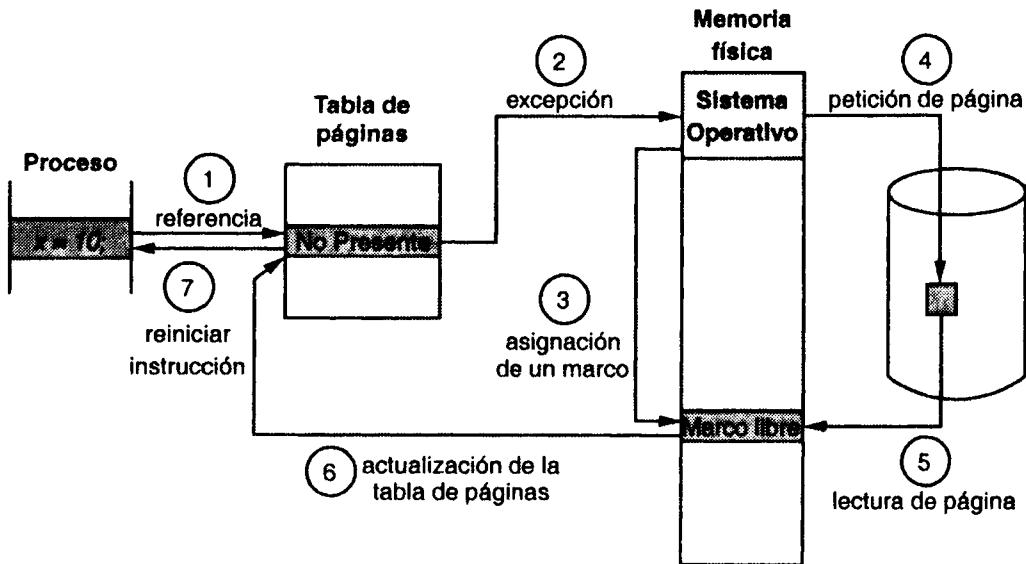


Figura 8.1: Pasos realizados durante un fallo de página

Mientras el proceso se ejecute y refiera posiciones de memoria del conjunto residente no habrá ningún tipo de problema. Mediante la tabla de páginas, el sistema es capaz de determinar si las páginas referenciadas se encuentran o no en memoria. En este último caso, se genera una excepción que indica un fallo de acceso a memoria. Esto es lo que se conoce como **fallo de página**. Cuando se referencia una dirección virtual que no está presente en memoria física, tendremos que ir a la memoria secundaria para traer la página. La figura 8.1 muestra gráficamente los pasos a realizar:

**Paso 1** Cuando se produce una referencia a memoria durante la ejecución de un proceso, se consulta la tabla de páginas para realizar la traducción de direcciones.

Paso 2 Si la página no se encuentra en memoria se produce una excepción por parte del proceso y el sistema operativo toma el control. Éste lleva a cabo las siguientes acciones:

- (a) Realiza un cambio de contexto, es decir, se guarda la información que contienen los registros del procesador sobre el proceso.
- (b) Determina que la excepción producida se debe a un fallo de página.
- (c) Realiza las comprobaciones de protección, es decir, determina si la página es válida para el proceso y si el tipo de acceso es el adecuado. Si alguna de éstas falla, termina la ejecución del proceso.

Paso 3 Si la página es válida, el sistema intenta asignar un nuevo marco al proceso:

- (a) Si hay marcos libres, se le asigna uno y se marca como ocupado.
- (b) Si no los hay, tendrá que elegir uno de los ocupados para almacenar la nueva página (es lo que se conoce como **sustitución de página**). Si la página seleccionada ha sido modificada desde que fue introducida en memoria, tiene que ser reescrita en disco. El marco escogido se marca como ocupado.

Paso 4 Solicita al subsistema de E/S traer la nueva página del disco al marco libre anterior. El proceso que ocasionó el fallo de página pasa al estado de bloqueado a la espera de la página pedida, y el sistema selecciona un nuevo proceso para ejecutar. Es decir, se produce un cambio de proceso.

Paso 5 El dispositivo transfiere la página al marco seleccionado, produciendo una interrupción al finalizar. El sistema operativo atiende la interrupción y si se estaba ejecutando otro proceso habrá que hacer un nuevo cambio de contexto.

Paso 6 El sistema operativo actualiza la tabla de páginas para indicar que la página está en memoria. Cambia el estado al proceso que la esperaba, es decir, lo pasa del estado de bloqueado a listo.

Paso 7 El proceso espera a que el planificador a corto plazo le asigne de nuevo la CPU para continuar con la ejecución de la instrucción interrumpida.

Debido a que cuando se produce un fallo de página se interrumpe el proceso y se guarda su contexto, podemos reanudar su ejecución en la misma instrucción que produjo el fallo, con la diferencia de que la página solicitada se encuentra en memoria principal. De este modo, es posible ejecutar un proceso incluso si no dispone de páginas en la memoria física.

Durante la gestión de un fallo de página se pueden producir varias operaciones de lectura y/o escritura en disco. Debido a que en esos instantes el procesador permanecería ocioso se produce el cambio de proceso para mejorar el rendimiento.

Los fallos de página afectan de forma significativa al rendimiento del sistema, debido a que el tiempo de acceso efectivo a memoria aumenta, principalmente por las operaciones de E/S. Por lo tanto, un esquema de gestión de memoria virtual debe procurar que el número de fallos de páginas sea mínimo.

### 8.3.1 El área de intercambio

Un aspecto interesante en la memoria virtual es la gestión del área de intercambio. En los primeros sistemas de gestión de memoria basados en particiones, el área de intercambio desempeñaba una función bien distinta a la actual. Cuando en un sistema se encontraban varios procesos bloqueados, se podía suspender la ejecución de uno de ellos, descargándolo de memoria principal y pasándolo al área de intercambio. Posteriormente ese proceso volvería a memoria principal para continuar con su ejecución. Esta técnica de intercambio se empleaba en sistemas con una planificación de asignación por turnos, de forma que cuando terminaba el cuento de un proceso, el administrador de memoria lo intercambiaba, para poder admitir un nuevo proceso al sistema.

Actualmente, esta técnica de intercambio de procesos se suele emplear en pocos sistemas. Así, en sistemas UNIX se emplea una modificación, que consiste en activarla solamente cuando la carga del sistema es elevada debido a la ocurrencia de numerosos fallos de página. Esto se estudiará con más detalle en el apartado 8.5.6.

A pesar de no emplearse la técnica de intercambio, el área de intercambio se sigue utilizando en la actualidad en los sistemas de memoria virtual. Existen diversos esquemas de utilización de esta área. En sistemas UNIX cuando se crea un proceso se asigna el espacio suficiente para alojar las páginas de código y datos en el área de intercambio. Cuando se inicia el proceso, las páginas de código se leen del sistema de ficheros y las de datos se traen también de éste o se crean (si no tienen valores iniciales). Cuando se realiza una sustitución para resolver un fallo de página, la página seleccionada se lleva al área de intercambio. De este modo, la primera vez que se referencia una página se lee del sistema de ficheros, y si es sustituida y vuelve a referenciarse lo hace desde el área de intercambio.

En otros sistemas como Windows NT o Solaris 2, cuando se hace referencia por primera vez a una página se trae del propio fichero ejecutable. Cuando una página que ha sido modificada tiene que salir de memoria principal se escribe en

el área de intercambio. De este modo, sólo se mantienen en ésta las páginas que han sido referenciadas y modificadas alguna vez. Las páginas de código que nunca se modifican, siempre se leen del propio fichero ejecutable; si tienen que salir de memoria se desechan.

Un aspecto relacionado con el área de intercambio es su ubicación. En algunos sistemas, como Windows NT, se utiliza un fichero grande dentro del propio sistema de ficheros para mantenerla. Esto permite aprovechar la propia estructura del sistema de ficheros (se verá en el capítulo 10) para gestionarla y que ésta presente un tamaño variable según las necesidades del sistema; sin embargo, las operaciones de E/S se realizan de forma más lenta. En otros, como UNIX, permiten que el área de intercambio pueda residir en una partición del disco; de este modo, al no necesitar ninguna estructura del sistema de ficheros, las operaciones de E/S se realizan de forma más rápida, pero requiere un gestor exclusivo del espacio del área de intercambio.

## 8.4 Estructuras hardware y de control

Para que los sistemas de memoria virtual sean prácticos y efectivos necesitan dos ingredientes. Por un lado, debe haber un soporte hardware para que se pueda emplear un esquema de paginación o segmentación. Por otro, el sistema operativo debe incluir software adecuado para manejar el movimiento de páginas y/o segmentos entre memoria secundaria y memoria principal. A continuación examinaremos los aspectos hardware, así como las estructuras de control necesarias que crea y mantiene el sistema operativo, pero que son utilizados por el hardware de manejo de memoria.

Consideraremos un sistema de memoria virtual paginado, por lo que necesitaremos las estructuras vistas en el capítulo 7, para llevar el control de la memoria: la tabla de marcos y la tabla de páginas.

### 8.4.1 Tabla de marcos

La tabla de marcos permite al sistema operativo conocer los detalles de asignación de la memoria física, es decir, qué marcos están asignados y cuáles están disponibles. Esta tabla es única en el sistema y posee una entrada por cada marco de memoria. Cada entrada contiene la siguiente información:

**Bit de estado** Indica si el marco está o no ocupado.

**Página** Número de página que lo ocupa.

**Proceso** Identificador del proceso propietario de la página.

**Contador** Número que procesos que comparten la página.

**Bit de bloqueo** Indica si el marco está o no bloqueado. Si lo está, la página que contiene no podrá ser sustituida. Un ejemplo de esto son los marcos que contienen al sistema operativo.

#### 8.4.2 Tabla de páginas

En el capítulo anterior se introdujo el concepto de tabla de páginas y su utilidad en la traducción de direcciones lógicas a físicas. Una vez visto el concepto de memoria virtual estamos en condiciones de profundizar más en su estructura. Las entradas de las tablas de páginas reales son más complejas que las vistas hasta ahora. Su estructura depende del sistema, pero de forma general se suele encontrar en ellas la siguiente información:

**Bit de presencia** Dado que sólo algunas de las páginas del proceso pueden estar en memoria principal, se necesitará este bit para indicarlo. También se conoce como **bit de fallo**.

**Número de marco** Lugar de la memoria física donde se encuentra situada la página.

**Bit de modificación** Indica si el contenido de la página correspondiente ha sido alterado desde que fue cargada por última vez en memoria principal. Si no lo ha sido, no será necesario escribirla en disco cuando tenga que salir de la memoria.

**Bit de referencia** Señala si la página ha sido referenciada recientemente. Lo utilizan los algoritmos de sustitución de páginas.

**Bits de protección** Denotan las operaciones permitidas para la página.

**Bit de compartición** Muestra si la página puede o no ser compartida por varios procesos.

En los sistemas actuales, cuando la página no está presente en memoria (bit de presencia a 0), el resto de bits de la entrada se emplea para almacenar la dirección dentro de memoria secundaria donde se encuentra.

En la implementación de las tablas de páginas, hemos de tener en cuenta dos aspectos, el aumento del tiempo de acceso a una dirección de memoria y el elevado número de entradas que pueden tener en los sistemas actuales.

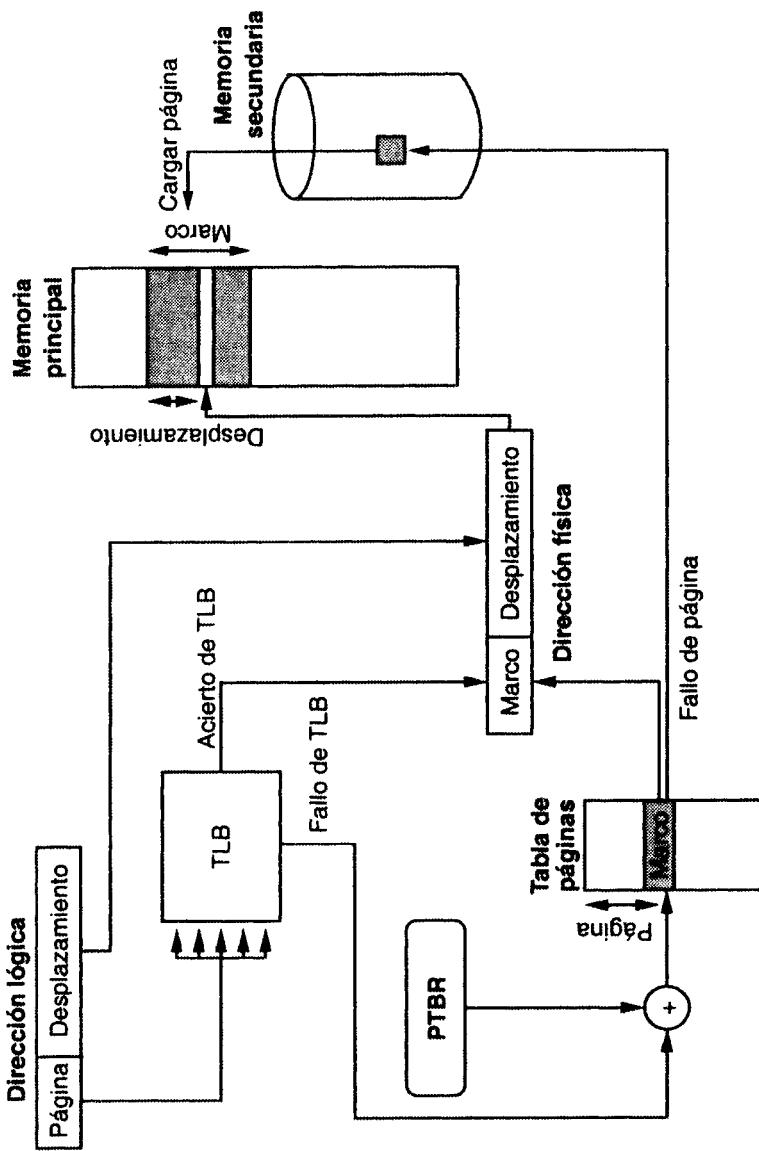


Figura 8.2: Traducción en un sistema de memoria virtual con TLB

En un sistema de paginación el acceso a una dirección de memoria es mapeado. Es decir, la dirección lógica tiene que ser previamente traducida, para lo cual se necesita acceder a la tabla de páginas. Esto implica un aumento en el tiempo de acceso a la dirección requerida, ya que habrá que hacer dos accesos por cada una, uno a la tabla de páginas y otro a la propia dirección. Este aumento será mayor o menor, dependiendo de cuál sea la implementación hardware de la tabla de páginas. En el capítulo 7 se estudiaron diferentes posibilidades, que son válidas en un sistema de memoria virtual paginado, siendo los *buffers* de traducción adelantada (TLB) los más empleados. La figura 8.2 muestra la traducción de direcciones en un sistema de memoria virtual paginado que usa TLB.

El segundo aspecto a considerar es el elevado número de entradas de las tablas de páginas. En los sistemas de memoria virtual el tamaño de la tabla de páginas puede ser muy grande dado que se puede tener procesos de mayor tamaño que la memoria física, aprovechando de este modo todo el espacio de direcciones que permite la arquitectura. Hoy en día, la mayoría de los sistemas de computación soportan un espacio de direcciones lógicas muy grande ( $2^{32}$  o  $2^{64}$ ). En un ambiente de este tipo las tablas de páginas serían demasiado grandes. Por ejemplo, consideremos un sistema con un espacio de direcciones lógicas de 32 bits. Si el tamaño de la página es de 4 KiB, son necesarias  $2^{20}$  entradas en la tabla de páginas, algo más de un millón de entradas. Si cada una tiene 4 bytes, se necesitan unos 4 MiB de memoria para almacenarlas. Además, hay que recordar que cada proceso necesita su propia tabla de páginas.

Se han propuesto diversos métodos para almacenar las tablas de páginas sin que sea necesario gastar tanta memoria. A continuación se verán dos de ellos, las **tablas de páginas multinivel**, y las **tablas de páginas invertidas**.

#### 8.4.2.1 Tabla de páginas multinivel

Una solución sencilla para no tener que asignar un espacio de memoria contiguo tan grande a la tabla de páginas, consiste en dividirla en fragmentos más pequeños, es decir, paginar la tabla de páginas. La figura 8.3 muestra la diferencia entre utilizar una tabla de páginas convencional con un solo nivel, y una tabla de páginas de dos niveles. En este esquema hay una tabla de páginas de primer nivel, en la que cada entrada apunta a una tabla de páginas. Normalmente se establece una longitud máxima para la tabla de páginas, que suele ser igual al tamaño de una página.

Para ilustrar esta técnica usaremos un ejemplo con una máquina de 32 bits con páginas de 4 KiB y entradas de 4 bytes. Una dirección lógica está dividida en un número de página que consta de 20 bits y un desplazamiento de página que

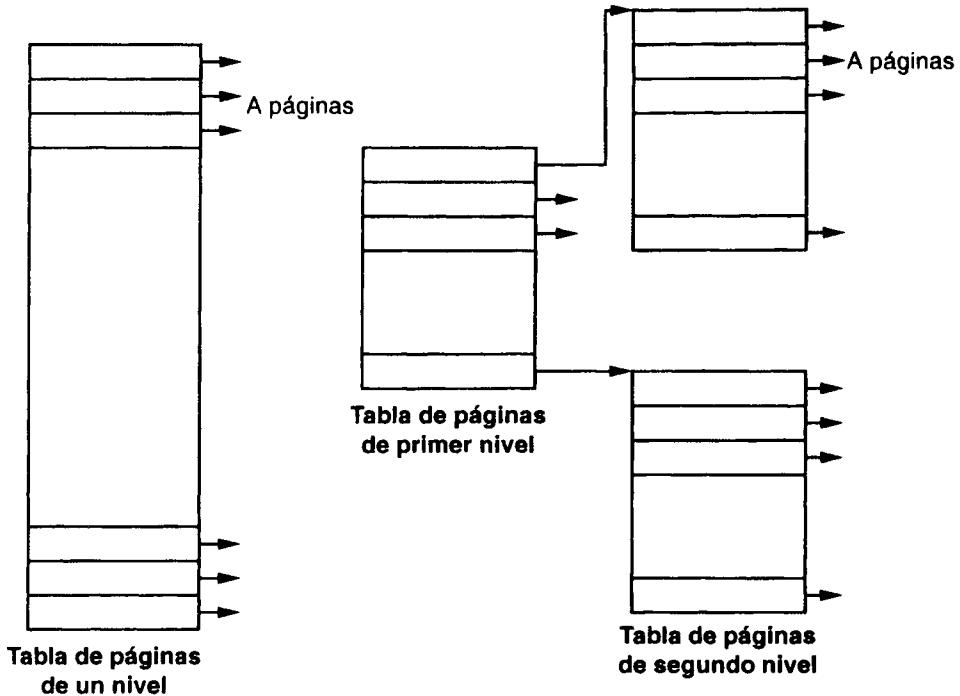


Figura 8.3: Tabla de páginas de un nivel y de dos niveles

consta de 12. Puesto que vamos a utilizar una tabla de páginas de dos niveles, el número de página deberá estar a su vez dividido en dos componentes. De esta forma las direcciones tendrían la siguiente estructura:

- Un campo,  $p_1$ , de 10 bits para el primer nivel, lo que supone una tabla de 1024 entradas, para que ocupe como mucho una página.
- Un campo,  $p_2$ , de 10 bits para el segundo nivel, que supone otras 1024 entradas.
- Un campo,  $d$ , de 12 bits para el desplazamiento puesto que la página es de 4 KiB.

El esquema de la traducción de direcciones para esta arquitectura se puede ver en la figura 8.4. Dada una dirección lógica, empleamos el primer campo para buscar en una tabla de primer nivel de 1024 entradas. Ésta a su vez apunta a otra tabla de 1024 entradas, utilizándose el segundo campo para escoger una entrada de ella, que es la que contiene la dirección base del marco buscado. A esta

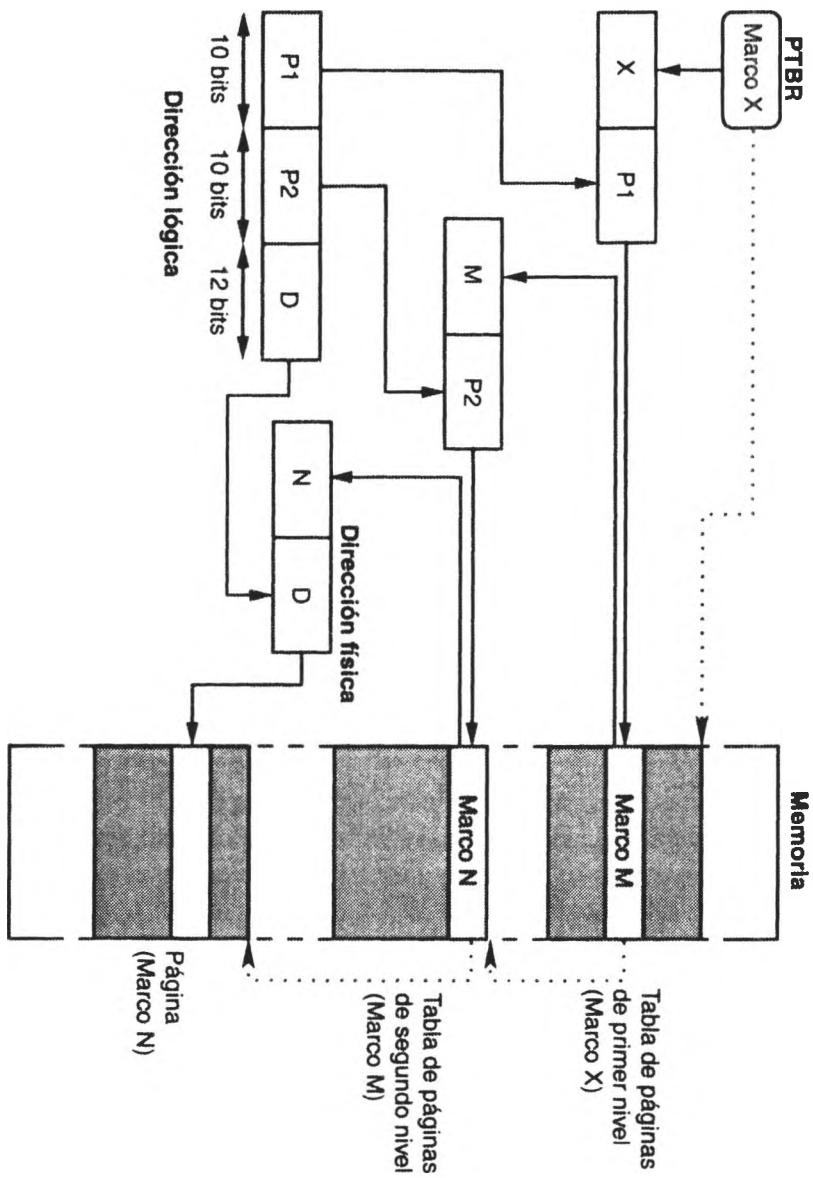


Figura 8.4: Tabla de páginas de dos niveles

dirección se le suma el tercer campo y se obtiene la dirección física buscada. De este modo, en este sistema un proceso podrá ocupar  $1024 * 1024$  páginas de 4 KiB cada una, empleando 1025 tablas para indicar dónde se encuentra almacenado (1 de primer nivel, y 1024 de segundo) cada una con 1024 entradas.

La ventaja de este esquema es que no se necesita mantener en memoria todas estas tablas a la vez, sino sólo unas pocas, manteniéndose el resto en memoria virtual. Además, si el proceso no ocupa todas las páginas disponibles, se necesitarán menos tablas ya que sólo se necesitan tantas de segundo nivel como entradas de las de primer nivel haya ocupadas.

Para un sistema con un espacio de direcciones lógicas de 64 bits, el esquema de paginación de 2 niveles ya no es apropiado. Para ilustrar esto, supongamos que el tamaño de página en tal sistema es de 4 KiB. En este caso, la tabla de páginas constará de  $2^{52}$  entradas. Si usamos un esquema de paginación de dos niveles, entonces la tabla de páginas de primer nivel constará de  $2^{26}$  entradas. Claramente, no podemos asignar la tabla de páginas de primer nivel de forma contigua en memoria (suponiendo 4 bytes por entrada, se requerirían 256 MiB de memoria física para almacenarla). La solución obvia es dividir la tabla de páginas de primer nivel en piezas más pequeñas, es decir, realizar un mayor número de niveles. Esta solución se usa también en algunos procesadores de 32 bits para obtener mayor flexibilidad y eficiencia. Por ejemplo, las arquitecturas SPARC y Motorola 68030, ambas de 32 bits, proporcionan un esquema de paginación de 3 y 4 niveles, respectivamente. Sin embargo, un mayor número de niveles implica más complejidad en el cálculo de la dirección física, y un mayor tiempo de acceso a la memoria.

#### 8.4.2.2 Tabla de páginas invertida

Un enfoque alternativo es el empleo de las tablas de páginas invertidas. En este caso, sólo existe una en el sistema, y posee una entrada por cada marco de memoria. La estructura de las entradas de esta tabla de páginas difiere de la comentada anteriormente, ya que ahora debe mantener, además de la información anterior, el número de página y el identificador del proceso al que pertenece.

La figura 8.5 muestra la traducción de direcciones con una tabla de páginas invertida. En ella las direcciones lógicas consisten en una tripleta (*id\_proceso*, *página*, *desplazamiento*). Cada entrada en la tabla de páginas es en realidad un par (*id\_proceso*, *página*). Cuando se realiza una referencia a memoria, parte de la dirección lógica, en concreto (*id\_proceso*, *página*), se presenta al subsistema de memoria y se busca una concordancia en la tabla de páginas. Si coincide en la entrada *i*, entonces la dirección física es (*i*, *desplazamiento*). Si no hay

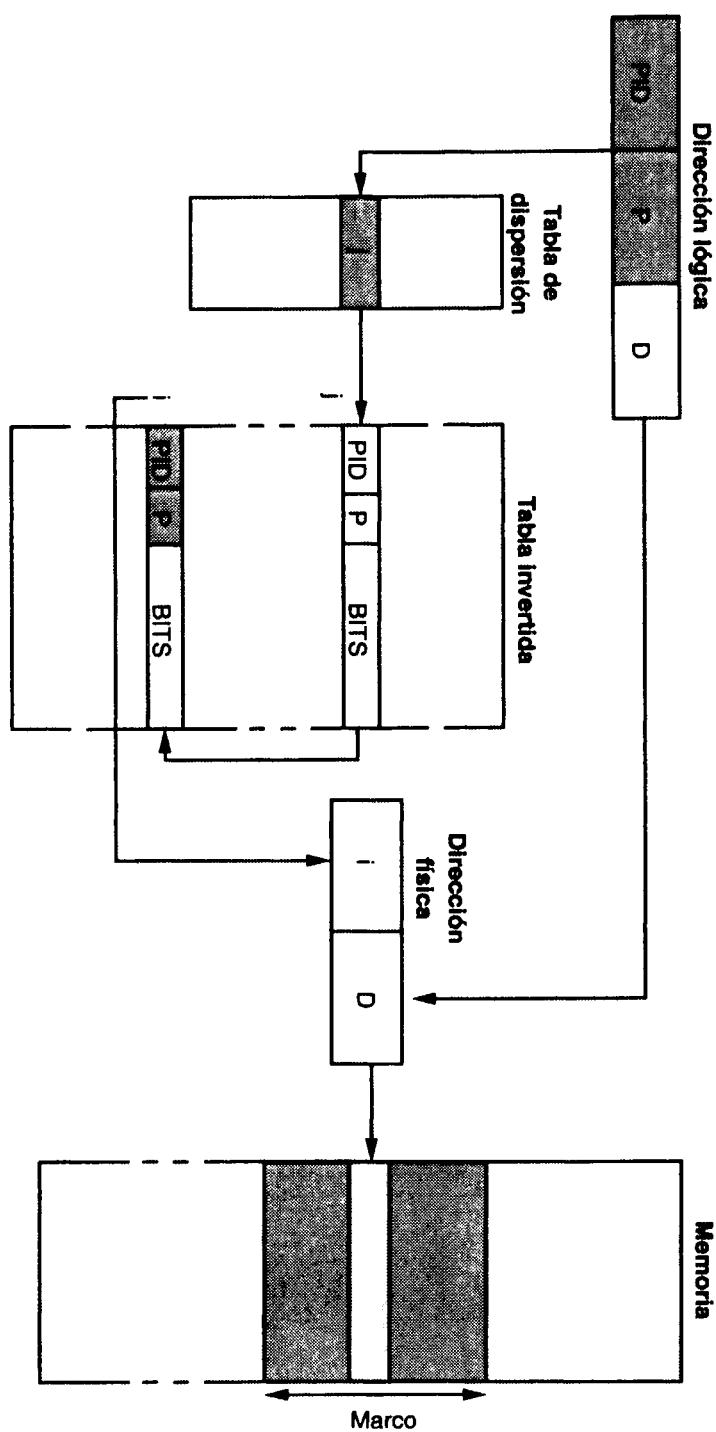


Figura 8.5: Tabla de páginas invertida

concordancia, entonces se produce un fallo de página.

Este esquema reduce la cantidad de memoria necesaria para almacenar la tabla de páginas. Sin embargo, aumenta el tiempo de búsqueda de una entrada, debido a que la tabla de páginas invertida está ordenada por direcciones físicas, y las búsquedas se realizan por direcciones lógicas. En algunos casos puede ser necesario recorrer toda la tabla para encontrar una coincidencia. Esto origina que la búsqueda sea demasiada larga.

Para aliviar el problema se utiliza una **tabla de dispersión**<sup>2</sup>, para limitar la búsqueda a una o unas pocas entradas de la tabla, tal como se observa en la figura 8.5. Por supuesto, cada acceso a la tabla de dispersión añade una referencia más a memoria. De este modo una referencia a memoria virtual requiere al menos dos accesos a memoria real, una para la entrada de la tabla de dispersión y otra para la de páginas. Para mejorar el rendimiento, normalmente se emplean los registros asociativos o TLB (ver capítulo 7). De este modo, si la página buscada se encuentra en estos registros no es necesario acudir a la tabla de páginas invertida.

La tabla de páginas invertida reduce considerablemente el tamaño de las tablas de páginas, pero sólo mantiene información de las páginas alojadas en memoria física. Sin embargo, para mantener la información completa se necesita una tabla de páginas externa por proceso. Éstas se utilizan solamente cuando la página referenciada no se encuentra actualmente en memoria. Su estructura es similar a las tradicionales, conteniendo información acerca de donde está localizada cada página virtual.

Estas tablas de páginas externas se mantienen en memoria secundaria, por lo que cuando se produzca un fallo de página, se necesitará un acceso adicional a memoria secundaria. Por tanto, existe una demora extra en el procesamiento de búsqueda de una página.

Ejemplos de sistemas que utilizan este esquema son el IBM System/38, el IBM RISC System 6000, IBM RT, IBM AS/400 y las estaciones de trabajo Hewlett-Packard Spectrum.

## 8.5 Funciones del gestor de memoria virtual

El gestor de memoria virtual está compuesto de diversos elementos software, que aparecen en la tabla 8.1. Todos tienen como objetivo obtener un buen rendimiento del sistema para lo cual intentan minimizar el porcentaje de fallos de página. Ya que como se ha comentado anteriormente, éstos suponen una considerable

<sup>2</sup>También denominada tabla *hash*.

sobrecarga de trabajo.

<b>Política de lectura</b>	Demanda Prepaginación
<b>Política de colocación</b>	Algoritmos básicos
<b>Política de sustitución</b>	<i>Buffering</i> de páginas
<b>Gestión del conjunto residente</b>	Tamaño del conjunto residente Alcance del reemplazamiento
<b>Política de limpieza</b>	Demanda Limpieza previa
<b>Control de la carga</b>	

**Tabla 8.1:** Funciones del gestor de memoria virtual

### 8.5.1 Política de lectura

También se denomina **política de carga**. Es la encargada de determinar qué elemento traer y cuándo se incorpora desde el almacenamiento secundario a la memoria principal, es decir, cuándo se debe cargar una página en memoria principal. Existen dos alternativas, la **paginación por demanda** y la **prepaginación o paginación previa**.

#### 8.5.1.1 Paginación por demanda

Las páginas son traídas a memoria principal cuando se produce el fallo de página. Con esta política, cuando comienza la ejecución de un proceso no dispone de ninguna página en memoria, por lo que producirá un elevado número de fallos de página. Según el principio de localidad, a medida que se vayan cargando páginas en memoria el número de fallos disminuirá. Pasado un cierto tiempo, el proceso empezará a hacer referencia a direcciones que no se encuentran en las páginas cargadas, por lo que se producirá de nuevo un incremento en el número de fallos de página. Esto irá ocurriendo sucesivas veces hasta la terminación del proceso.

#### 8.5.1.2 Prepaginación

Con esta técnica se llevan a memoria otras páginas, además de las demandadas por los fallos de página. De este modo, el sistema se adelanta a las necesidades futuras, es decir, intenta tener el elemento antes de que sea necesario. La eficacia

de esta estrategia depende de si las páginas que se traen van a ser referenciadas en un futuro próximo.

El fundamento de esta estrategia se encuentra en las características de los dispositivos de almacenamiento secundario. Dado que traer una página a memoria principal requiere de un tiempo de acceso al dispositivo más un tiempo de transferencia, es más eficiente traer varias páginas contiguas en el almacenamiento secundario, cuando se produce un fallo de página, en lugar de traerlas una a una. Sin embargo, la utilidad de la prepaginación no ha sido demostrada.

### 8.5.2 Política de colocación

Determina dónde se va a situar una porción de un proceso en memoria principal. En los sistemas de paginación pura o segmentación paginada, la ubicación es inmediata y no presenta ninguna relevancia esta política, pues cualquier marco libre es adecuado para alojar una página.

Sin embargo, en el caso de un sistema de segmentación, la política de ubicación sí cobra importancia en el diseño del gestor de memoria virtual. En este caso se necesitará un algoritmo de colocación de los segmentos, tales como el mejor ajuste, primer ajuste, etc., vistos en el capítulo 7.

### 8.5.3 Política de sustitución

Cuando un proceso produce un fallo de página no puede continuar hasta que no se incorpore a memoria principal la página ausente. Puede ocurrir que no existan marcos libres de memoria para asignárselo a la página solicitada. En este caso, habrá que sacar una de las que están en memoria para atender la demanda de la página que ocasionó el fallo. Para ello, tenemos que elegir una página víctima mediante un algoritmo de sustitución, para que su lugar lo ocupe la página solicitada.

A la hora de abordar la sustitución de páginas hay que tener en cuenta varios aspectos:

- El número de marcos asignados a cada proceso activo.
- El conjunto de páginas candidatas para la sustitución, sólo las del proceso que ocasiona el fallo o todas las que residen en memoria.
- La elección de la página a ser sustituida.

El estudio conjunto de los dos primeros aspectos se conoce como **gestión del conjunto residente**, mientras que el último es la **política de sustitución**.

#### 8.5.3.1 Tasa de fallos de página

Existen numerosos algoritmos de sustitución, para compararlos utilizaremos la **tasa media de fallos de páginas**, ya que es ésta la que influye directamente en el rendimiento del sistema. Para mejorarlo, los algoritmos deberían seleccionar aquella página que tenga la menor probabilidad de volver a ser referenciada próximamente. El principio de localidad establece una relación entre el comportamiento actual y en un futuro cercano del proceso. De ahí que los distintos algoritmos de sustitución intenten predecir el comportamiento futuro en función del pasado reciente.

La tasa de fallos de página se ve influida por varios factores. Por un lado, va a variar según el número de marcos de memoria de los que dispone el proceso. Estudios experimentales han comprobado que el comportamiento de los procesos es el que se muestra en la figura 8.6. En ella se observa que cuando el número de marcos del que dispone el proceso es muy pequeño la tasa de fallos es muy alta; a medida que aumenta la memoria asignada al proceso, la tasa disminuye. Sin embargo esa disminución no es lineal, a partir de un cierto valor la asignación de memoria adicional no repercute de forma significativa en la tasa de fallos de página. Obviamente, si a un proceso se le asigna un número de marcos suficiente para alojar todas sus páginas su tasa de fallos será cero.

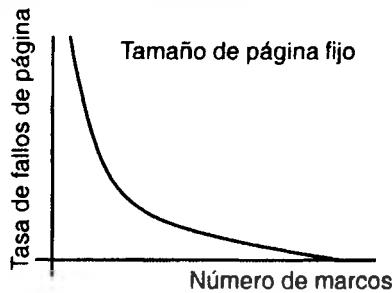
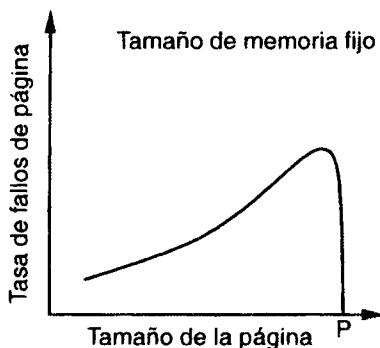


Figura 8.6: Relación entre el nº de marcos y la tasa de fallos de página

Otro factor que influye sobre la tasa de fallos de página es el tamaño de ésta. La figura 8.7 muestra cómo varía la tasa de fallos frente al tamaño de la página para una cantidad de memoria fija. El comportamiento puede explicarse teniendo en cuenta el principio de localidad. Cuando el tamaño de la página es pequeño se observa que la tasa de fallos es baja, debido a que el proceso dispone de un elevado

número de páginas en memoria y, por tanto, es probable que toda su localidad esté residente. Cuando aumenta el tamaño de la página se observa que también lo hace la tasa de fallos. Esto es debido a que el número de páginas de las que dispone el proceso en memoria es más reducido y a que éstas contienen un mayor número de direcciones que no pertenecen a la localidad actual. Obviamente, cuando el tamaño de la página coincide con el del proceso la tasa de fallos de página es cero (punto P).



**Figura 8.7:** Relación entre el tamaño de la página y la tasa de fallos de página

### 8.5.3.2 Algoritmos de sustitución

Se han propuesto diversos algoritmos de sustitución de páginas; aquí vamos a considerar algunos de ellos: óptimo, FIFO, LRU y 2 variantes del algoritmo del reloj. Es importante tener en cuenta que cuanto más compleja sea la política de sustitución, mayores serán sus requisitos hardware para implementarla, y mayor sobrecarga producirá.

Con objeto de compararlos consideraremos un proceso que dispone de cuatro marcos para cargar sus páginas y que realiza la siguiente cadena de referencias a páginas:

3 1 4 2 3 1 5 3 1 4

#### Algoritmo óptimo

Selecciona para su sustitución la página que tardará más tiempo en volver a ser referenciada, es decir, aquella que no se usará durante un período de tiempo

más largo. Se debe a Belady [Bela66], y es la política que garantiza la tasa de fallo más baja para un número fijo de marcos.

Un ejemplo de utilización de este algoritmo aparece en la tabla 8.2. En ella podemos ver la página que referencia el proceso en cada instante, así como el contenido de los marcos asignados, resaltándose en negrita la última página que entra. Se indican también, las páginas que entran y salen cuando se produce un fallo de página. En este ejemplo se producen 5 fallos de página.

Ref.	3	1	4	2	3	1	5	3	1	4
Marco 0	<b>3</b>	3	3	3	3	3	3	3	3	3
Marco 1	-	1	1	1	1	1	1	1	1	1
Marco 2	-	-	4	4	4	4	4	4	4	4
Marco 3	-	-	-	<b>2</b>	2	2	<b>5</b>	5	5	5
Fallo	✓	✓	✓	✓			✓			
Entra	3	1	4	2			5			
Sale	-	-	-	-			2			

Tabla 8.2: Comportamiento del algoritmo óptimo

Este algoritmo resulta imposible de implementar porque requiere que el sistema operativo tenga un conocimiento exacto de las páginas que se van a referenciar en un futuro. Sin embargo, posee un interés teórico, al servir como estándar para comparar el resto de algoritmos.

### Algoritmo FIFO

Como su nombre indica saca de la memoria aquella página que llegó en primer lugar, es decir, la página más antigua en memoria. Un ejemplo de funcionamiento del algoritmo aparece en la tabla 8.3. En ella el símbolo  $\rightarrow$  denota la página más antigua, que es la candidata a salir. En este caso se producen 8 fallos de página.

Ref.	3	1	4	2	3	1	5	3	1	4
Marco 0	$\rightarrow$ 3	<b>5</b>	5	5	$\rightarrow$ 5					
Marco 1	-	1	1	1	1	1	$\rightarrow$ 1	<b>3</b>	3	3
Marco 2	-	-	4	4	4	4	4	$\rightarrow$ 4	1	1
Marco 3	-	-	-	<b>2</b>	2	2	2	2	$\rightarrow$ 2	4
Fallo	✓	✓	✓	✓			✓	✓	✓	✓
Entra	3	1	4	2			5	3	1	4
Sale	-	-	-	-			3	1	4	2

Tabla 8.3: Comportamiento del algoritmo FIFO

Su ventaja principal es su facilidad de implementación. Sólo necesita una lista de  $M$  elementos (tantos como marcos de memoria), con un puntero que apunte a la primera página (la más antigua) y otro a la última (la más reciente). Cuando se produce un fallo de página, se elimina la primera y se añade la nueva al final de la lista.

Un modo alternativo de implementarlo es con una cola circular y un solo puntero que apunte al principio de ésta. Cuando se produce un fallo de página, se elimina de memoria la página apuntada por el puntero, y se sitúa en su lugar el nuevo elemento. En ese momento, el puntero se incrementa, apuntando al elemento más antiguo.

El principio en el que se basa este algoritmo es que una página que lleve mucho tiempo en memoria puede que ya no se necesite. Sin embargo, este razonamiento es incorrecto, porque hay páginas que son usadas durante todo el tiempo de ejecución del proceso. De este modo pueden salir de memoria páginas que todavía se necesitan.

Ref.	2	3	4	5	2	3	6	2	3	4	5	6
Marco 0	→2	→2	→2	5	5	→5	6	6	6	6	→6	→6
Marco 1	-	3	3	→3	2	2	→2	→2	→2	4	4	4
Marco 2	-	-	4	4	→4	3	3	3	3	→3	5	5
Fallo	✓	✓	✓	✓	✓	✓	✓			✓	✓	
Entra	2	3	4	5	2	3	6			4	5	
Sale	-	-	-	2	3	4	5			2	3	

(a) Comportamiento del algoritmo FIFO con 3 marcos

Ref.	2	3	4	5	2	3	6	2	3	4	5	6
Marco 0	→2	→2	→2	→2	→2	→2	6	6	6	→6	5	5
Marco 1	-	3	3	3	3	3	→3	2	2	2	→2	6
Marco 2	-	-	4	4	4	4	4	→4	3	3	3	→3
Marco 3	-	-	-	5	5	5	5	5	→5	4	4	4
Fallo	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓
Entra	2	3	4	5			6	2	3	4	5	6
Sale	-	-	-	-			2	3	4	5	6	2

(b) Comportamiento del algoritmo FIFO con 4 marcos

Tabla 8.4: Anomalía de Belady

Como se vió en la figura 8.6 al aumentar el número de marcos asignados a un proceso debería disminuir su tasa de fallos de página. Sin embargo, algunos algoritmos presentan lo que se conoce como **anomalía de Belady**, por la cual la tasa

de fallos de página puede aumentar al hacerlo el número de marcos asignados. El algoritmo FIFO presenta esta anomalía. En la tabla 8.4(a) podemos observar que cuando el proceso dispone de 3 marcos se producen 9 fallos de páginas, mientras que cuando se le asignan 4 marcos (tabla 8.4(b)) el número de fallos sube a 10.

### Algoritmo LRU

El algoritmo LRU<sup>3</sup> reemplaza la página que hace más tiempo que no se utiliza, es decir, aquella que no ha sido referenciada desde hace más tiempo. Se basa en el principio de localidad, ya que la página que hace más tiempo que no se utiliza es la que tiene menor probabilidad de volver a ser referenciada. En la práctica el comportamiento de este algoritmo se aproxima mucho al óptimo.

Forma parte de los denominados **algoritmos de pila**, que se caracterizan porque no presentan la anomalía de Belady. Para estos algoritmos se cumple que el conjunto de páginas en memoria para  $N$  marcos es siempre un subconjunto de las páginas que estarían en memoria con  $N + 1$  marcos. Para LRU con  $N$  marcos, el conjunto de páginas en memoria serían las  $N$  páginas referenciadas más recientemente. Si el número de marcos es  $N + 1$ , tendríamos en memoria la últimas  $N + 1$  páginas referenciadas.

Ref.	3	1	4	2	3	1	5	3	1	4
Marco 0	3	3	3	3	3	3	3	3	3	3
Marco 1	-	1	1	1	1	1	1	1	1	1
Marco 2	-	-	4	4	4	4	5	5	5	5
Marco 3	-	-	-	2	2	2	2	2	2	4
Fallo	✓	✓	✓	✓			✓			✓
Entra	3	1	4	2			5			4
Sale	-	-	-	-			4			2
Pila	3	1	4	2	3	1	5	3	1	4
	-	3	1	4	2	3	1	5	3	1
	-	-	3	1	4	2	3	1	5	3
	-	-	-	3	1	4	2	2	2	5

Tabla 8.5: Comportamiento del algoritmo LRU

Un ejemplo se aprecia en la tabla 8.5, en la que se muestra una pila en la que se mantienen en orden las páginas referenciadas que residen en memoria, situándose en la cima la página referenciada más recientemente y en la base la página que hace más tiempo que no se utiliza. Cuando se hace referencia a una página que reside en memoria, se saca de la pila y se coloca en la cima de ésta. Cuando se produce un fallo de página, ésta se sitúa en la cima y se saca la que está en la base.

<sup>3</sup>Siglas correspondientes a su denominación inglesa *Less Recently Used*.

El número de fallos de página que se produce es 6. Este algoritmo da resultados mucho mejores que FIFO y parecidos al óptimo.

El problema que presenta este algoritmo es la dificultad de implementación de la pila. La mejor forma de implementarla es utilizar una lista doblemente enlazada con punteros a la cima y a la base. Así, en el peor de los casos habrá que modificar seis punteros para sacar una página y colocarla en la cima de la pila. Las actualizaciones son costosas, pero no es necesario buscar la página a reemplazar, puesto que el puntero a la base de la pila apunta a ella.

Este algoritmo se comporta como un FIFO en el caso de que cada página sea referenciada sólo una vez, pues no van a alterar sus posiciones dentro de la pila.

### Algoritmo del reloj

Dada la sobrecarga que produce el mantenimiento de la pila en el algoritmo LRU, pese a sus buenos resultados, y el bajo rendimiento del algoritmo FIFO, se han desarrollado nuevos algoritmos. Éstos intentan obtener buenos rendimientos con poca sobrecarga.

Algunos de estos algoritmos hacen uso del soporte hardware para facilitar su implementación y producir menos sobrecarga. Los más conocidos son el **algoritmo del reloj** o **de la segunda oportunidad** y algunas de sus variantes.

El algoritmo del reloj hace uso de un bit que se asocia a cada marco y que se denomina bit de referencia o de uso. Éste se utiliza para saber si una página ha sido referenciada recientemente. Cuando la página se carga en memoria, se pone a uno. Si posteriormente se hace referencia a ella, también se pone a uno (si no lo está ya). A la hora de elegir una página, se seleccionará una con el bit a cero, con objeto de deshacerse de las páginas de uso menos frecuente.

#### Algoritmo 8.1

#### Algoritmo del reloj

```
mientras no seleccione página a ser reemplazada
hacer
    si el puntero está en una página con bit de referencia a 0
        entonces
            La página es seleccionada para ser reemplazada
            Avanzar puntero
        si no
            Poner el bit de referencia a 0
            Avanzar puntero
    fin si
fin mientras
```

El funcionamiento de este algoritmo es similar al FIFO, pero le da a cada página una segunda oportunidad. Considera el conjunto de páginas candidatas a la sustitución como una lista circular y cuando hay que sustituir una, recorre la lista hasta encontrar una página con su bit de referencia a cero. Cada vez que pasa por una con el bit a uno, lo pone a cero. En el caso de que todas las páginas tengan su bit a uno, se dará una vuelta completa a toda la lista poniendo los bits a cero, parándose en la posición original y escogiendo esa página para su sustitución. En este caso, se comporta como FIFO. El algoritmo 8.1 muestra este comportamiento.

Un ejemplo se ve en la tabla 8.6, donde aparece el bit de referencia como superíndice de la página. En este caso se producen 8 fallos de página.

Ref.	3	1	4	2	3	1	5	3	1	4
Marco 0	→3 <sup>1</sup>	5 <sup>1</sup>	5 <sup>1</sup>	5 <sup>1</sup>	→5 <sup>1</sup>					
Marco 1	-	1 <sup>1</sup>	→1 <sup>0</sup>	3 <sup>1</sup>	3 <sup>1</sup>	3 <sup>1</sup>				
Marco 2	-	-	4 <sup>1</sup>	4 <sup>1</sup>	4 <sup>1</sup>	4 <sup>1</sup>	4 <sup>0</sup>	→4 <sup>0</sup>	1 <sup>1</sup>	1 <sup>1</sup>
Marco 3	-	-	-	2 <sup>1</sup>	2 <sup>1</sup>	2 <sup>1</sup>	2 <sup>0</sup>	2 <sup>0</sup>	→2 <sup>0</sup>	4 <sup>1</sup>
Fallo	✓	✓	✓	✓			✓	✓	✓	✓
Entra	3	1	4	2			5	3	1	4
Sale	-	-	-	-			3	1	4	2

**Tabla 8.6:** Comportamiento del algoritmo del reloj

### Algoritmo del reloj mejorado

Si se aumenta el número de bits empleados se obtienen distintas variantes del algoritmo del reloj que pueden obtener mejores resultados. Una posibilidad consiste en hacer uso del bit de modificación que se asocia a cada página en memoria en los sistemas de paginación. El empleo de éste junto con el de referencia permite clasificar las páginas en cuatro tipos:

**Página (0,0)** Es una página que no ha sido referenciada recientemente y que su contenido no se ha modificado desde que se cargó en memoria.

**Página (0,1)** No ha sido referenciada recientemente pero su contenido se ha modificado durante su estancia en memoria.

**Página (1,0)** Se ha referenciado recientemente pero no ha sido modificada.

**Página (1,1)** Aquella que ha sido referenciada y modificada.

El algoritmo del reloj mejorado, que vamos a describir a continuación, hace uso de estos dos bits de la siguiente forma:

Paso 1 En primer lugar busca páginas de tipo (0,0) y si encuentra alguna la sustituye.

Paso 2 Si no ha encontrado ninguna, recorre de nuevo la lista circular buscando páginas (0,1), seleccionando la primera que encuentre. Durante esta búsqueda, modifica el bit de referencia de aquellas páginas que lo tienen a uno.

Paso 3 Si todavía no ha encontrado ninguna, vuelve al paso 1, ya que todas las páginas habrán cambiado de categoría.

El objetivo que persigue este algoritmo es doble, seleccionar aquella página que tenga menos posibilidades de ser referenciada de nuevo y, además, que su sustitución no conlleve una operación de escritura en el área de intercambio. Si no encuentra ninguna, volvería a rastrear la lista circular descartando el segundo objetivo. Esto implica que se pueden dar varias vueltas a la lista hasta encontrar la página a sustituir.

Un ejemplo se aprecia en la tabla 8.7, en la que aparece como superíndice el bit de referencia y como subíndice el bit de modificación. El número de fallos de página que se produce es 6, siendo menor que en la versión original del algoritmo.

Ref.	$3^w$	$1^w$	4	2	3	1	5	3	1	4
Marco 0	$\rightarrow 3_1^1$	$3_0^0$	$3_1^1$	$3_1^1$	$\rightarrow 3_1^1$					
Marco 1	-	$1_1^1$	$1_1^1$	$1_1^1$	$1_1^1$	$1_1^1$	$1_1^0$	$1_1^0$	$1_1^1$	$1_1^1$
Marco 2	-	-	$4_0^1$	$4_0^1$	$4_0^1$	$4_0^1$	$5_0^1$	$5_0^1$	$5_0^1$	$5_0^1$
Marco 3	-	-	-	$2_0^1$	$2_0^1$	$2_0^1$	$\rightarrow 2_0^0$	$\rightarrow 2_0^0$	$\rightarrow 2_0^0$	$4_0^1$
Fallo	✓	✓	✓	✓			✓			✓
Entra	3	1	4	2			5			4
Sale	-	-	-	-			4			2

Tabla 8.7: Comportamiento del reloj mejorado

## Otros algoritmos

Además de los vistos anteriormente, existen otros como el LFU<sup>4</sup> y el MFU<sup>5</sup>, que consisten en seleccionar la página que menos y más veces se ha usado, respec-

<sup>4</sup>Siglas correspondientes a su denominación inglesa *Less Frequently Used*.

<sup>5</sup>Siglas correspondientes a su denominación inglesa *Most Frequently Used*.

tivamente. Estos algoritmos emplean un contador por cada página, que indica el número de veces que ha sido referenciada. En cada caso, se escogerá la que tenga el menor y el mayor valor, respectivamente. La política LFU se basa en que si la página presenta un valor grande significa que se usa mucho por lo que no debe reemplazarse. Por otro lado, el algoritmo MFU supone que una página con un valor pequeño significa que tiene que ser referenciada en un futuro próximo, por lo que no debe descargarse.

### 8.5.3.3 Almacenamiento intermedio de páginas

Los algoritmos de sustitución introducen una sobrecarga grande intentando escoger la página más adecuada. Otro enfoque posible sería la utilización de algoritmos simples, que provoquen menos sobrecarga, haciendo que una mala elección de la página a sustituir no sea tan decisiva para el sistema. Esto es lo que se intenta con el **almacenamiento intermedio de páginas**.

Éste establece una zona de memoria que se utiliza para mantener temporalmente las páginas seleccionadas por el algoritmo de sustitución. De este modo, cuando se selecciona una página, ésta no sale realmente de memoria, sino que se mantiene durante un tiempo en este almacenamiento intermedio y se modifica su entrada en la tabla de páginas. Así, si la página vuelve a ser referenciada no tiene que ser traída a memoria de nuevo, puesto que no ha salido de ella.

El sistema Mach nos puede servir como ejemplo. Éste utiliza el algoritmo FIFO y para mejorar el rendimiento, las páginas que van a ser reemplazadas, no se pierden, sino que se mantienen en una lista de páginas libres.

La figura 8.8 muestra el funcionamiento de este esquema en un sistema con seis marcos, donde el almacenamiento intermedio está formado por dos marcos. En los pasos (b) y (c) las páginas sustituidas pasan al almacenamiento intermedio. En el paso (d) se vuelve a referenciar una página sustituida, pero que todavía se encuentra en el almacenamiento intermedio. De este modo es devuelta al conjunto residente con un coste bajo, puesto que no hay que acceder al disco. En el paso (e) hay que introducir una nueva página, por lo que sale de la memoria una que se encontraba en el almacenamiento intermedio, y la página seleccionada por el algoritmo de sustitución pasa a éste.

El sistema LINUX también hace uso de esta técnica; su estudio lo haremos en el apartado 8.6.7. También la emplea el sistema VAX/VMS, pero empleando dos listas, una para las páginas que no han sido modificadas y otra para las que sí lo han sido. El uso de estas listas permite escribir las páginas modificadas en bloques y sólo cuando sea necesario. Esto reduce significativamente el número de

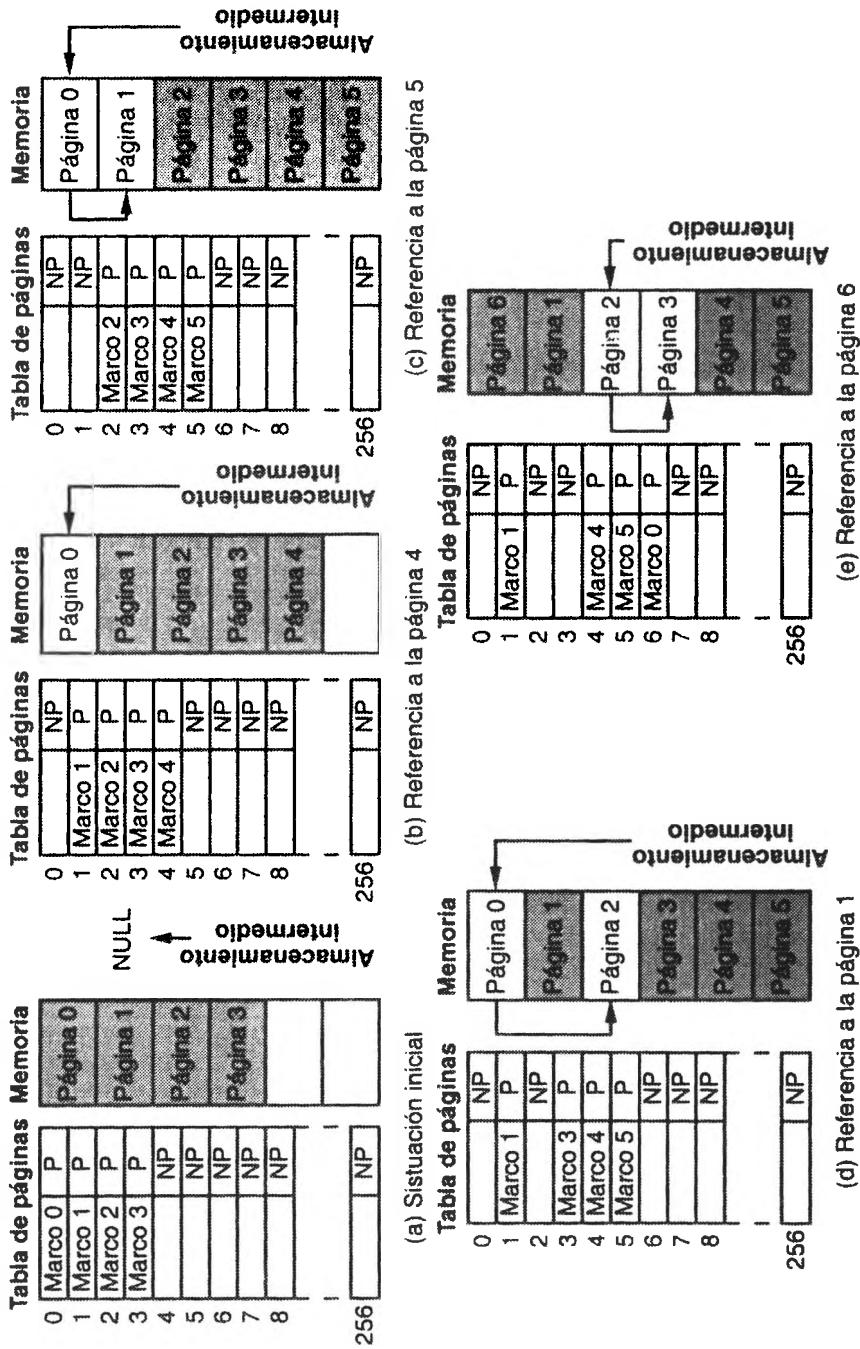


Figura 8.8: Funcionamiento del almacenamiento intermedio de páginas

operaciones de E/S y, por tanto, la cantidad de tiempo de acceso al disco.

### 8.5.4 Gestión del conjunto residente

En el apartado 8.5.3 se comentó que en la sustitución de páginas se encuentran involucrados varios aspectos interrelacionados, el tamaño del conjunto residente, el alcance de la sustitución y la estrategia de sustitución. Esta sección la vamos a dedicar al estudio conjunto de los dos primeros aspectos, conocido como gestión del conjunto residente.

#### 8.5.4.1 Tamaño del conjunto residente

Uno de los aspectos de diseño del gestor de memoria virtual es decidir el número de marcos que se asignará a cada proceso, es decir, el tamaño del conjunto residente. Existen dos alternativas, la **asignación fija** y la **asignación variable**.

Con la primera, el sistema proporciona a cada proceso un número fijo de marcos para su ejecución. Este número se establece en el momento de la creación del proceso, según el tipo, prioridad, tamaño, etc. Se ha de tener en cuenta que cuanto menor sea la cantidad de memoria que se le asigne al proceso, mayor será el grado de multiprogramación del sistema. Sin embargo, según el principio de localidad, si el tamaño del conjunto residente de un proceso es demasiado pequeño su tasa de fallos de página puede aumentar considerablemente, pudiendo provocar una situación conocida como **hiperpaginación**. Ésta se caracteriza por una caída del rendimiento del sistema. Este concepto se estudiará más detenidamente en el apartado 8.5.6. Por otro lado, estudios del comportamiento de los procesos han permitido comprobar que la asignación de más memoria a un proceso por encima de un determinado valor, no tiene un efecto significativo sobre su tasa de fallos de página (véase la figura 8.6).

La estrategia de asignación variable establece conjuntos residentes de tamaño variable, que se adaptan a las necesidades de ejecución de los procesos. De este modo, un proceso que produce muchos fallos de página verá incrementado el número de marcos que tiene asignados. Mientras que aquellos que producen pocos fallos de página podrán reducir el tamaño de su conjunto residente sin incidir de forma negativa en su rendimiento.

Según las afirmaciones anteriores, la política de asignación variable parece ser la más flexible y adecuada. Sin embargo, requiere que el sistema operativo valore las necesidades de los procesos produciendo una sobrecarga en el sistema.

#### 8.5.4.2 Alcance de la sustitución

Otro de los aspectos de diseño del gestor de memoria virtual es decidir cuál va a ser el conjunto de páginas candidatas a ser sustituidas cuando se produce un fallo de página y no hay marcos libres. Esto es lo que se conoce como alcance de la sustitución. Éste puede ser **global** o **local**.

Una estrategia de alcance local sólo elige entre las páginas residentes del proceso que provoca el fallo de página, mientras que una política de alcance global puede elegir entre todas las que se encuentran en memoria, independientemente del proceso al que pertenezcan.

Una restricción aplicable a estas políticas es la existencia de páginas que no pueden ser sacadas de memoria ya que contienen una información especial. La mayor parte del núcleo del sistema operativo, así como estructuras de control claves para el funcionamiento del sistema, *buffers* de E/S, etc., son ejemplos de éstas. Estas páginas se encuentran en marcos bloqueados (ver apartado 8.4.1).

El alcance de la sustitución y el tamaño del conjunto residente son conceptos que están relacionados, dando lugar a diferentes esquemas. A continuación veremos las combinaciones posibles.

#### Asignación fija, alcance local

Cuando se emplea una política de asignación fija, cada vez que se necesita aplicar un algoritmo de sustitución, el conjunto de páginas candidatas se reduce a las del propio proceso. En este sistema es necesario establecer tanto el algoritmo de sustitución que se va a emplear como el tamaño del conjunto residente de cada proceso.

#### Asignación variable, alcance global

En estos sistemas los tamaños de los conjuntos residentes de los procesos se van adaptando a sus necesidades. Así, cuando un proceso produce un fallo de página, si existen marcos libres se le añade a su conjunto residente; si no los hay, se aplicará el algoritmo de sustitución a todas las páginas residentes en memoria que no estén bloqueadas. De este modo, un proceso que produzca muchos fallos de página aumentará el tamaño de su conjunto residente a costa de disminuir el de otros procesos.

Este esquema es uno de los más empleados en los sistemas actuales debido a su facilidad de implementación, ya que sólo necesita establecer el algoritmo de sustitución. Sin embargo, el inconveniente que plantea es que podría seleccionarse para su sustitución una página de un proceso al que no conviene reducir su

conjunto residente. Esto puede provocar que el sistema pase a una situación de hiperpaginación.

Una forma de contrarrestar este problema sería la utilización del almacenamiento intermedio de páginas (ver apartado 8.5.3.3).

### **Asignación variable, alcance local**

Esta estrategia también intenta adaptar el tamaño del conjunto residente a las necesidades del proceso, pero sin incurrir en los problemas de hiperpaginación que puede plantear el alcance global. Para ello, se comporta como una estrategia de asignación fija y alcance local a intervalos, es decir, inicialmente asigna un número de marcos a los procesos y durante un cierto período de tiempo emplea alcance local. Pasado un tiempo ajusta el tamaño del conjunto residente a las necesidades del proceso, según haya sido su comportamiento en el período anterior. Así, si un proceso ha sufrido muchos fallos de página aumentará su conjunto residente y si ha sufrido muy pocos, éste podrá disminuir. Esta evaluación periódica hace que esta estrategia sea más compleja que la de alcance global, pero ofrece un mejor rendimiento.

En este sistema hay que determinar el tamaño del conjunto residente de cada proceso, los intervalos de cambio y los criterios para modificar los tamaños de los conjuntos residentes. A continuación se estudiarán dos estrategias de este tipo, el **modelo del conjunto de trabajo y la frecuencia de fallos de página**.

**Modelo del conjunto de trabajo** Esta estrategia, introducida por Denning [Denn68], ha recibido mucha atención en la bibliografía. Se basa en el principio de localidad, según el cual los procesos, durante períodos cortos de tiempo, hacen referencia a un conjunto de direcciones agrupadas. Sin embargo, a lo largo de períodos de tiempo largos estos conjuntos de direcciones cambian. Es decir, los procesos, durante su ejecución, se mueven de una localidad a otra.

La filosofía subyacente en el modelo del conjunto de trabajo es que, en un momento dado, la cantidad de almacenamiento que requiere un proceso puede estimarse basándose en su comportamiento en el pasado reciente.

Se puede definir el conjunto de trabajo de un proceso como el conjunto de páginas a las que hace referencia activamente el proceso; según el principio de localidad antes citado, este conjunto varía a lo largo del tiempo.

Más formalmente podemos decir que el conjunto de trabajo ( $W$ ) es función del tiempo y de un parámetro  $\Delta$  que se denomina **ventana del conjunto de trabajo**; así,  $W(t, \Delta)$  es el conjunto de páginas que ha referenciado el proceso

en las últimas  $\Delta$  unidades de tiempo. Puesto que  $\Delta$  es una medida de tiempo, y estamos considerando sistemas de tiempo compartido,  $\Delta$  debe medir el tiempo de ejecución del proceso en vez del tiempo real, por lo que se habla de tiempo virtual. Podemos decir que una unidad de tiempo virtual es el tiempo que tarda una instrucción en ejecutarse.

Pasamos a considerar ahora las dos variables de las que depende  $W$ . La variable  $\Delta$  es una ventana de tiempo durante la cual se observa el comportamiento del proceso, por tanto, el tamaño del conjunto de trabajo es una función no decreciente de  $\Delta$ . De forma matemática:

$$W(t, \Delta) \subseteq W(t, \Delta + 1)$$

En la figura 8.9 podemos ver un ejemplo del funcionamiento de esta estrategia para la ejecución de un proceso con distintos valores de  $\Delta$ . En cada celda se muestra el conjunto de trabajo del proceso, estando ordenadas las páginas por el instante de su última referencia. Así, la página situada más a la derecha es la última referenciada. De este modo, cuando se produce un fallo de página, indicado con el símbolo  $\checkmark$ , la página que se sustituye es la que se encuentra situada a la izquierda.

Secuencia de referencias a páginas	Tamaño de la ventana, $\Delta$			
	2	3	4	5
12	12 $\checkmark$	12 $\checkmark$	12 $\checkmark$	12 $\checkmark$
27	12 27 $\checkmark$	12 27 $\checkmark$	12 27 $\checkmark$	12 27 $\checkmark$
33	27 33 $\checkmark$	12 27 33 $\checkmark$	12 27 33 $\checkmark$	12 27 33 $\checkmark$
16	33 16 $\checkmark$	27 33 16 $\checkmark$	12 27 33 16 $\checkmark$	12 27 33 16 $\checkmark$
12	16 12 $\checkmark$	33 16 12 $\checkmark$	27 33 16 12	27 33 16 12
①44	12 44 $\checkmark$	16 12 44 $\checkmark$	33 16 12 44 $\checkmark$	27 33 16 12 44 $\checkmark$
33	44 33 $\checkmark$	12 44 33 $\checkmark$	16 12 44 33	16 12 44 33
12	33 12 $\checkmark$	44 33 12	44 33 12	16 44 33 12
33	12 33	44 12 33	44 12 33	44 12 33
44	33 44 $\checkmark$	12 33 44	12 33 44	12 33 44
44	44	33 44	12 33 44	12 33 44
②44	44	44	33 44	12 33 44
12	44 12 $\checkmark$	44 12 $\checkmark$	44 12 $\checkmark$	33 44 12
27	12 27 $\checkmark$	44 12 27 $\checkmark$	44 12 27 $\checkmark$	44 12 27 $\checkmark$
12	27 12	27 12	44 27 12	44 27 12
33	12 33 $\checkmark$	27 12 33 $\checkmark$	27 12 33 $\checkmark$	44 27 12 33 $\checkmark$

Figura 8.9: Conjuntos de trabajo de un proceso para distintos  $\Delta$

Se puede observar que cuanto mayor es el tamaño de la ventana, mayor es

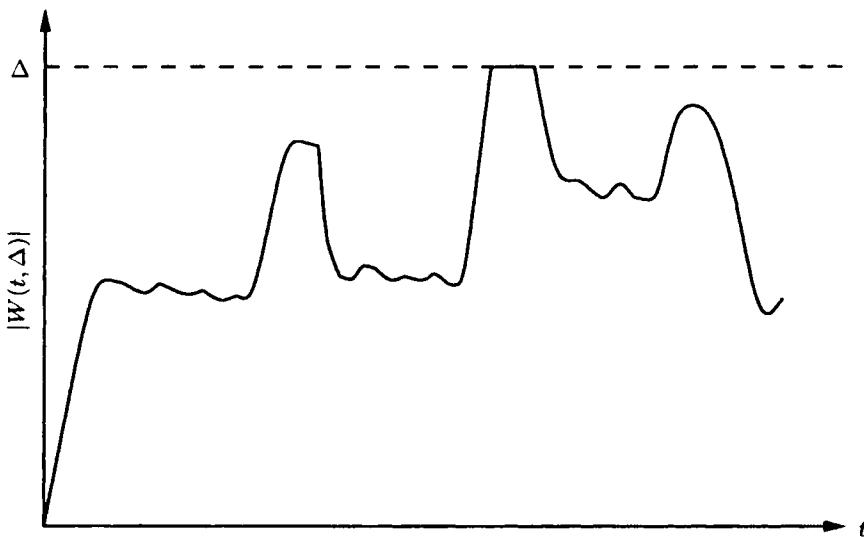
también el conjunto de trabajo. Así, cuando se referencia la página 44 por primera vez (indicado como ①) alcanza el tamaño máximo de la ventana.

Hay que destacar que la elección del tamaño de la ventana influye en la tasa de fallos de página. A medida que aumenta  $\Delta$ , el número de fallos de página disminuye.

La otra variable de la que depende el conjunto de trabajo es el tiempo. En la figura 8.9 se puede observar que para un valor de  $\Delta$  fijo, el conjunto de trabajo va variando no sólo de contenido, sino también de tamaño. De este modo, cuando un proceso hace referencia a páginas que se encuentran dentro del conjunto su tamaño puede mantenerse o, incluso, disminuir. Así, en la última referencia a la página 44 (indicado como ②), el conjunto de trabajo sólo contiene a ésta para  $\Delta = 2$  y  $\Delta = 3$ . De forma matemática esta relación se puede expresar:

$$1 \leq |W(t, \Delta)| \leq \min\{\Delta, N\}$$

donde  $N$  es el número de páginas del proceso.



**Figura 8.10:** Evolución del conjunto de trabajo con el tiempo

Esta relación se puede observar en la figura 8.10 en la que se muestra cómo varía el tamaño del conjunto de trabajo a lo largo del tiempo para un valor fijo de  $\Delta$ . En ella se alternan períodos donde el tamaño del conjunto de trabajo

es relativamente estable con otros donde se producen cambios rápidos. Cuando un proceso empieza a ejecutarse, el tamaño del conjunto de trabajo aumenta rápidamente hasta alcanzar la localidad de referencias, donde se estabiliza. Pasado cierto tiempo, se produce un aumento brusco del tamaño del conjunto de trabajo seguido de un descenso rápido. Esto se debe a que el proceso cambia de localidad, por lo que su conjunto de trabajo irá admitiendo inicialmente páginas de la nueva localidad para posteriormente liberar las de la antigua.

La importancia de los conjuntos de trabajo es debida a que relacionan la administración de la memoria y de los procesos a través del **Principio del conjunto de trabajo**, que además se puede utilizar como una estrategia guía para la sustitución de páginas. Este principio dice:

1. Un proceso debe ejecutarse sólo si su conjunto de trabajo reside en memoria principal; es decir, si su conjunto residente incluye a su conjunto de trabajo.
2. Una página no debe ser retirada de memoria principal si es miembro del conjunto de trabajo de un proceso en ejecución.

Por tanto, con respecto al tamaño del conjunto residente, el número de páginas asignadas a cada proceso debería coincidir con el tamaño del conjunto de trabajo, es decir, el número de páginas distintas contenidas en éste. De esta forma evita la hiperpaginación, porque no permitiría ejecutar un proceso que no contara con su conjunto de trabajo en memoria. Con respecto a la sustitución de páginas, nos indica que una página debería ser sustituida sólo si no pertenece al conjunto de trabajo del proceso, ya que en caso contrario es probable que la tasa de fallos de página del proceso aumente, puesto que se está retirando una página con grandes posibilidades de ser referenciada en un futuro cercano.

Aunque conceptualmente es muy atractiva, la estrategia del conjunto de trabajo es difícil de implementar. El programa 8.1 muestra una posible codificación de este algoritmo. Cada proceso posee un vector con las páginas que pertenecen a su conjunto de trabajo. La situada en la posición cero es la página que tiene que salir, y la situada en la última (`TAM_CONJUNTO - 1`) es la referenciada más recientemente. La entrada en la tabla de páginas almacena, para cada una, un contador que indica el número de veces que ésta aparece en el conjunto de trabajo. De este modo, cuando una página sale del conjunto el contador decrece en una unidad; si llega al valor de cero es cuando la página sale de memoria.

Uno de los problemas que plantea es la estimación del tamaño apropiado de la ventana  $\Delta$  (la variable `TAM_CONJUNTO`), que es el parámetro crucial de esta política. Si se elige un valor demasiado pequeño, el conjunto de trabajo puede no englobar todas las páginas de la localidad, con el aumento consiguiente de la

tasa de fallos de página. Por el contrario, si se elige un valor demasiado grande, puede contener páginas de más de una localidad, con lo que se está desperdiciando memoria y el grado de multiprogramación será menor.

Programa 8.1	Conjunto de trabajo
<pre>#include "memoria.h"  /* Ventana del conjunto de trabajo */ extern int TAM_CONJUNTO;  void conjunto_trabajo(int proceso, int pagina_entra) {     int i;     int pagina_sale;     /* seleccionar página que sale del conjunto de trabajo */     pagina_sale = ConjuntoTrabajo[proceso][0];      /* actualizar conjunto de trabajo */     for (i=0; i &lt; TAM_CONJUNTO - 1; i++)         ConjuntoTrabajo[proceso][i] = ConjuntoTrabajo[proceso][i+1];      ConjuntoTrabajo[proceso][TAM_CONJUNTO - 1] = pagina_entra;      /* actualizar contador de referencias en tabla de páginas */     PTBR[pagina_entra].contador++;     PTBR[pagina_sale].contador--;      /* comprobar si sale la página de memoria */     if (PTBR[pagina_sale].contador == 0) {         sacar_página(proceso, pagina_sale);         PTBR[pagina_sale].presente = FALSE;     }      /* comprobar si página que entra estaba en conjunto de trabajo */     if (PTBR[pagina_entra].presente == FALSE) {         fallo_página(proceso, pagina_entra); /* fallo de página */         PTBR[pagina_entra].presente = TRUE;         PTBR[pagina_entra].contador = 1;     } }</pre>	

Un problema aún más serio es que la medida del conjunto de trabajo de cada proceso provoca una sobrecarga de trabajo en el sistema, ya que sería necesario llevar el control de qué página referencia cada proceso en cada momento. En el programa 8.1 esto se realiza mediante el empleo de un vector que mantiene las páginas referenciadas por el proceso de forma ordenada; por tanto, debe actualizarse

siempre que el proceso realice una referencia a memoria.

Otra posibilidad sería mantener una lista de páginas referenciadas, ordenada por tiempo. Una forma de hacer más eficiente esta estrategia es emplear un hardware hecho a la medida. Como ejemplo podemos citar el computador Maniac II [Morr72], que contaba con un registro de conjunto de trabajo, un registro contador asociado a cada marco de memoria y un registro-T. El registro de conjunto de trabajo es un mapa de bits con un bit por marco, que identifica cuáles contienen páginas del proceso en ejecución. Los registros contadores se incrementan periódicamente siempre que el marco pertenezca al conjunto de trabajo del proceso que se está ejecutando. El registro-T controla la frecuencia de esta actualización. Cuando un contador de un marco se desborda, se establece un bit de alarma en el registro del marco para indicar que la página ya no pertenece al conjunto de trabajo del proceso. Cuando se produce un fallo de página, los marcos que tienen el bit de alarma establecido indican páginas candidatas a ser sacadas de la memoria. Cada vez que se referencia una página, el contador del marco se pone a cero.

**Frecuencia de fallos de página** Debido a los problemas que plantea la implementación de la estrategia del conjunto de trabajo, han aparecido otras que intentan aproximarse a ella. Una de las más conocidas es la de **frecuencia de fallos de página** que utiliza la tasa de fallos de página para controlar el tamaño del conjunto residente. Este algoritmo utiliza el bit de referencia de la tabla de páginas. Se establece un umbral de tiempo  $F$ , que indica el tiempo<sup>6</sup> que debería de transcurrir entre dos fallos de página. Este umbral es el inverso de la tasa de fallos de página.

El programa 8.2 muestra una posible implementación de este algoritmo. Cada proceso dispone de un contador para almacenar el tiempo transcurrido desde el último fallo de página (*tiempo*). Cuando se produce un fallo de página, el sistema operativo calcula el tiempo transcurrido desde el anterior para ese proceso. Si éste es menor que el umbral, se añade la página al conjunto residente del proceso. En caso contrario, se descartan aquellas páginas con su bit de referencia a cero. Al mismo tiempo, restablece el bit de referencia de las páginas restantes del proceso a cero.

El razonamiento que sigue esta estrategia es el siguiente, si la tasa de fallos de página de un proceso está por debajo de un valor dado, se le podría disminuir el tamaño de su conjunto residente sin incidir notablemente en su tasa de fallos de página. Por el contrario, si la tasa de fallos de página de un proceso está por

<sup>6</sup>Puesto que estamos considerando un sistema de tiempo compartido, éste sería un tiempo virtual, es decir, cada unidad de tiempo equivale a la ejecución de una instrucción.

encima de un valor umbral, se debería aumentar el número de marcos asignados para disminuir su tasa de fallos de página. De este modo, la política de frecuencia de fallos de página garantiza que el conjunto residente crece cuando los fallos de página son frecuentes, y disminuye cuando la tasa de fallos de página baja.

**Programa 8.2****Frecuencia de fallos de página**

```
#include "memoria.h"

/* vector con los tiempos virtuales (referencias a memoria) de cada proceso */
extern int tiempo[MAX_PROC];

/* variable global con el umbral de tiempo definido para disminuir */
/* el conjunto de páginas de un proceso */
extern int UMBRAL;

void frecuencia_fallo_pagina (int proceso, int pagina)
{
    int pagina_sale, i;

    tiempo[proceso]++; /* variable global */

    if (PTBR[pagina].presente == TRUE)
        PTBR[pagina].referencia = 1;
    else {
        fallo_pagina(proceso, pagina);
        PTBR[pagina].presente = TRUE;
        PTBR[pagina].referencia = 1;
        if (tiempo[proceso] > UMBRAL) {
            for (i=0; i < PAG_MAX; i++) {
                /* sacar páginas no referenciadas */
                if ((PTBR[i].presente) && (PTBR[i].referencia == 0)) {
                    sacar_pagina(proceso, i);
                    PTBR[i].presente = FALSE;
                }
                /* restablecer bit de referencia */
                if ((PTBR[i].presente) && (i != pagina))
                    PTBR[i].referencia = 0;
            }
            tiempo[proceso] = 0;
        }
    }
}
```

La principal ventaja que presenta el algoritmo de frecuencia de fallos de página frente al modelo del conjunto de trabajo es que el conjunto residente sólo se ajusta cuando se produce un fallo de página en vez de en cada referencia.

Más formalmente podemos decir que si  $t_c$  es el instante en el que se produce el último fallo de página, y el anterior se produjo en  $t_{c-1}$ , entonces todas las páginas no referenciadas durante el intervalo  $t_{c-1} \leq t \leq t_c$  se sacarán del conjunto residente si y sólo si  $t_c - t_{c-1} > F$ , donde  $F$  es un parámetro del sistema. En otras palabras, el conjunto de páginas residentes en el instante  $t_c$ , que se denota como  $R(t_c, F)$ , se describe como

$$R(t_c, F) = \begin{cases} Z(t_c, t_{c-1}) + P(t_c) & \text{si } t_c - t_{c-1} > F \\ R(t_{c-1}, F) + P(t_c) & \text{en cualquier otro caso} \end{cases}$$

donde  $Z(t_c, t_{c-1})$  denota el conjunto de páginas referenciadas durante el intervalo  $t_c - t_{c-1}$  y  $P(t_c)$  es la página referenciada en el instante  $t_c$  (que no pertenecía al conjunto residente).

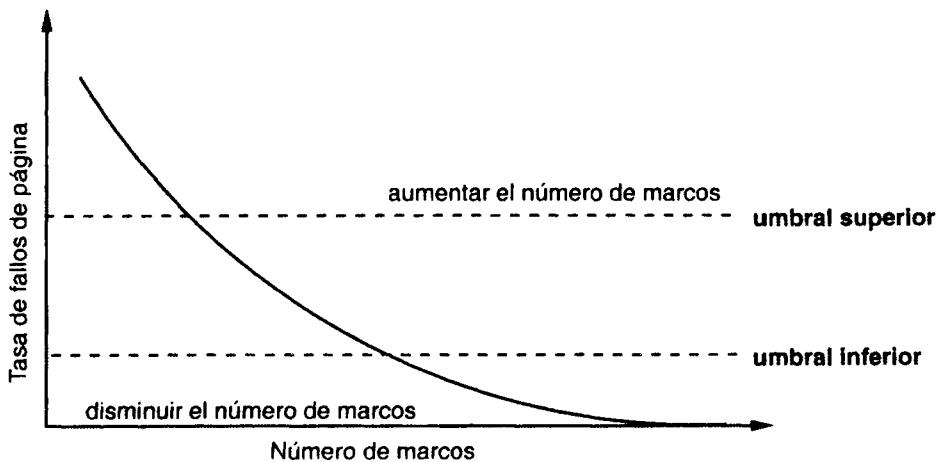
La figura 8.11 muestra un ejemplo del comportamiento de este algoritmo con un umbral  $F = 2$ . Podemos ver como va aumentando inicialmente el tamaño del conjunto residente del proceso. Cuando llega a un periodo estable donde referencia las páginas cargadas no se producen fallos. Posteriormente, cuando se producen fallos, no sólo incorpora las nuevas páginas a memoria, sino que aquellas que no han sido referenciadas se descargan.

Ref. a páginas	Conjunto residente	Fallo de página	Entra	Sale
1	1 <sup>1</sup>	✓	1	-
4	1 <sup>0</sup> 4 <sup>1</sup>	✓	4	-
5	1 <sup>0</sup> 4 <sup>0</sup> 5 <sup>1</sup>	✓	5	-
3	1 <sup>0</sup> 4 <sup>0</sup> 5 <sup>0</sup> 3 <sup>1</sup>	✓	3	-
3	1 <sup>0</sup> 4 <sup>0</sup> 5 <sup>0</sup> 3 <sup>1</sup>			
3	1 <sup>0</sup> 4 <sup>0</sup> 5 <sup>0</sup> 3 <sup>1</sup>			
4	1 <sup>0</sup> 4 <sup>1</sup> 5 <sup>0</sup> 3 <sup>1</sup>			
2	4 <sup>0</sup> 3 <sup>0</sup> 2 <sup>1</sup>	✓	2	1 5
3	4 <sup>0</sup> 3 <sup>1</sup> 2 <sup>1</sup>			
5	4 <sup>0</sup> 3 <sup>0</sup> 2 <sup>0</sup> 5 <sup>1</sup>	✓	5	-
3	4 <sup>0</sup> 3 <sup>1</sup> 2 <sup>0</sup> 5 <sup>1</sup>			
5	4 <sup>0</sup> 3 <sup>1</sup> 2 <sup>0</sup> 5 <sup>1</sup>			
1	3 <sup>0</sup> 5 <sup>0</sup> 1 <sup>1</sup>	✓	1	4 2
4	3 <sup>0</sup> 5 <sup>0</sup> 1 <sup>0</sup> 4 <sup>1</sup>	✓	4	-

Figura 8.11: Comportamiento de la frecuencia de fallos de página

Esta estrategia se puede refinar mediante el uso de dos umbrales (figura 8.12). Si el intervalo de tiempo entre dos fallos de página está por encima de un umbral

superior, aumentará el tamaño del conjunto residente; por contra, si está por debajo del umbral inferior disminuirá su tamaño.



**Figura 8.12:** Frecuencia de fallos de página

### 8.5.5 Política de limpieza

La **política de limpieza** decide cuando debe ser escrita en el área de intercambio una página que ha sido modificada. Existen varias alternativas. Una es la denominada **limpieza previa** que escribe las páginas modificadas cada cierto tiempo, independientemente de que vayan a salir o no de la memoria. La ventaja de esta técnica es que permite realizar muchas operaciones de escritura aprovechando las características de los dispositivos de almacenamiento secundario. Sin embargo, el problema que plantea es que las páginas que se escriben permanecen en memoria principal pudiendo ser modificadas nuevamente, por lo que si tienen que ser reemplazadas deberán escribirse de nuevo.

La utilización del almacenamiento intermedio (ver apartado 8.5.3.3) puede mejorar esta estrategia. Si se limpian sólo aquellas páginas que son reemplazables, es decir, las que se encuentran en el almacenamiento intermedio, disminuiríamos el número de operaciones de escritura innecesarias.

Otra alternativa, conocida como **limpieza por demanda**, consiste en escribir una página modificada en el área de intercambio sólo cuando sea seleccionada por el algoritmo de sustitución. De este modo, la escritura de una página modificada va asociada y precede a la lectura de una nueva página. La ventaja de esta

estrategia es que permite minimizar el número de operaciones de escritura en el área de intercambio. Pero presenta el inconveniente de que algunos fallos de página provoquen dos operaciones de E/S, la escritura de la página a sustituir y la lectura de la nueva.

### 8.5.6 Control de la carga y la hiperpaginación

Uno de los principales problemas de un sistema de gestión de memoria virtual es mantener un equilibrio entre el grado de multiprogramación y la tasa de fallos de página. Esta es una de las labores del gestor de memoria virtual y se conoce como **control de la carga**.

La figura 8.13 muestra cómo varía la utilización del procesador en función del grado de multiprogramación. Como se puede observar, a medida que éste crece a partir de un valor pequeño, la utilización del procesador también aumenta. Sin embargo, si aumenta por encima de un cierto valor, se observa una disminución drástica del rendimiento. Esto se debe a que al tener muchos procesos en memoria el tamaño de sus conjuntos residentes es demasiado pequeño lo que provoca una tasa muy alta de fallos de página. Esta situación se conoce como hiperpaginación.

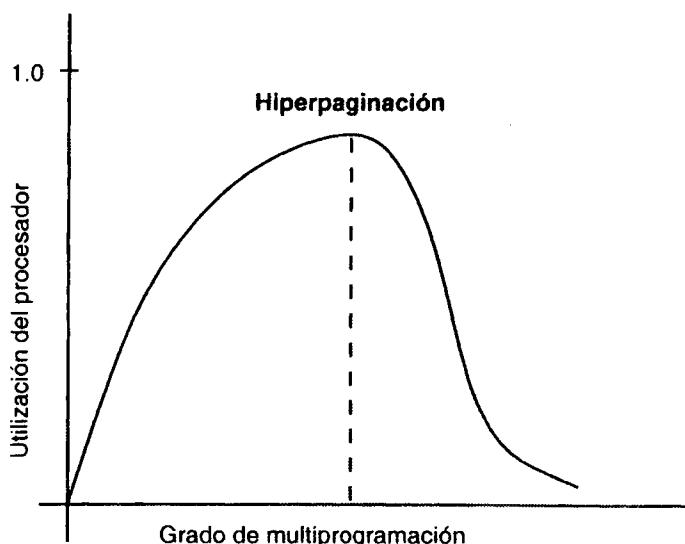


Figura 8.13: Hiperpaginación

Cuando se emplea una estrategia de gestión del conjunto residente de asignación variable y alcance local, el control de la carga viene determinado por ésta.

Por ejemplo, si empleamos el modelo del conjunto de trabajo, sólo podrán ejecutarse aquellos procesos que tengan su conjunto de trabajo en memoria, de esta forma se determina automáticamente el grado de multiprogramación. De forma matemática, la demanda total de marcos en un instante dado, vendrá dada por:

$$D = \Sigma W(t, \Delta)$$

De este modo, si la demanda total es mayor que el número total de marcos disponibles en memoria se produciría hiperpaginación. Para evitarla, el sistema sólo crea un nuevo proceso si existen marcos suficientes para albergar su conjunto de trabajo. Dado que el tamaño de los conjuntos va variando a lo largo del tiempo, si en un instante la suma de todos los conjuntos de trabajo es superior al número de marcos del sistema, habrá que suspender algún proceso, descargando sus páginas y asignándoselas a otros.

En los otros esquemas de gestión del conjunto residente hay que añadir un mecanismo de control de la carga. A la hora de implementarlo habrá que tener en cuenta dos aspectos:

1. Criterio para modificar el grado de multiprogramación.
2. Elección del proceso a intercambiar, si fuese necesario.

En cuanto al primer aspecto, existen varias posibilidades:

**Criterio L=S** ajusta el grado de multiprogramación de forma que se cumpla que el tiempo medio entre fallos de página ( $L$ ) sea igual al tiempo medio requerido para procesar un fallo de página ( $S$ ). Los estudios de rendimiento indican que éste es el punto en el cual el uso del procesador alcanza un máximo [Denn80].

**Criterio del 50%** intenta mantener un grado de multiprogramación, tal que el dispositivo de paginación esté ocupado el 50% del tiempo. De nuevo, los estudios de rendimiento indican que éste es un punto de uso máximo del procesador [Lero76].

**Algoritmo del reloj adaptado** es una modificación del algoritmo del reloj que consiste en supervisar la velocidad con la que el puntero completa una vuelta a la lista circular de marcos. Así, si se detecta que la velocidad es menor que un valor umbral, puede deberse a que se están generando pocos fallos de página o a que se encuentra fácilmente una página para su sustitución cuando se produce un fallo de página. En ambos casos, puede aumentar

el nivel de multiprogramación sin degradar el rendimiento del sistema. Si la velocidad es mayor que un cierto umbral, puede estar motivado por una alta tasa de fallos de página o por la dificultad de encontrar una página para sustituir. [Carr84] describe esta técnica para un sistema de alcance global.

Respecto a la decisión de intercambiar un proceso para reducir el grado de multiprogramación, también existen diversos criterios:

- Prioridad del proceso.
- Instante de carga del proceso en memoria, si un proceso lleva poco tiempo cargado en memoria es poco probable que tenga en ella su conjunto de trabajo.
- Número de fallos de página. Un proceso con un número elevado de fallos de página es poco probable que tenga su conjunto de trabajo en memoria.
- Tamaño del proceso. Cuanto mayor sea el proceso que se intercambia mayor número de marcos libres se obtienen.
- Tamaño del conjunto residente. Si se escoge un proceso con un tamaño de conjunto residente pequeño el esfuerzo para cargar de nuevo su conjunto de trabajo será menor.

## 8.6 Gestión de memoria en LINUX

LINUX utiliza un esquema de memoria virtual bajo paginación por demanda. El tamaño de la página dependerá de la arquitectura, así en los sistemas Alpha AXP la página es de 8 KiB, mientras que en los Intel x86 es de 4 KiB.

LINUX presenta dos modos de direccionamiento, el virtual y el físico. El primero es el que utilizan los procesos de usuario y sigue un esquema de paginación. El segundo lo utiliza el núcleo y se caracteriza porque no emplea tablas de páginas ni traducción de direcciones de ningún tipo. No es conveniente mantener al núcleo del sistema operativo en memoria virtual ya que esto supondría una disminución del rendimiento del sistema.

Cada proceso dispone de un espacio de direcciones virtuales cuyo tamaño dependerá de la arquitectura. Éste se divide en dos segmentos, uno para el código y los datos del proceso de usuario que emplea direccionamiento virtual, y otro para el núcleo con direccionamiento físico. Así, en la familia Intel x86 este espacio

es de 4 GiB ( $2^{32}$ ), que se divide en un segmento de usuario de 3 GiB y un segmento de núcleo de 1 GiB.

### 8.6.1 Regiones de memoria virtual

El administrador de la memoria en LINUX mantiene dos visiones del espacio de direcciones virtuales de un proceso, como un conjunto de regiones separadas y como un conjunto de páginas.

La primera concuerda con la visión lógica que tiene el usuario del espacio de direcciones del proceso. Cada región consiste en un subconjunto de páginas contiguas del espacio de direcciones. El sistema mantiene información sobre el rango de direcciones virtuales que ocupa la región, sus modos de acceso y los ficheros asociados. El bloque de control del proceso apunta a las distintas estructuras que representan las regiones virtuales.

Cada región puede representar una parte de la imagen ejecutable, el código, los datos inicializados, los datos no inicializados, etc. Dependiendo del tipo de región, estará almacenada en un fichero o no. Por ejemplo, en el caso del código y los datos inicializados estarán en el fichero ejecutable; los datos no inicializados no tendrán un fichero asociado.

La segunda visión se corresponde con la estructura física del espacio de direcciones del proceso dividido en páginas.

### 8.6.2 Traducción de direcciones

LINUX define un modelo de memoria independiente de la arquitectura. En él una dirección virtual se traduce a una dirección física mediante un esquema de paginación de tres niveles.

En la figura 8.14 se puede observar este proceso de traducción. La dirección virtual se divide en 4 componentes, los tres primeros hacen referencia a las tablas de páginas de primer, segundo y tercer nivel, mientras que el último es el desplazamiento.

Este esquema general se adapta a las características de las arquitecturas para las que se implementa. Así, en el caso de la familia Intel x86 emplea sólo dos niveles, mientras que el Alpha AXP emplea los tres.

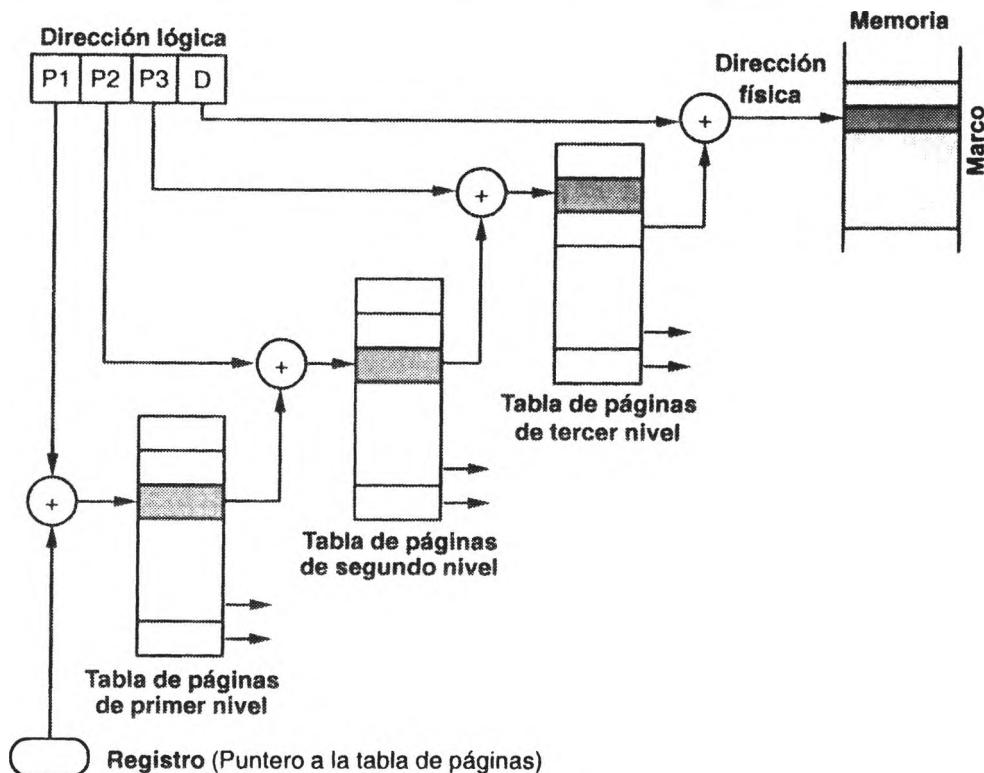


Figura 8.14: Traducción de direcciones en LINUX

### 8.6.3 Estructura de la tabla de páginas

Las entradas de la tabla de páginas son iguales independientemente del nivel. La estructura y tamaño de éstas dependen de la arquitectura. Para ilustrarlo se mostrarán dos ejemplos.

En el caso de la familia Intel x86 las entradas son de 32 bits y almacenan la siguiente información (figura 8.15):

**Bit de protección (R/W)** Controla el modo de acceso a la página.

**Bit de tipo de página (U/S)** Indica si se trata de una página de usuario o del núcleo.

**Bit de presencia (P)** Indica si la página está cargada en memoria o no.

**Bits para soportar la paginación por demanda**

**Bit de modificación (M)** Se activa cuando se modifica la página, desactivándose cuando se actualiza en memoria secundaria. Este bit no posee significado en la tabla de páginas de primer nivel.

**Bit de referencia (R)** Se activa cuando se hace referencia a la página.

**Bits del sistema operativo (SO)** Es un conjunto de bits que pueden ser empleados por el sistema operativo para implementar la política de sustitución de páginas.

**Dirección del marco** Dependiendo de si es de primer o segundo nivel corresponderá a una tabla de páginas o a una página del proceso, en el caso de que se encuentre en memoria principal.

	31 - 12	11 10 9	8	7	6	5	4	3	2	1	0
DIRECCIÓN	SO			M	R		U/S	R/W	P		

**Figura 8.15:** Entrada de la tabla de páginas en Intel x86

Cuando una página se descarga de memoria se almacena en los bits 1 a 31 su posición en memoria secundaria. El bit de presencia indica que la página no está en memoria principal.

Las entradas de las tablas de páginas en la arquitectura Alpha AXP tienen 64 bits y mantienen la siguiente información (figura 8.16):

**Bit de presencia (P)**

**Bits de fallo de página**

**Bit de fallo en lectura (FR)** Se activa cuando se produce un fallo de página al intentar leer una página que no reside en memoria.

**Bit de fallo en escritura (FW)** Se activa cuando se produce un fallo de página al intentar escribir en una página que no reside en memoria.

**Bit de fallo en ejecución (FX)** Se activa cuando se produce un fallo de página al intentar ejecutar una instrucción de una página que no reside en memoria.

**Bits para el manejo de TLB (TLB)** Conjunto de bits utilizados para la gestión de la TLB.

**Bits de protección** Se utilizan cuatro bits de protección que permiten indicar si una página puede ser leída en modo usuario (RU) o supervisor (RS), o bien, escrita en modo usuario (WU) o supervisor (WS).

### Bits para soportar la paginación por demanda

**Bit de modificación (M)**

**Bit de referencia (R)**

**Dirección del marco** Dependiendo del nivel corresponderá a una tabla de páginas o a una página del proceso. Cuando la página no está en memoria, se almacena la dirección donde se encuentra en el área de intercambio.

11 10	9	8	7	6 5 4	3	2	1	0
	RU	RS		TLB	FX	FW	FR	P
63 - 32	31 - 23	22	21	20	19 - 14	13	12	
DIRECCIÓN		R		M		WU	WS	

**Figura 8.16:** Entrada de la tabla de páginas en Alpha AXP

#### 8.6.4 Estructura de la tabla de marcos

La información más relevante que se suele almacenar en una entrada de la tabla de marcos es la siguiente:

**Bit de bloqueo** Indica si la página se encuentra implicada en una operación de E/S.

**Bit de actualización** Indica si el contenido de la página ha sido traído desde memoria secundaria con éxito.

**Bit de error** Indica si una operación de E/S ha finalizado con un error.

**Bit de referencia** Indica si la página ha sido referenciada. Lo utilizan los algoritmos de sustitución.

**Bit de modificación** Indica si la página ha sido modificada.

**Contador de uso** Muestra el número de procesos que comparten una página.

**Dirección de la página en memoria secundaria**

### 8.6.5 Gestión del espacio libre

La gestión de los marcos libres se lleva a cabo mediante una tabla. Cada elemento de ésta contiene información sobre bloques de páginas consecutivos en potencia de dos. Así, el primer elemento contiene información sobre los marcos individuales libres, el siguiente sobre bloques de dos marcos consecutivos, y así sucesivamente. Estos elementos contienen punteros a las entradas de la tabla de marcos correspondientes. Además, posee un mapa de bits por cada conjunto de bloques de marcos para seguir la pista de los marcos libres. De este modo, el bit  $n$ -ésimo valdrá 1 si el bloque  $n$ -ésimo de marcos está libre.

La figura 8.17 muestra la estructura de esta tabla. El elemento 0 tiene un marco libre (marco 0), y el elemento 2 tiene dos bloques libres (cada uno de cuatro marcos), el primero empezando en el marco 4 y el segundo en el marco 56.

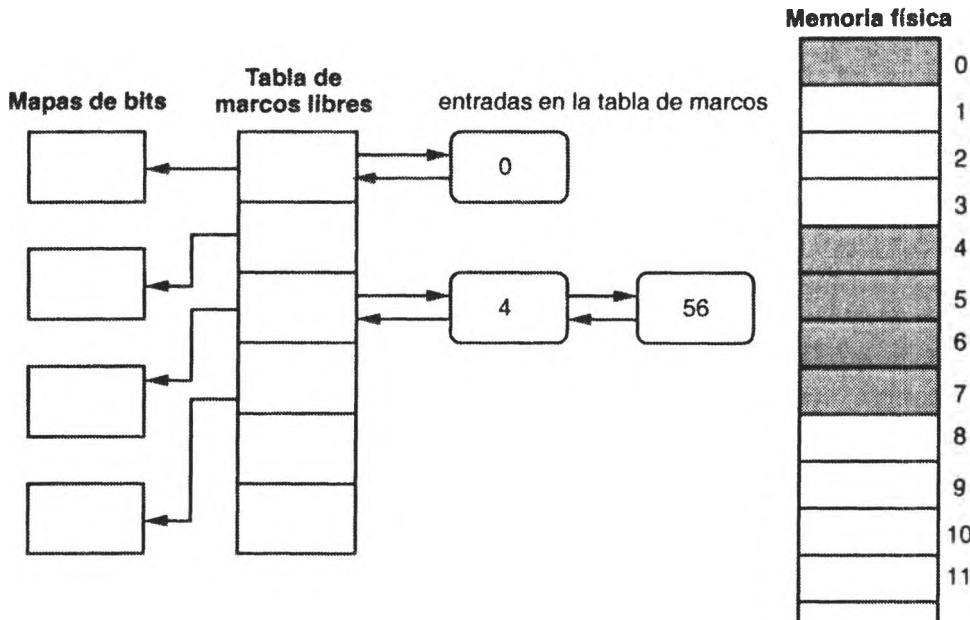


Figura 8.17: Estructura de la lista de marcos libres

#### 8.6.5.1 Asignación de páginas

La gestión del espacio libre comentada anteriormente, permite que **LINUX** emplee el algoritmo compañero para la asignación de páginas. Las solicitudes pueden

ser de marcos individuales o de conjuntos de marcos. Mediante el empleo de paginación por demanda, la mayoría de las solicitudes son de marcos individuales. Sin embargo, hay ocasiones en las que un proceso necesita más de un marco de forma simultánea, como por ejemplo, cuando se crea un proceso hijo. En este caso, inicialmente se comparten las páginas de código y de datos entre padre e hijo, pero cuando se realiza una modificación, hay que mantener imágenes distintas. También cuando un proceso quiere crear un segmento de memoria compartido con otros, puede necesitar más de un marco. Igualmente, cuando un proceso en estado de suspendido vuelve a activarse hay que devolverle la memoria que tenía asignada en ese momento.

Cuando un proceso solicita un conjunto de marcos, se busca en la tabla de marcos libres un bloque libre del tamaño requerido. Si no existen bloques de páginas del tamaño requerido, se busca en los bloques del siguiente tamaño. Este proceso continúa hasta encontrar un bloque de marcos libre. Si el bloque asignado es mayor que el solicitado, los marcos sobrantes se introducen en la tabla de bloques libres de acuerdo a su nuevo tamaño.

Así, en la figura 8.17 si se solicita un bloque de dos páginas, el primer bloque de 4 marcos libres (empieza en el marco 4) se divide en dos bloques de dos marcos. El primero, que empieza en el marco 4, se asignará al proceso que lo solicitó, y el segundo bloque, que empieza en el marco 6, se introducirá en la lista de bloques de marcos libres de tamaño 2.

Una consecuencia de este algoritmo es que el proceso puede ocupar, siempre que sea posible, posiciones contiguas en la memoria.

#### 8.6.5.2 Liberación de páginas

Cuando se libera una página, se comprueba que el bloque adyacente (bloque compañero) del mismo tamaño está libre. En ese caso, se combinan los dos formando un nuevo bloque libre. Cada vez que se unen dos bloques, el algoritmo intenta fusionarlo con bloques adyacentes para formar uno más grande. De este modo, se busca conseguir que los bloques de marcos libres sean tan grande como lo permita la memoria utilizada.

Así, en la figura 8.17 si el marco número 1 se libera, se combinará con el marco 0, creando un bloque de dos marcos libres que comienza en el marco 0.

#### 8.6.6 Fallos de página

El sistema puede incurrir en dos tipos de fallos de página:

**Fallo de validez** La página referenciada no se encuentra en memoria.

**Fallo de protección** Se intenta acceder a una página en un modo no permitido o bien a una página que no pertenece al proceso.

La forma de indicar el tipo de fallo depende de la arquitectura. En el caso de la familia Intel se utiliza el registro CR2 para indicar la dirección causante del fallo de página. De este registro de 16 bits sólo se utilizan los tres primeros, como se muestra en la tabla 8.8.

Bit	Significado con 0	Significado con 1
0	Página no presente	Nivel de protección
1	Fallo debido a lectura	Fallo debido a escritura
2	Modo supervisor	Modo usuario

**Tabla 8.8:** Códigos de error de un fallo de página

En la arquitectura Alpha AXP se emplean los bits de la tabla de páginas FR, FW y FX.

Cuando se produce un fallo de página, el sistema comprueba en primer lugar si la página pertenece al proceso y si se intenta acceder a ella de un modo adecuado. Esta comprobación se realiza buscando la región de memoria virtual asociada. Si la página y el modo son correctos se inicia la lectura de ésta.

La página que provoca el fallo puede encontrarse en el área de intercambio o bien, en el fichero ejecutable. En el primer caso, su dirección está almacenada en la entrada correspondiente de la tabla de páginas. Mientras que en el segundo, se encuentra en la estructura asociada a la región de memoria virtual correspondiente.

### 8.6.7 El demonio de paginación

El demonio de paginación es un hilo del núcleo que se ejecuta en segundo plano y que se activa una vez cada segundo. Su función es elegir los marcos que tienen que liberarse para poder alojar nuevas páginas, aplicando el algoritmo del reloj.

Cuando se activa comprueba el número de marcos libres que hay en el sistema, si está por debajo de un valor umbral empieza a liberar memoria. Dispone de tres umbrales que modifican su forma de actuación, es decir, el intervalo de tiempo para su activación.

Si una página seleccionada por el demonio de paginación contiene código del

proceso, simplemente se descarta, debido a que puede volver a ser leída de nuevo del fichero ejecutable, ya que nunca se modifican. Por el contrario, si la página seleccionada ha sido modificada desde que se cargó en memoria principal, ésta se escribirá en el área de intercambio para liberar memoria. La entrada en la tabla de páginas se marcará como no válida, y se almacenará la dirección de la página en el área de intercambio.

Una vez que se han liberado los marcos, se termina el trabajo actualizando la tabla de marcos y las tablas de páginas. Las páginas que contienen al núcleo, así como las tablas de páginas de primer nivel, son páginas reservadas que nunca se descargarán de memoria.

### 8.6.8 Gestión del área de intercambio

LINUX puede utilizar como área de intercambio tanto una partición como un fichero. Si se utiliza una partición las operaciones con esta área son más rápidas. Sin embargo, la utilización de un fichero permite cambiar de forma fácil su tamaño, ya que no se necesita reparticionar el disco. Se pueden utilizar varias de estas áreas de forma simultánea.

Para llevar el control del espacio libre en las áreas de intercambio se utiliza un mapa de bits con un bit por página. En esta estructura no se indica el proceso propietario de la página, esta información se mantiene en su tabla de páginas.

La utilidad del área de intercambio es el almacenamiento de aquellas páginas que no pueden ser leídas del fichero ejecutable. Así, las páginas de código siempre se lee del propio programa, mientras que las de datos, pila, así como las tablas de páginas de segundo nivel pueden ser intercambiadas y descargadas a dicha área.

### 8.6.9 Llamadas al sistema

LINUX, al igual que su progenitor UNIX, tampoco especifica llamadas al sistema para la administración de la memoria. Esto se consideró como demasiado dependiente de la máquina como para lograr su estandarización. No obstante posee unas pocas llamadas al sistema relacionadas con la gestión de memoria, que se pueden ver en la tabla 8.9.

Llamada	Descripción
brk	Especificar el tamaño del segmento de datos.
malloc	Solicitar memoria.
free	Liberar memoria.
swapon	Activar área de intercambio.
swapoff	Desactivar área de intercambio.

**Tabla 8.9:** Llamadas al sistema relacionadas con la gestión de memoria en LINUX

## 8.7 Resumen

Un sistema de memoria virtual es capaz de ejecutar procesos sin que la imagen de éstos se encuentre por completo en memoria principal. Estos sistemas se implementan basándose en un sistema paginado, segmentado, o una combinación de ambos. En este capítulo se ha realizado un estudio de los sistemas de memoria virtual paginados.

Una de las características de los sistemas de memoria virtual es que un proceso puede tener un tamaño muy superior a la memoria física disponible. Esto produce un aumento considerable en el tamaño de las tablas de páginas, que se pueden reducir mediante el empleo de tablas de páginas multinivel o tablas de páginas invertidas.

En estos sistemas, cuando un proceso realiza una referencia que no se encuentra en memoria principal se produce un fallo de página. La existencia de éstos afecta de forma significativa al rendimiento del sistema, debido a las operaciones de E/S que se encuentran implicadas en su resolución. Por este motivo, uno de los objetivos que debe perseguir el diseño de un gestor de memoria virtual es la minimización de los fallos de página.

Un gestor de memoria virtual incorpora un conjunto de políticas para asegurar el buen funcionamiento del sistema, tales como la de lectura, colocación, sustitución, gestión del conjunto residente, política de limpieza y el control de la carga.

La política de lectura es la encargada de determinar qué página traer y cuándo se incorpora desde el almacenamiento secundario a la memoria principal. En este sentido, se estudian las técnicas de paginación por demanda y prepaginación. La política de colocación es la responsable de ubicar la página en memoria principal.

La política de sustitución decide qué página tiene que salir de memoria principal cuando ésta se encuentra llena y hay que cargar una nueva. Se estudian los

algoritmos óptimo, LRU, FIFO, etc. Esta política está íntimamente relacionada con la gestión del conjunto residente, que se encarga de determinar su tamaño, fijo o variable, así como el alcance de la sustitución, local o global.

La política de limpieza realiza la escritura de las páginas que han sido modificadas en memoria principal. Se puede realizar en el momento en que la página sale de memoria principal, conocido como limpieza por demanda, o bien antes de salir, denominándose limpieza previa.

El control de la carga hace referencia al control que lleva el sistema para la evitación del problema de hiperpaginación. Esta situación se caracteriza porque produce una disminución del rendimiento del sistema.

## 8.8 Ejercicios

1. ¿Por qué es generalmente más deseable descargar una página que no ha sido modificada frente a una que sí lo ha sido? ¿Es posible que un algoritmo LRU escoja una página modificada frente a otra que no lo ha sido? ¿Por qué? Explíquelo con un ejemplo.
2. Hemos diseñado un nuevo algoritmo de sustitución de páginas que es bastante complejo, pero pensamos que es óptimo. Sin embargo, en algunos casos específicos, presenta la anomalía de Belady. ¿Es óptimo el nuevo algoritmo? Razone la respuesta.
3. ¿Qué es la hiperpaginación? ¿Cómo la detecta el sistema? ¿Qué puede hacer para eliminar este problema?
4. El modelo del conjunto de trabajo viene definido por el tamaño de la ventana ( $\Delta$ ). Si disponemos de un tamaño de ventana  $\Delta = 3$ , ¿cuántos marcos como mínimo tendrá asignado un proceso? ¿Y como máximo? Razone las respuestas.
5. Disponemos de un sistema con direcciones virtuales de 32 bits y físicas de 16 bits. El tamaño de la página es de 4 KiB y el de la entrada de la tabla de páginas de 32 bits. ¿Cuántas tablas de páginas tendremos si la implementamos en dos niveles? ¿Y si implementamos una tabla invertida? ¿Cuántas entradas poseerá cada tabla?
6. En un sistema de memoria virtual con paginación, ¿es posible que un proceso que está bloqueado pueda seguir ejecutándose si no posee ninguna página en memoria? Razone la respuesta.
7. ¿Qué operaciones de disco requiere un fallo de página? Razone la respuesta.

8. ¿Cómo realiza el sistema LINUX la gestión del espacio libre en memoria?
9. ¿Por qué no es posible un esquema de asignación fija y alcance global?
10. Consideremos la siguiente secuencia de referencias a memoria de un proceso de 460 palabras:

10, 11, 104, 170, 73, 185, 245, 246, 434, 458, 364

- (a) Indique la cadena de referencias a páginas suponiendo un tamaño de página de 100 palabras.
  - (b) Calcule la tasa de fallos de páginas para esta cadena de referencias, suponiendo que el proceso dispone de una memoria de 200 palabras para los siguientes algoritmos de sustitución: FIFO, LRU y óptimo.
  - (c) Supongamos que vamos a aplicar el modelo del conjunto de trabajo con  $\Delta = 4$ . Determine en cada referencia a memoria cuál es el conjunto de trabajo, e indique los fallos de página que se producen.
  11. Disponemos de un sistema paginado con direcciones físicas y lógicas de 32 bits y páginas de 1024 bytes. En un instante dado, la tabla de páginas presenta el siguiente aspecto:
- | Página | Marcos |
|--------|--------|
| 0      | 3      |
| 1      | 10     |
| 2      | 4      |
| 3      | NULL   |
| ...    | ...    |
- (a) ¿Cuántas páginas puede tener un proceso? ¿Cuántos marcos de memoria tendremos?
  - (b) ¿Cuál es el tamaño del marco de memoria?
  - (c) Exprese las siguientes direcciones relativas en lógicas (página y desplazamiento), y en físicas (marco, desplazamiento).
    - i. 1120
    - ii. 2398
    - iii. 3597
  12. Un sistema tiene 32 marcos, cada uno de ellos de 1024 bytes. Las direcciones lógicas tienen el doble de tamaño que las direcciones físicas. Conteste las siguientes preguntas de forma razonada:
    - (a) ¿Cuántas entradas tendremos en la tabla de marcos?

- (b) Si cada entrada ocupa 4 bits, ¿qué memoria se necesita para almacenar esta tabla?
- (c) ¿Cuántos bits tiene la dirección física?
- (d) ¿Cuántas entradas habrá en la tabla de páginas de cada proceso?
13. Un proceso tiene asignado cuatro marcos. En la tabla se muestran los instantes de carga, del último acceso a la página, así como los bits de referencia (R) y de modificación (M).

Página	Marco	Instante carga	Instante referencia	R	M
2	0	60	161	0	1
1	1	130	160	0	0
0	2	26	162	1	0
3	3	20	163	1	1

En el instante 165 se hace referencia a la página 4. ¿Qué página se reemplazará en cada una de las siguientes políticas?

- (a) FIFO.
- (b) LRU.
- (c) Reloj.
- (d) Reloj mejorado.
14. Un sistema paginado tiene un tamaño de página de 512 palabras, una memoria virtual de 512 páginas (numeradas de 0 a 511), y una memoria física de 10 marcos (numerados de 0 a 9). El contenido de la memoria física es el siguiente:

Dirección física	Contenido
0	...
1536	Página 34
2048	Página 9
...	...
3072	Tabla de páginas
3584	Página 65
...	...
4608	Página 10
...	...

- (a) Determine el contenido de la tabla de páginas.
- (b) ¿Qué direcciones físicas son referenciadas con las siguientes direcciones virtuales: 4608, 5120, 5119 y 33300?



---

**Parte 5**

## **ENTRADA/SALIDA**

---



# Capítulo 9

## Gestión de dispositivos

---

La gestión de los dispositivos de E/S es una de las funciones principales del sistema operativo, debido a que proporciona un medio para que éstos sean manipulados uniformemente mediante un conjunto de órdenes de alto nivel. En este capítulo veremos los principales aspectos del diseño de esta interfaz entre los dispositivos de E/S y los usuarios, prestando especial atención al rendimiento de las operaciones de E/S.

### 9.1 Introducción

El sistema de E/S es el elemento del sistema operativo que permite el intercambio de información entre éste y el exterior. En su diseño se persiguen dos objetivos fundamentales, **rendimiento y uniformidad**.

Debido a que la mayor parte de los dispositivos de E/S son muy lentos en comparación con la memoria principal y el procesador, pueden constituir un cuello de botella. Por este motivo, uno de los objetivos principales del sistema de E/S es obtener un buen rendimiento de los dispositivos.

Dada la diversidad de dispositivos de E/S existentes, un objetivo importante es realizar un tratamiento uniforme de ellos, tanto desde el punto de vista del usuario como de la gestión por parte del sistema operativo. Este objetivo es difícil de alcanzar en la práctica debido a las diferentes características que presentan los

dispositivos.

## 9.2 Dispositivos de E/S

Los dispositivos de E/S son el medio de comunicación entre el procesador y el exterior, ya que permiten transferir información en ambos sentidos. Existe una gran variedad de dispositivos de E/S. Así, tenemos dispositivos de almacenamiento (discos, cintas), de transmisión (*modems*, tarjetas de red), e interfaces con los usuarios (pantallas, teclados, ratones), cada uno con sus propias características. Sin embargo, éstos se pueden clasificar según dos criterios. Por un lado, si tenemos en cuenta el número de bytes transferidos en una operación de E/S individual podemos distinguir:

**Dispositivos de bloques** Almacenan la información en bloques de tamaño fijo, cada uno con su propia dirección. Su característica principal es la posibilidad de leer o escribir bloques de forma independiente. Ejemplos de éstos son los discos, CD-ROM, etc.

**Dispositivos de caracteres** Envían o reciben un flujo de caracteres sin estructura. Al contrario que los dispositivos de bloques, no son direccionables por lo que no se puede acceder directamente a un determinado carácter. Ejemplos de este tipo de dispositivos son las impresoras, terminales, cintas, etc.

También podemos clasificarlos en función de su propósito; en este caso, podemos distinguir:

**Dispositivos de almacenamiento** Se utilizan para almacenar información de forma permanente. Como ejemplo podemos citar los discos, cintas, etc.

**Dispositivos de comunicación** Se emplean para la comunicación entre los usuarios y el sistema de computación o entre diferentes sistemas. Ejemplos de estos son los *modems*, terminales, impresoras, etc.

Aunque la primera clasificación no es perfecta, es la más empleada. En ella, algunos dispositivos como los relojes, no están contemplados. Sin embargo, este modelo de clasificación es bastante general y se puede utilizar como base para el software del sistema operativo.

### 9.2.1 Controladoras de dispositivos

Los dispositivos de E/S, por lo general, constan de un componente mecánico y uno electrónico. A menudo es posible separar estos dos componentes para ofrecer un diseño más modular y general. El primero es el dispositivo propiamente dicho, y el segundo se denomina **controladora**.

Es necesario establecer esta distinción debido a que las controladoras actúan como intermediarias entre el sistema y los dispositivos de E/S. Su propósito es superar las incompatibilidades de velocidad, señalización de niveles entre el procesador y periféricos, traducir las órdenes de E/S genéricas emitidas por la CPU a controles específicos del dispositivo, codificación, etc. Es decir, la controladora se entiende con la CPU a alto nivel, mientras que la comunicación que establece con el periférico es de muy bajo nivel, adaptándose a las características de éste.

En la figura 9.1 se representa un diagrama general de una controladora de E/S, donde podemos observar que se divide en tres capas funcionales:

- Interfaz con el bus.
- Controladora de dispositivo genérico.
- Interfaz del dispositivo.

Las dos capas que actúan de interfaz son de gran importancia para los diseñadores de hardware. La capa intermedia —la controladora de dispositivo genérico— es la más importante para los diseñadores de software. Ésta hace que cada dispositivo tenga la apariencia de un conjunto de registros dedicados, que se conocen como **puerto de E/S**. Éstos disponen normalmente de tres tipos de registros, de datos, de estado y de órdenes.

Los registros de datos forman un *buffer* hardware, almacenando temporalmente los datos hasta que la CPU o el dispositivo pueda recibirlos. El sistema operativo efectúa las operaciones de E/S escribiendo las órdenes oportunas en los registros de órdenes. El dispositivo indica su estado a la CPU mediante los registros de estado. Éstos contienen información sobre el dispositivo (preparado u ocupado), la memoria intermedia (vacía o llena) e indicaciones de error.

Para poder efectuar las operaciones de E/S la CPU ha de poder direccionar los registros. Tradicionalmente, se incorporan una serie de instrucciones específicas para las operaciones de E/S, en las cuales se referencia cada registro mediante un identificador único. De esta forma los registros de la controladora forman un espacio de direcciones diferente al de la memoria principal. En este caso, se

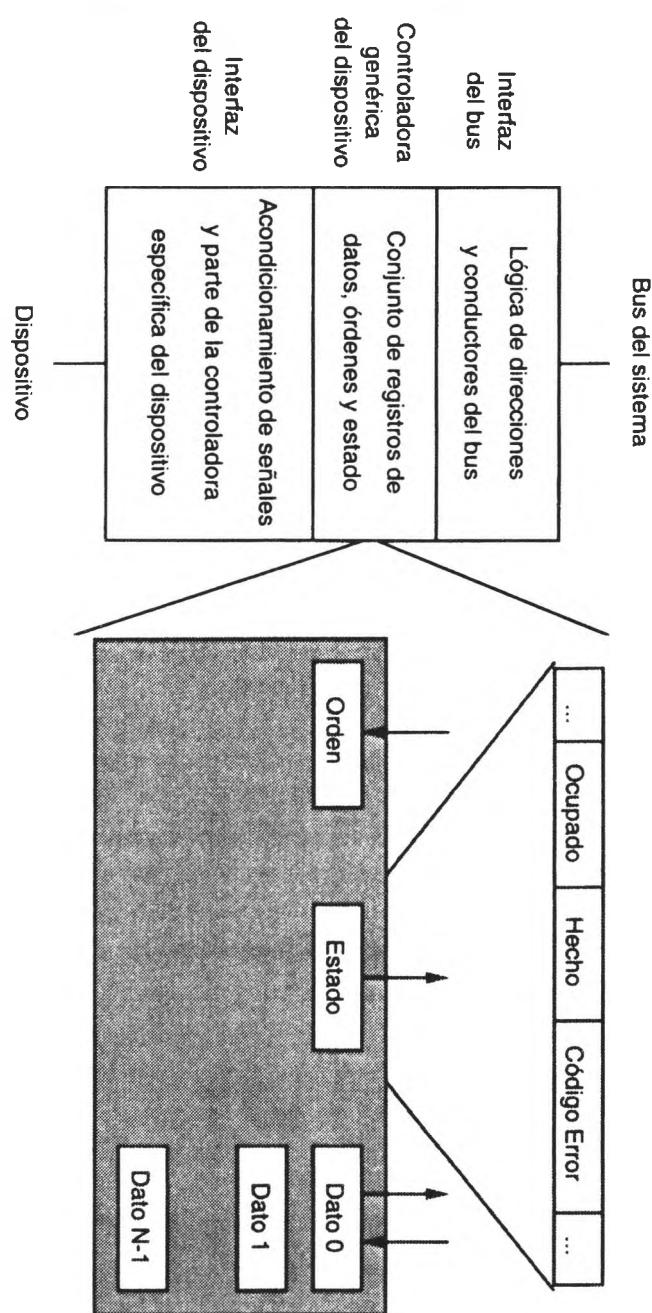


Figura 9.1: Estructura de una controladora de E/S

dice que la E/S está **mapeada en E/S** o es una **E/S aislada**, y el conjunto de instrucciones de E/S recibe el nombre de **instrucciones de E/S directa**. Ejemplos de este tipo de instrucciones son las siguientes:

*input dirección\_dispositivo*

*output dirección\_dispositivo*

*copy\_in registro\_CPU, dirección\_dispositivo, registro\_controladora*

*copy\_out registro\_CPU, dirección\_dispositivo, registro\_controladora*

*test registro\_CPU, dirección\_dispositivo*

Otra alternativa es asociar a los registros una dirección de memoria, en lugar de un identificador específico. De esta forma, las direcciones de los registros forman parte del espacio de direcciones. Este esquema recibe el nombre de **E/S mapeada en memoria**. Así, por ejemplo, a un dispositivo se le podría asignar el espacio de direcciones desde 0xFFFF0120 hasta 0xFFFF012F para referenciar sus registros. Una instrucción de E/S mapeada en memoria podría ser la siguiente:

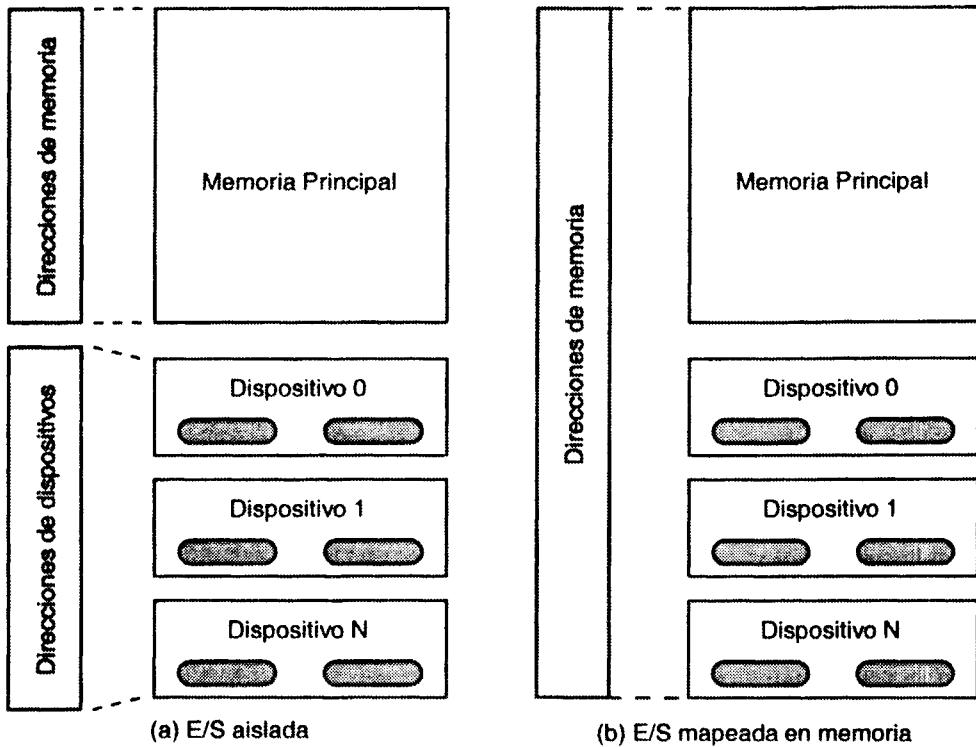
**store R3, 0xFFFF0124**

La E/S mapeada en memoria reduce el número de instrucciones de la CPU ya que las propias instrucciones de acceso a la memoria pueden ser empleadas para realizar las operaciones con los registros. Sin embargo, la E/S aislada necesita incorporar instrucciones específicas para manipularlos. La figura 9.2 muestra las dos alternativas para referenciar los registros de la controladora.

## 9.3 Organización del sistema de E/S

Para lograr los objetivos de uniformidad y buen rendimiento, el sistema de E/S se estructura en capas, de forma que los niveles inferiores ocultan las particularidades del hardware a los procesos y a los niveles superiores del software de E/S.

Una posible organización es la que aparece en la figura 9.3 que muestra los distintos componentes hardware y software que intervienen en las operaciones de E/S. Situándonos en el nivel más bajo, cada dispositivo utiliza una controladora para conectarse a los buses de datos y direcciones del ordenador. La capa de software que está en contacto directo con las controladoras de dispositivos, está formada por un conjunto de manejadores de dispositivos que se encargan de tratar las particularidades de éstos. Por encima de ésta se implementa otra que presenta una interfaz uniforme de todos los dispositivos a los usuarios y programadores del sistema. Esta capa se denomina **subsistema de E/S** e implementa una interfaz



**Figura 9.2:** Direccionamiento de los registros de las controladoras

de programación de aplicaciones abstracta (API).

Es decir, el subsistema de E/S se encarga de conseguir el objetivo de la uniformidad proporcionando las funciones disponibles en los dispositivos mediante la API, mientras que los manejadores de dispositivos implementan estas funciones. El objetivo de obtener un buen rendimiento es compartido por ambas capas.

Un problema que plantea la división del sistema de E/S en dos capas es que los manejadores de dispositivos, además de ser específicos para el dispositivo, deben respetar la API del sistema operativo. Como cada sistema operativo tiene su propia interfaz, los fabricantes de dispositivos tienen que proporcionar un manejador para cada sistema operativo.

Un aspecto importante en el diseño de un manejador es que necesita ejecutar instrucciones privilegiadas, ya que debe leer y escribir información en el espacio de los procesos de usuario y manipular ciertas estructuras del sistema operativo, como la tabla de estado de los dispositivos. En sistemas antiguos, el manejador

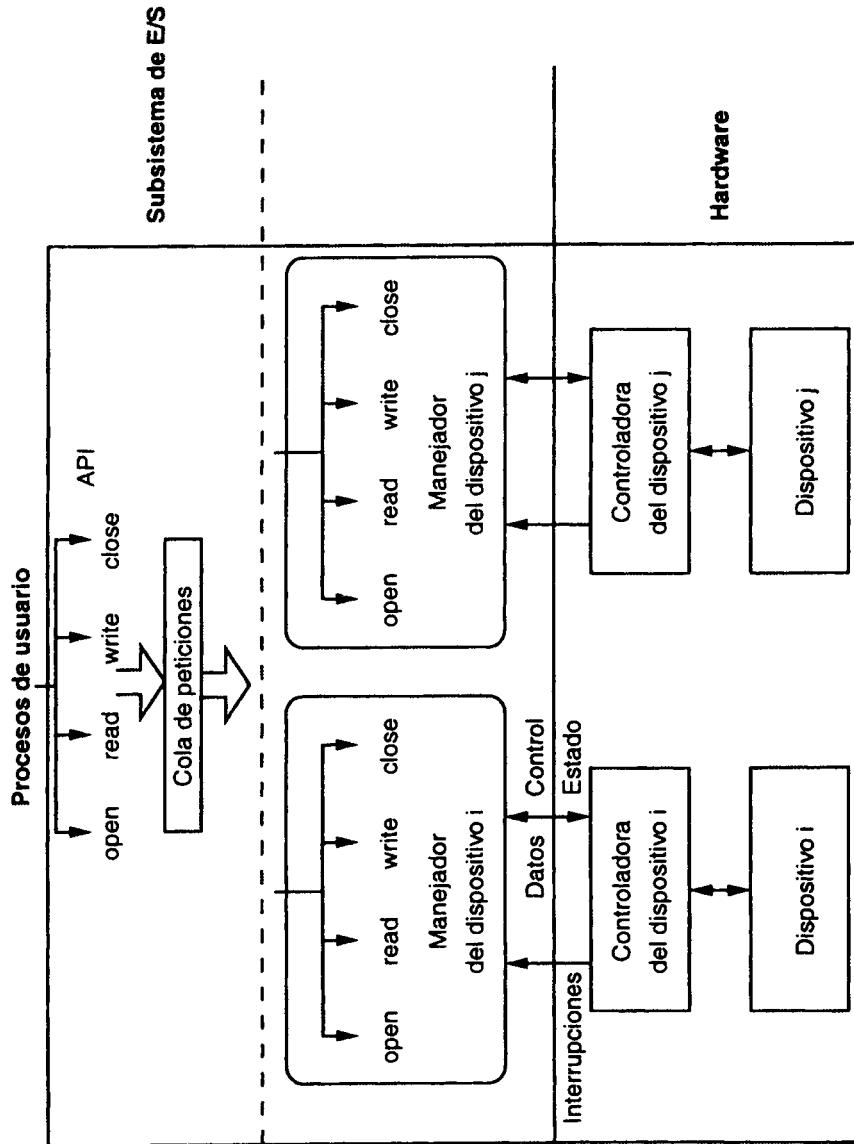


Figura 9.3: Estructura del sistema de E/S

se incorporaba en el núcleo. Por lo que añadir un nuevo dispositivo implicaba modificar el código fuente para introducir su manejador y recompilarlo. Esto era aceptable cuando el propio vendedor de hardware lo instalaba.

En los sistemas modernos se simplifica esta instalación utilizando manejadores de dispositivos reconfigurables, que permiten añadir un nuevo manejador al sistema sin necesidad de volver a compilar. En estos casos, las funciones de los manejadores de dispositivos se enlazan de forma dinámica al código del núcleo. La figura 9.4 muestra las estructuras necesarias en este proceso. El sistema operativo emplea una **tabla de referencias indirectas** para acceder a cada función del manejador del dispositivo. De esta forma, el sistema crea una API independiente del dispositivo. Cuando un proceso realiza una petición de un servicio para una operación de E/S, el núcleo la pasa al manejador del dispositivo correspondiente a través de la tabla de referencias indirectas. Cuando el manejador se instala, se actualiza la información de dicha tabla para que el sistema pueda efectuar las llamadas correspondientes.

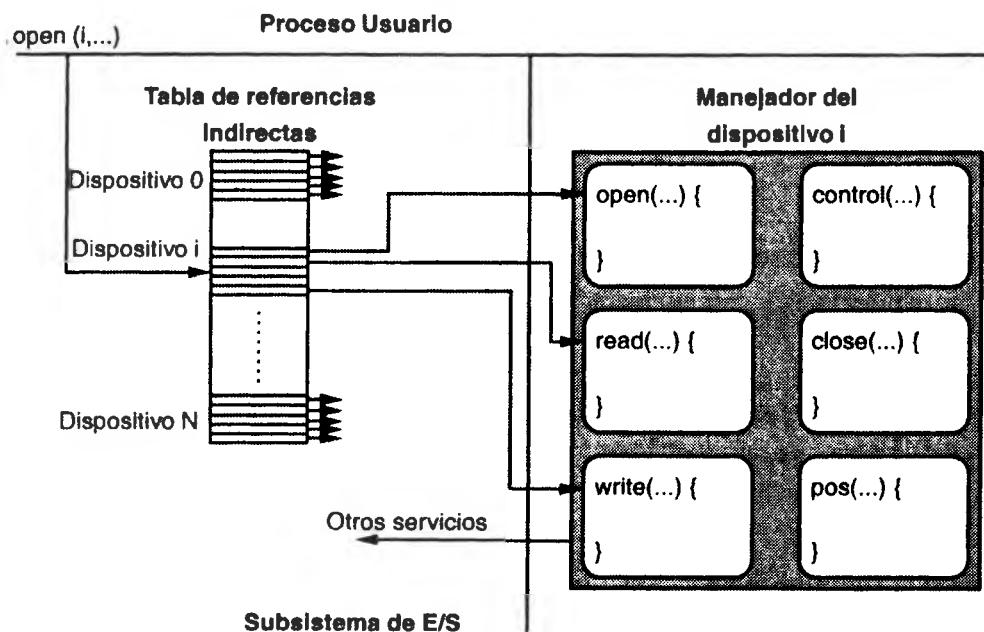


Figura 9.4: Manejadores de dispositivos reconfigurables

## 9.4 Modos de realizar las operaciones de E/S

Los manejadores de dispositivos, para realizar las operaciones de E/S, tienen que hacer uso del puerto de E/S. Dependiendo de las características del dispositivo, esto se puede llevar a cabo de tres formas diferentes:

- **E/S controlada por programa.**
- **E/S controlada por interrupciones.**
- **Acceso directo a memoria.**

### 9.4.1 E/S controlada por programa

Esta técnica también se conoce como **escrutación**. En ella, el procesador tiene un control completo sobre la operación de E/S, encargándose de transferir la información entre la memoria principal y los registros de la controladora.

La controladora no realiza ninguna acción para alertar al procesador, sino que es responsabilidad del sistema comprobar periódicamente el estado de la controladora hasta que el dispositivo haya completado la operación.

Para una operación de entrada los pasos a realizar son los siguientes (ver figura 9.5):

1. El proceso de usuario realiza una operación de lectura; a través del servicio correspondiente da la orden al manejador del dispositivo.
2. El manejador del dispositivo comprueba el registro de estado de la controladora para determinar si el dispositivo está libre. Si está ocupado, espera hasta que termine.
3. El manejador almacena una orden de entrada en el registro de órdenes de la controladora y, de esta forma, inicia el dispositivo.
4. El manejador comprueba continuamente el registro de estado de la controladora, esperando a que el dispositivo finalice su operación. Esta comprobación se denomina **escrutinio**.
5. El manejador copia el contenido de los registros de datos de la controladora en el espacio de memoria del proceso de usuario.
6. Se devuelve el control al proceso de usuario.

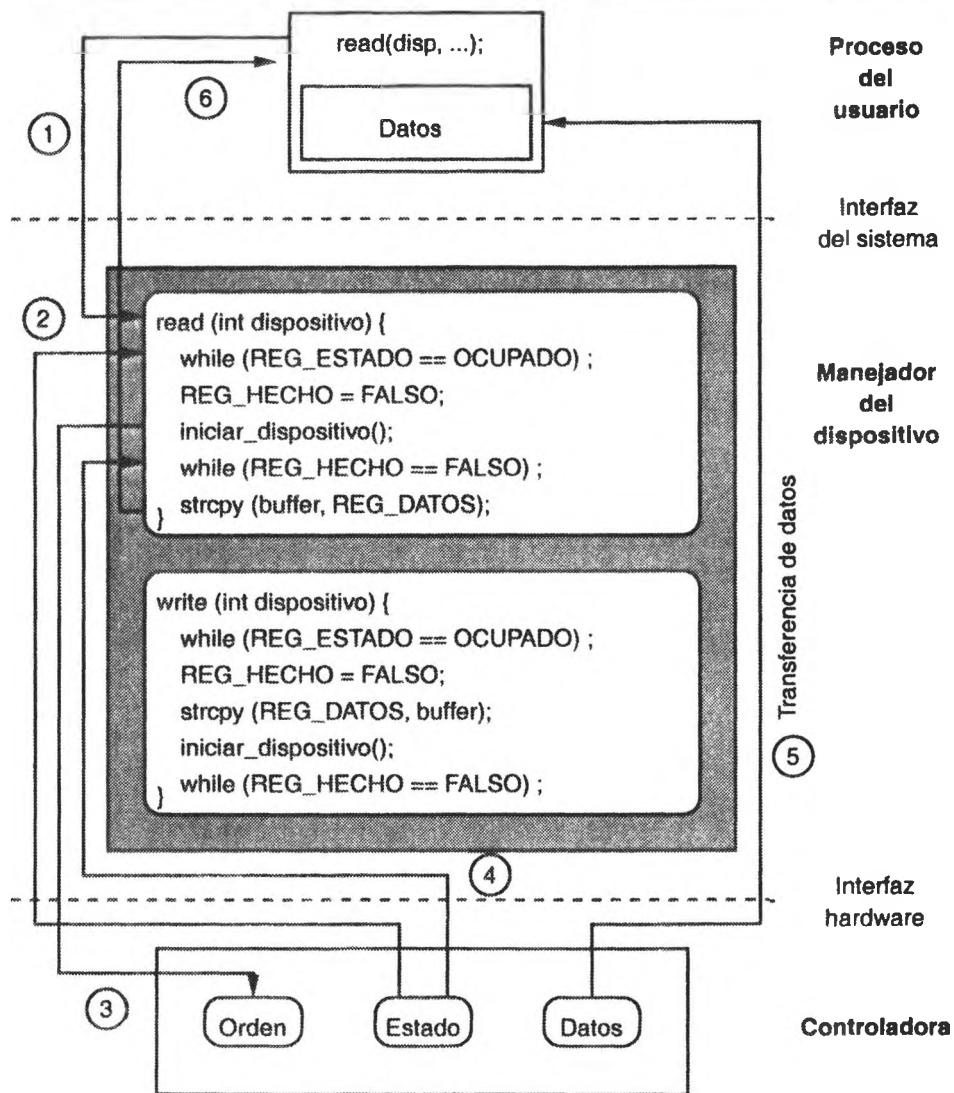


Figura 9.5: E/S controlada por programa

En el caso de realizar una operación de escritura, una vez determinado que el dispositivo está libre se copian los datos del espacio de usuario a los registros de la controladora y se inicia la operación.

Podemos ver que cuando se utiliza esta técnica el código del manejador del dispositivo incluye instrucciones para que el procesador tenga un control total sobre la operación de E/S. Entre éstas se incluyen la comprobación del estado de

los dispositivos, el envío de órdenes de lectura o escritura y la transferencia de los datos.

#### 9.4.2 E/S controlada por interrupciones

El problema que plantea la técnica anterior es que el procesador consume ciclos esperando la finalización de la operación de E/S, interrogando repetidamente el estado del dispositivo. Es decir, se encuentra en una espera activa. Como consecuencia, el rendimiento del sistema en conjunto se degrada considerablemente.

Con objeto de eliminar la necesidad de interrogar continuamente al dispositivo, una alternativa es el empleo de las interrupciones. En este caso, cuando el procesador da una orden de E/S a la controladora puede continuar con otro trabajo, ya que ésta le notificará mediante una interrupción del fin de la operación de E/S. El procesador, al igual que en la técnica anterior, ejecutará la transferencia de datos.

Con este esquema se necesitan otros elementos para la gestión del dispositivo, una **tabla de estado de los dispositivos**, que contiene una entrada por cada uno, y el manejador de interrupciones, encargado de determinar el tipo de interrupción y efectuar las acciones oportunas.

Los pasos a realizar para una operación de entrada son los siguientes (ver figura 9.6):

1. El proceso de usuario pide realizar una operación de lectura a través del servicio correspondiente.
2. El manejador del dispositivo comprueba el registro de estado de la controladora para determinar si el dispositivo está libre. Si está ocupado, espera a que termine.
3. El manejador almacena una orden de entrada en el registro de órdenes de la controladora y, de esta forma, inicia el dispositivo.
4. El manejador almacena la información de la operación iniciada en la tabla de estado de los dispositivos. En la entrada correspondiente se guarda la dirección de retorno de la llamada original del proceso, así como parámetros especiales para la operación de E/S. En este momento, el proceso pasa a estado bloqueado y se llama al planificador a corto plazo para continuar con la ejecución de otro proceso.

5. Cuando el dispositivo termina la operación emite una interrupción, que provoca que el sistema operativo tome el control y se lo pase al manejador de interrupciones.
6. El manejador de interrupciones determina qué dispositivo causó la interrupción y le pasa el control a su manejador.
7. El manejador del dispositivo recupera de la tabla de estado de los dispositivos la información de la operación de E/S.
8. El manejador del dispositivo copia el contenido de los registros de datos de la controladora en el espacio del usuario.
9. El manejador del dispositivo devuelve el control al planificador para que desbloquee al proceso de usuario.

En este esquema, el manejador del dispositivo dispone de dos elementos. El primero son las rutinas que se encargan de traducir la orden de E/S a la controladora (pasos 3 y 4). El otro es la rutina a ejecutar cuando se produce la interrupción de finalización de la operación de E/S (pasos 7, 8 y 9).

Esta técnica de E/S permite aumentar el rendimiento del sistema, ya que permite ejecutar un proceso mientras otro está esperando una operación de E/S. Por tanto, la E/S controlada por interrupciones es más eficiente que la controlada por programa porque elimina las esperas innecesarias. Sin embargo, sigue consumiendo una gran cantidad de tiempo del procesador, ya que éste se encarga de transferir cada palabra de datos que va de la memoria al dispositivo o viceversa. Esto hace que el rendimiento del sistema disminuya cuando se tiene que transferir un gran volumen de datos.

#### 9.4.3 Acceso directo a memoria

Una técnica más eficiente que las anteriores es el acceso directo a memoria o DMA<sup>1</sup>. Ésta permite que el dispositivo transfiera datos directamente a o desde la memoria, descargando al procesador de esta tarea, por lo que es ideal cuando se transfieren grandes volúmenes de datos.

Para poder utilizar esta técnica son necesarias controladoras especiales capaces de realizar las operaciones de E/S directamente en memoria. Éstas incorporan dos registros adicionales para almacenar la dirección del bloque de memoria implicado y el número de bytes a transferir en la operación.

<sup>1</sup> Direct Memory Access.

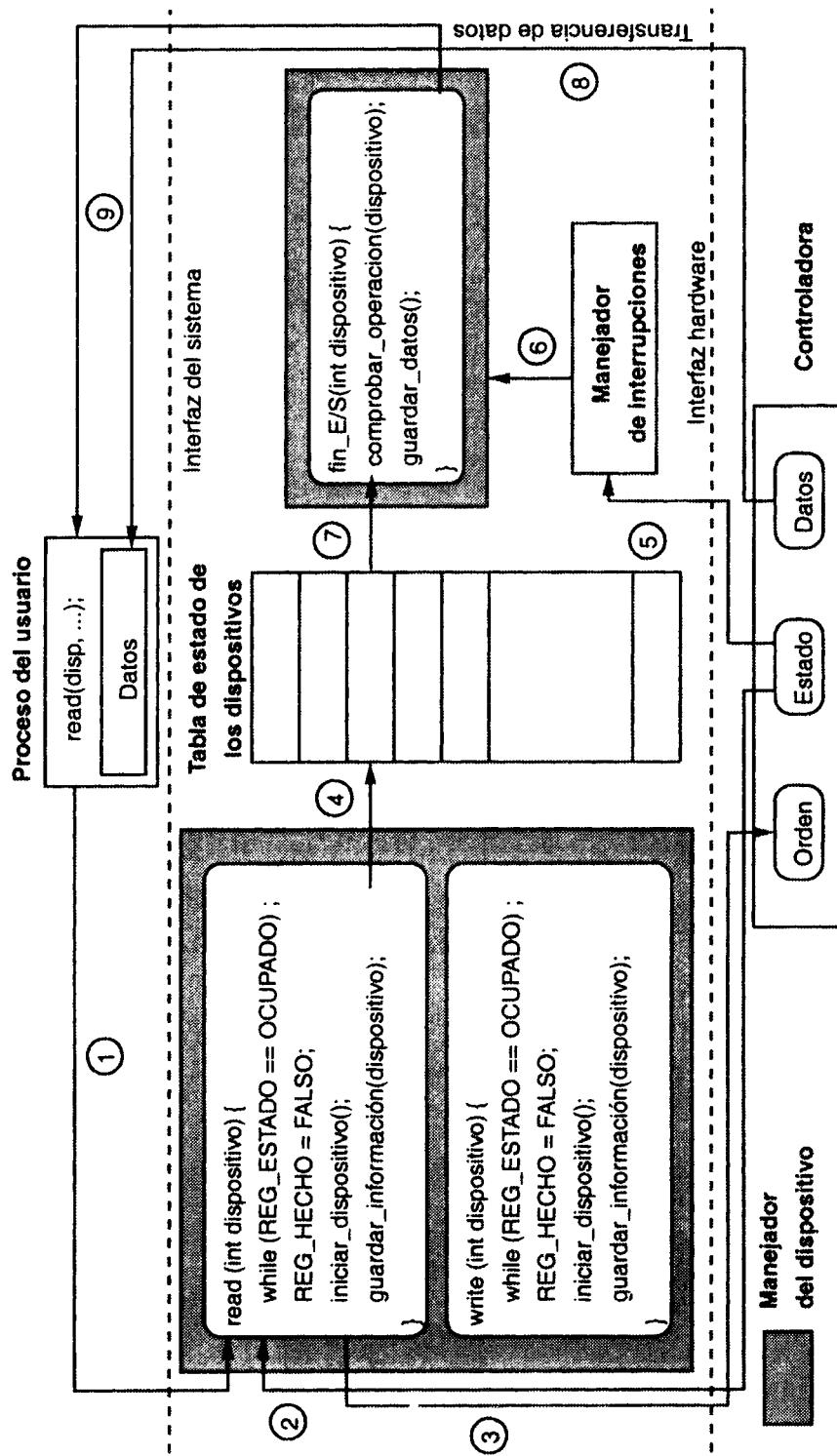


Figura 9.6: E/S controlada por interrupciones

Cuando se utiliza una controladora DMA, los pasos que se realizan para una operación de E/S son los siguientes:

1. El proceso de usuario solicita realizar una operación de E/S a través del servicio correspondiente.
2. El manejador del dispositivo envía la orden a la controladora de DMA con la siguiente información:
  - Tipo de operación (lectura o escritura).
  - Dirección del dispositivo de E/S implicado.
  - Dirección de memoria donde hay que empezar a leer o escribir.
  - Número de bytes a transferir.
3. El proceso que solicita la operación de E/S pasa a estado bloqueado, y el planificador a corto plazo escoge otro proceso listo para continuar con su ejecución. En este caso, se delega la realización de la operación de E/S a la controladora DMA.
4. Dependiendo del tipo de operación, los datos a transferir se copiarán desde la memoria (salida) o el dispositivo (entrada) al *buffer* de la controladora.
5. A continuación la controladora transferirá los datos desde su *buffer* a su destino (dispositivo o memoria).
6. Cuando la transferencia se completa, la controladora produce una interrupción.
7. El procesador comprueba que la operación ha finalizado correctamente, y el proceso bloqueado pasa a estado listo para que continúe con su ejecución.

Podríamos pensar que la controladora podría realizar la transferencia de la información en un solo paso, en lugar de los pasos 4 y 5, sin necesidad del *buffer*. Sin embargo, la transferencia de cada palabra de información requiere que la controladora tome el control del bus, que es compartido con el resto de componentes del sistema. Dado que el dispositivo envía la información a una velocidad constante, es posible encontrarnos con una palabra para transferir a memoria y no disponer del control del bus en ese momento. Esto originaría que la controladora necesitase almacenar la información mientras espera el bus. Si éste estuviera muy ocupado, puede ser necesario almacenar bastante información. Por este motivo, la controladora, en lugar de disponer de un registro de datos, dispone de un *buffer*, con objeto de almacenar un bloque de información, y agilizar las transferencias a y desde la memoria, evitando la pérdida de datos.

## 9.5 Optimización de las operaciones de E/S

Dado que uno de los objetivos del sistema de E/S es la eficiencia en la realización de las operaciones de E/S, en este apartado se van a estudiar técnicas que permiten mejorar el rendimiento.

### 9.5.1 Técnicas de *buffering*

Un *buffer* es un área de memoria que almacena datos temporalmente. Su objetivo principal es suavizar las diferencias de velocidad entre los dispositivos y el procesador.

La técnica de *buffering*, descrita en el capítulo 1, se puede implementar tanto en el sistema de E/S como por hardware. De hecho, los registros de datos de las controladoras suelen estar duplicados para actuar como un *buffer*.

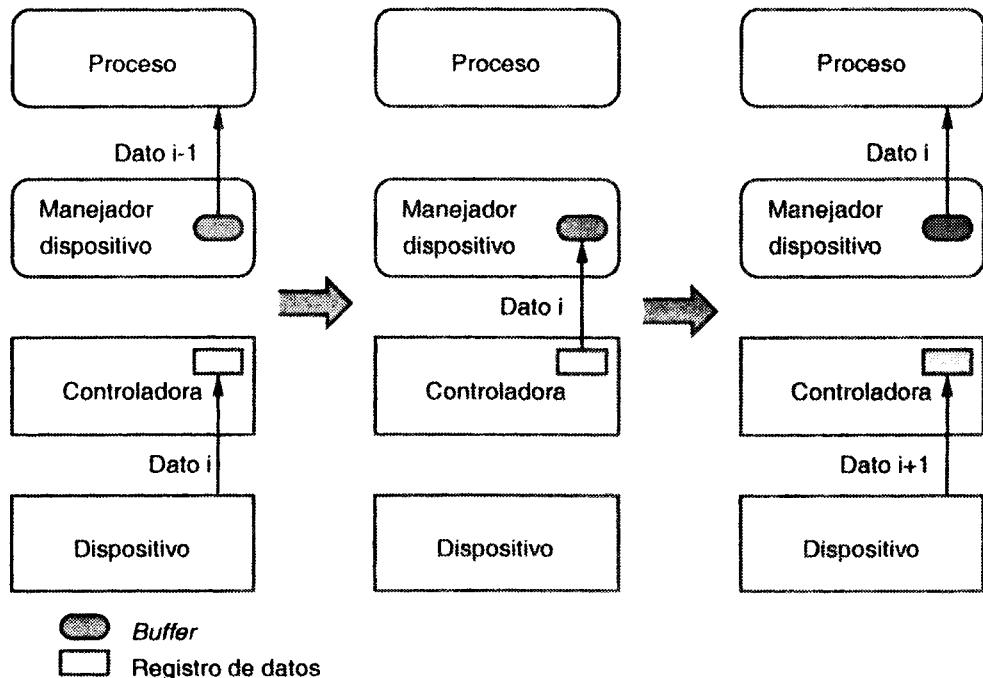


Figura 9.7: Operación de entrada con *buffer*

Esta técnica puede ser aplicada tanto a dispositivos de bloques como de caracteres. En ambos casos, el *buffer* debe tener el tamaño adecuado para almacenar

un bloque o carácter, según el dispositivo.

El *buffering* permite reducir la cantidad de tiempo que el proceso espera para leer o escribir una palabra, como muestra la figura 9.7. En ella, mientras el dispositivo realiza la lectura del dato  $i + 1$ , el procesador está trabajando con el dato  $i$ . En este caso, se consigue con la incorporación de un *buffer* en el manejador del dispositivo.

Se puede realizar una comparación de rendimiento entre un sistema que incorpora un *buffer* frente a otro que no lo posee. Sea  $T$  el tiempo para realizar la lectura de un dato, y  $C$  el tiempo de procesarlo.

Si no se emplea *buffer*, el tiempo de ejecución por dato es  $T + C$ . Sin embargo, con el empleo del *buffer* el tiempo viene dado por  $\max(T, C) + M$ , siendo  $M$  el tiempo necesario para mover los datos desde el *buffer* del manejador al espacio de usuario, que suele ser menor que los anteriores.

La técnica del *buffering* se puede mejorar empleando dos *buffers*. De este modo, mientras un proceso está transfiriendo datos desde uno de ellos, la controladora puede estar empleando el otro. Esta técnica se conoce como **doble buffer** o **intercambio de buffers**.

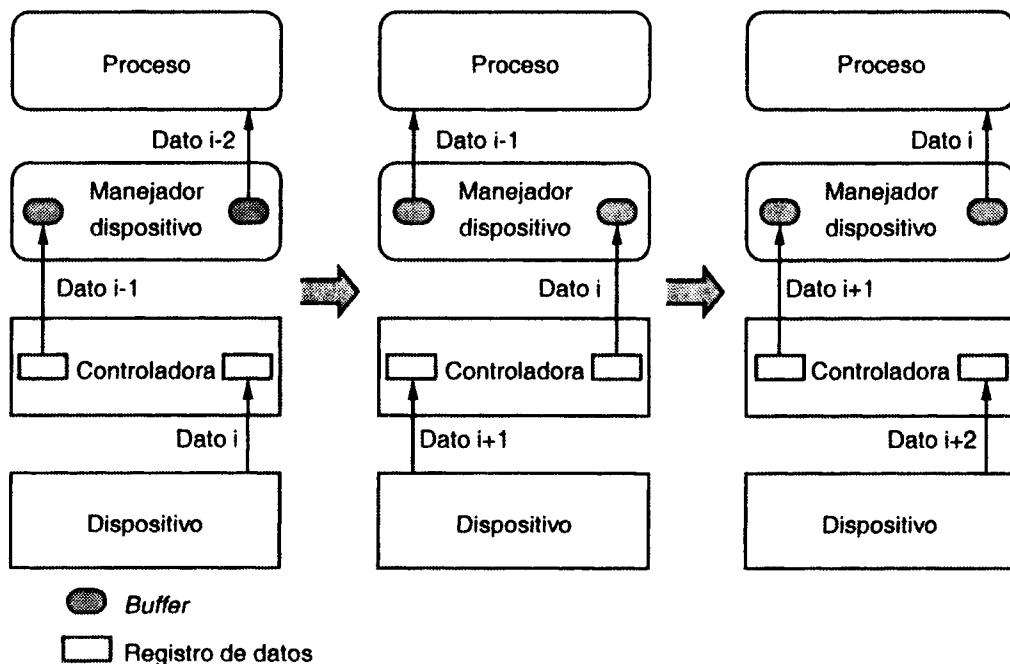
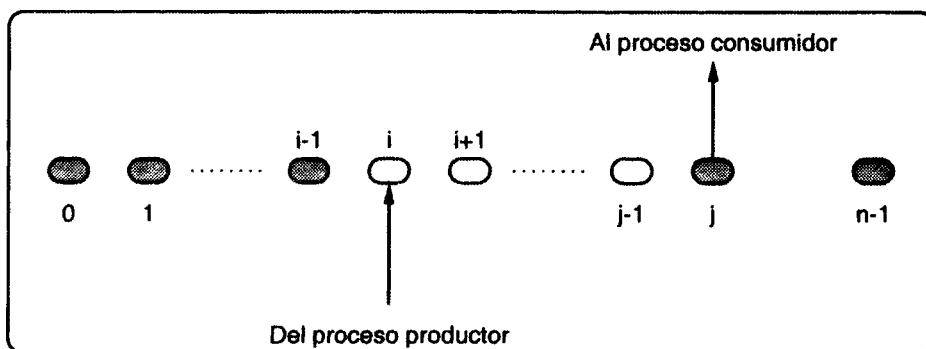


Figura 9.8: Doble buffer

Es posible mejorar esta técnica aumentando el número de *buffers* de 2 a  $N$ , recibiendo el nombre de **buffer circular**. La utilización del mismo se convierte en el problema de sincronización que se establece entre un proceso productor y otro consumidor. De esta forma, el proceso productor (la controladora en operaciones de lectura y el procesador en escritura) escribe en el *buffer i*, mientras que el proceso consumidor (controladora en escritura y procesador en lectura) lee del *buffer j*. La figura 9.9 muestra esta técnica. En ella, los *buffers j* hasta  $n - 1$  y desde 0 hasta  $i - 1$  están llenos, y pueden ser leídos por el consumidor, mientras el productor puede llenar los *buffers i* a  $j - 1$ .



**Figura 9.9:** Buffer circular

### 9.5.2 Caché de disco

Una caché de disco es una ampliación del concepto de *buffer* explicado anteriormente. La caché contiene datos de sectores del disco que, debido a que se encuentran en memoria principal, su acceso es más rápido que al dato original. El funcionamiento es el siguiente, cuando un proceso solicita una petición de E/S, se comprueba si el dato se encuentra en la caché, en cuyo caso no es necesario realizarla. En caso contrario, se inicia la operación almacenando el sector leído en la caché para futuras referencias.

La caché y el *buffer*, aunque son parecidos, presentan distintas funciones. Mientras el *buffer* mantiene datos de la operación en curso, la caché mantiene datos de las últimas operaciones realizadas.

En el diseño de esta caché es necesario tener en cuenta dos aspectos, el trasiego de información entre la caché y el proceso de usuario, y la política de sustitución. Respecto al primero, cuando se realiza una petición de E/S y los datos se encuentran en la caché, éstos deben ser entregados al proceso. En este caso, podemos

copiar los datos de la caché al espacio de usuario, o emplear la zona de la caché como una memoria compartida, devolviendo al proceso un puntero a ella. Esta última posibilidad evita el trasiego de información dentro de la memoria, además de permitir compartir la información entre varios procesos.

El otro aspecto viene motivado por el tamaño limitado de la caché de disco. Esto origina que cuando se realiza una operación de entrada y la caché esté llena, se deberá sustituir un dato antiguo por el nuevo. Este problema es idéntico al de sustitución de páginas (ver capítulo 8), y se pueden emplear los mismos algoritmos.

En algunos sistemas, como LINUX, la caché de disco se emplea también como *buffer*, es decir, las operaciones de E/S se realizan sobre un *buffer*, que formará parte de la caché de disco del propio sistema. Ésta recibe el nombre de **caché de buffers**.

### 9.5.3 Planificación de discos

Uno de los dispositivos de uso más frecuente es el disco, de forma que el rendimiento del sistema se va a ver afectado por su velocidad de acceso. Aunque en los últimos años se han producido grandes avances en la velocidad de acceso a éstos, las diferencias con los procesadores y la memoria ha ido en aumento. Por este motivo, se ha prestado una gran atención a la forma de realizar las operaciones con estos dispositivos para mejorar su rendimiento.

#### 9.5.3.1 Parámetros de rendimiento del disco

Todos los discos o CD-ROM se organizan en uno o más **platos**, cada uno con una o dos **superficies**. Cada superficie se divide lógicamente en varios **sectores** definidos como una porción angular del círculo del disco, similar a las porciones de una tarta. Cada superficie del disco se organiza en varias **pistas** concéntricas que pasan por cada sector. El conjunto de pistas en las diferentes superficies se denomina **cilindro**. La figura 9.10 muestra los distintos elementos de un disco. Cuando se realiza una petición de E/S, el sistema debe determinar en qué disco, superficie, pista y sector se encuentra, para poder posicionar la cabeza de lectura/escritura y realizar la transferencia.

Cuando se recibe una petición de E/S sobre un disco, el sistema debe posicionar la cabeza de lectura/escritura en la pista correspondiente. El tiempo que se tarda en realizarlo se conoce como **tiempo de búsqueda**. Éste se puede aproximar mediante la siguiente fórmula:

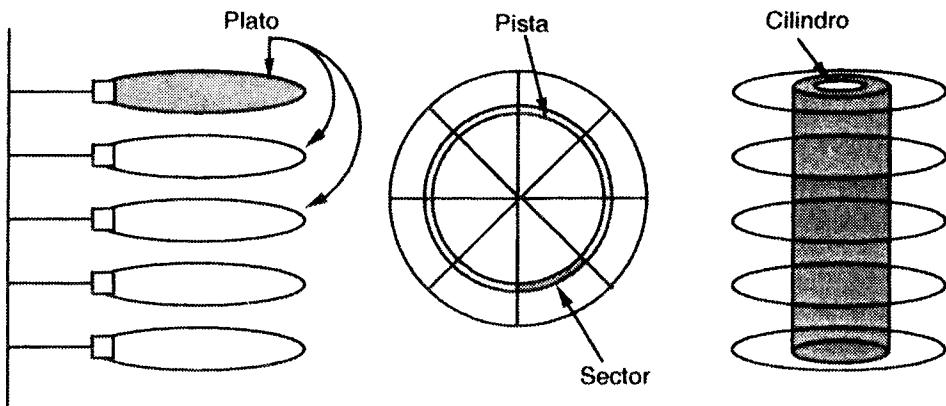


Figura 9.10: Componentes de un disco

$$T_s = m \times n + s$$

donde  $T_s$  es el tiempo de búsqueda estimado,  $m$  es el tiempo que tarda en pasar de una pista a otra, que depende de la unidad de disco,  $n$  es el número de pistas que va a recorrer, y  $s$  es el tiempo de arranque.

Una vez que la pista está seleccionada, la controladora del disco espera hasta que el principio del sector se sitúa debajo de la cabeza. El tiempo que tarda en producirse esta operación se conoce como **demora rotacional** o **tiempo de latencia**. Éste depende de la velocidad de rotación del dispositivo, que varía en función del tipo de disco. Así, un disquete gira a 360 rpm, es decir, una revolución cada 167 milisegundos, lo que supone un retardo medio de 83 milisegundos. Las unidades de discos duros tienen una velocidad de rotación de 7200 rpm o superior, lo que supone una latencia media de 4 milisegundos. La suma del tiempo de búsqueda y de la demora rotacional es el **tiempo de acceso**, es decir, el tiempo que tarda en situarse la cabeza en la posición adecuada donde tiene que leer o escribir.

Una vez que la cabeza está posicionada, la operación de lectura o escritura se realiza a medida que el sector se mueve bajo ésta. Al tiempo que se tarda en hacer la transferencia se denomina **tiempo de transferencia**, que viene dado por:

$$T = \frac{b}{r \times N}$$

donde  $T$  es el tiempo de transferencia,  $b$  el número de bytes a transferir,  $N$  el

número de bytes por pistas, y  $r$  es la velocidad de rotación en revoluciones por segundo.

Por tanto, el **tiempo de acceso total** se puede expresar como la suma de los tres tiempos anteriores, que comprende buscar la pista, después el sector y, finalmente, transferir la información. Este tiempo se expresa como sigue:

$$T_a = T_s + \frac{1}{2} \times r + \frac{b}{r \times N}$$

De estos tres componentes, el tiempo de búsqueda es el que tiene más peso en el total. Dado que en un sistema de tiempo compartido es habitual que haya varias peticiones pendientes de E/S al disco, éstas se podrían organizar de tal forma que se minimice el tiempo medio de acceso total. Esto es lo que se llama **planificación del disco**. Las características que debe tener un buen algoritmo de planificación son las siguientes:

- Maximizar la cantidad de peticiones servidas por unidad de tiempo.
- Minimizar el tiempo medio de retorno, es decir, el tiempo promedio de espera más el tiempo promedio de servicio.
- Minimizar la varianza, es decir, la desviación de una petición particular respecto al promedio.

A continuación veremos algunos de los algoritmos que se pueden utilizar.

#### 9.5.3.2 Algoritmo FIFO

La forma más simple de servir las peticiones consiste en respetar su orden de llegada (FIFO). Éste es un método justo de servirlas, pero puede conducir a tiempos de espera muy largos. FIFO exhibe un patrón de búsqueda aleatorio, en el cual peticiones sucesivas pueden ocasionar búsquedas costosas, que impliquen ir de los cilindros más internos a los más externos o viceversa. La figura 9.11 muestra un ejemplo de aplicación de este algoritmo a un conjunto de peticiones.

#### 9.5.3.3 Algoritmo SSTF

El algoritmo SSTF<sup>2</sup> selecciona la petición que requiere el menor tiempo de búsqueda, es decir, la que necesita recorrer un menor número de pistas desde la

---

<sup>2</sup>Shortest-Seek-Time-First.

Peticiones: 95, 179, 35, 119, 15, 125, 65, 69  
 Posición inicial de la cabeza: 53  
 Sentido ascendente  
 N° de pistas recorridas = 632

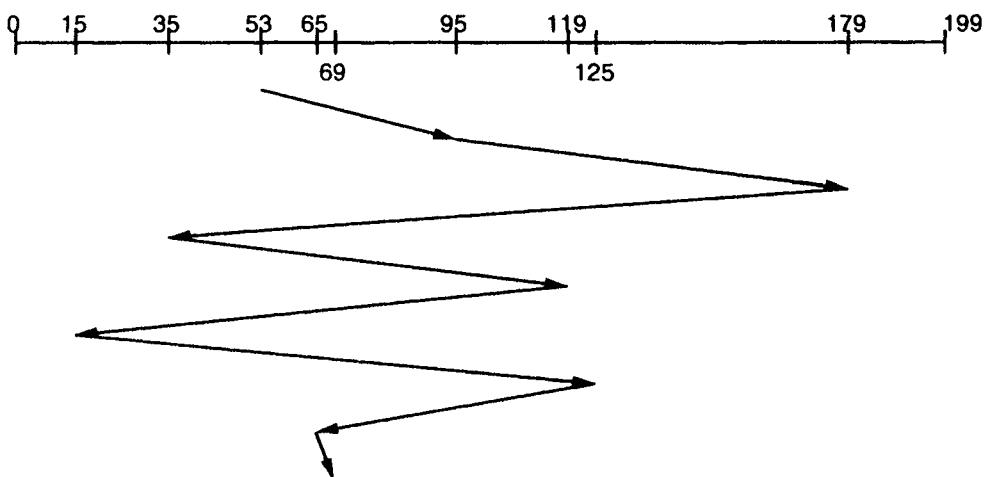


Figura 9.11: Algoritmo FIFO

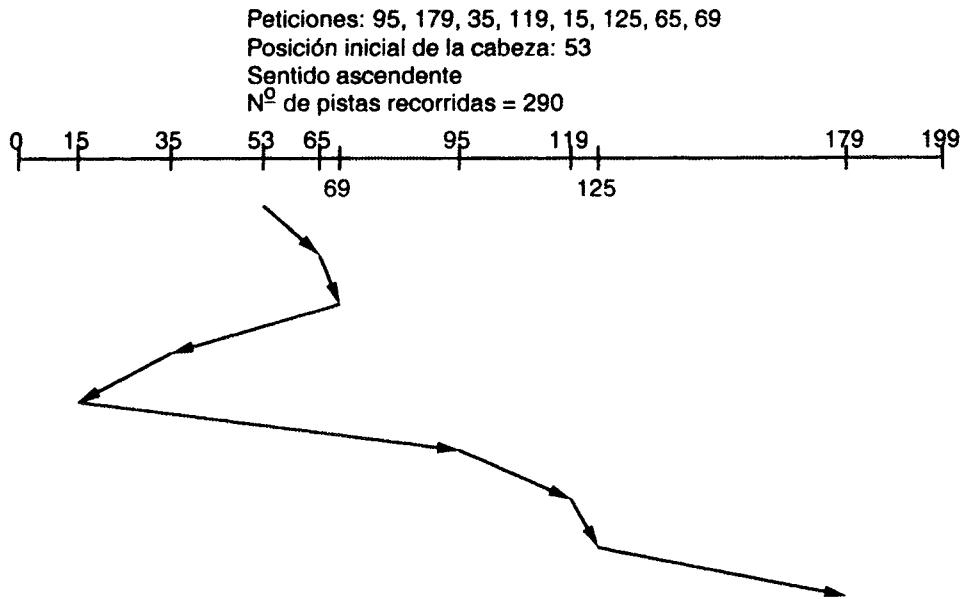
posición actual de la cabeza del disco. Por supuesto, esto no garantiza que el tiempo de búsqueda promedio para varios movimientos del brazo del disco vaya a ser mínimo. Sin embargo, suele dar mejores resultados que la política FIFO.

En la figura 9.12 se muestra el comportamiento de este algoritmo para la misma lista de peticiones anterior. En este caso, el número de pistas recorridas es 290, frente a las 632 del FIFO. Aunque el algoritmo SSTF proporciona mejores resultados que FIFO no es la estrategia óptima, ya que minimizar el tiempo de búsqueda de la siguiente petición no garantiza obtener el menor tiempo de búsqueda medio. Así, en el ejemplo anterior, si movemos la cabeza de la pista 53 a la 35, aunque ésta no sea la más cercana, y después a la 15, antes de dar servicio a 65,69,95,119,125 y 179, el movimiento total sería de 202 pistas.

Un problema que puede presentar el algoritmo SSTF es la inanición de aquellas solicitudes que se encuentran muy alejadas de la posición actual de la cabeza del disco. Si ésta se encuentra en un extremo y llega una solicitud del otro, puede ser que no se atienda si continuamente llegan peticiones del extremo donde nos encontramos. Esto hace que SSTF no sea aceptable en sistemas interactivos, ya que la varianza de los tiempos de retorno es alta.

SSTF necesita disponer de una regla de arbitraje, similar a la de los algoritmos

de planificación de la CPU, para decidir en los casos de igualdad.



**Figura 9.12:** Algoritmo SSTF

#### 9.5.3.4 Algoritmo SCAN

El algoritmo SSTF puede dejar discriminadas ciertas peticiones que implican un movimiento grande de la cabeza del disco. Para vencer este problema, Denning [Denn67] propuso el algoritmo SCAN.

En éste, la cabeza se mueve de un extremo del disco al otro, de forma que cuando llega a uno de ellos se invierte el sentido de su movimiento. Durante su recorrido va atendiendo todas las peticiones que encuentra. Así, peticiones recién llegadas pueden ser atendidas inmediatamente, mientras que las que afectan a posiciones por las que ha pasado recientemente tendrán que esperar hasta que la cabeza cambie de sentido.

Este algoritmo requiere conocer tanto la posición actual de la cabeza como el sentido del movimiento, para lo que emplea un bit de sentido.

En nuestro ejemplo, si la cabeza se está moviendo hacia la pista 199, su movimiento sería el que se muestra en la figura 9.13.

Este algoritmo elimina la posibilidad de inanición del SSTF, ofreciendo un

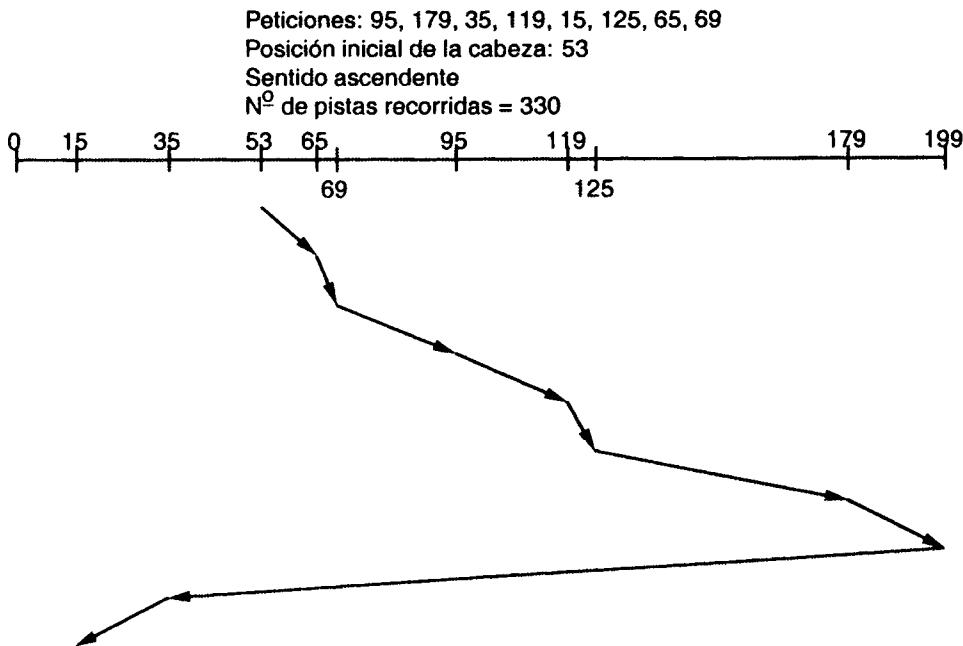


Figura 9.13: Algoritmo SCAN

rendimiento similar. Sin embargo, favorece a las pistas interiores debido a que en una vuelta del disco pasa dos veces por ellas, mientras que los extremos sólo los visita una vez. No obstante presenta una varianza mucho más baja que el algoritmo SSTF.

#### 9.5.3.5 Algoritmo LOOK

En la práctica, SCAN suele implementarse sin que sea necesario llegar hasta los extremos para invertir el sentido de la cabeza del disco. Esta variante se denomina algoritmo LOOK o del ascensor. La figura 9.14 muestra el comportamiento de este algoritmo en nuestro ejemplo.

#### 9.5.3.6 Algoritmos C-SCAN y C-LOOK

Una modificación interesante de la estrategia SCAN es C-SCAN, también denominado SCAN circular o exploración circular. En C-SCAN, la cabeza se mueve de un extremo a otro sirviendo las peticiones que encuentra en su camino. Cuan-

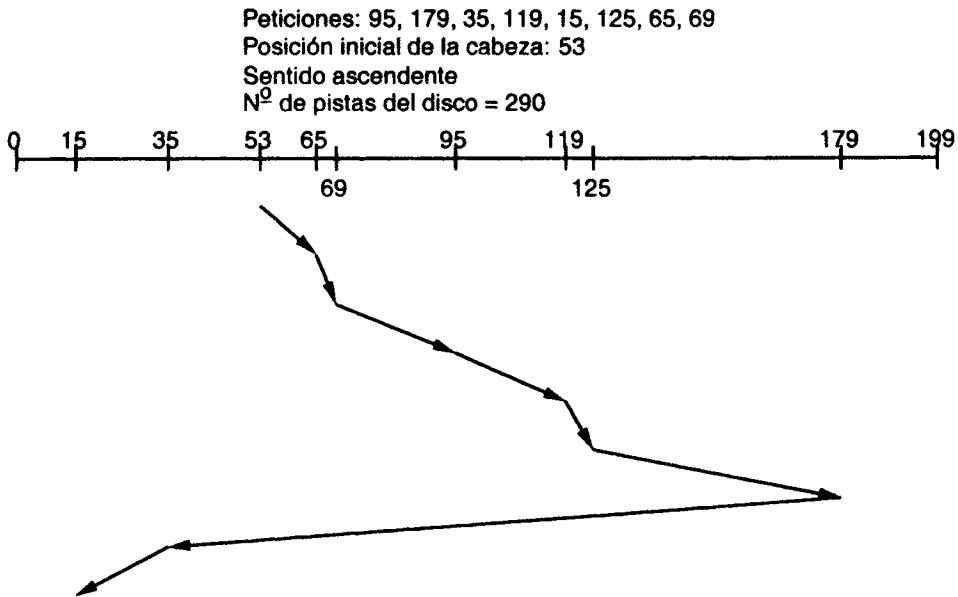


Figura 9.14: Algoritmo LOOK

do llega al extremo salta al otro sin servir peticiones, volviendo a comenzar su recorrido. Este algoritmo considera al disco como si la última pista y la primera fueran adyacentes. Así, C-SCAN elimina completamente la discriminación de los extremos que presenta SCAN, presentando una varianza menor que éste.

Estudios realizados han mostrado que el algoritmo SCAN produce buenos resultados cuando la carga del sistema es baja, sin embargo, cuando la carga es alta C-SCAN se comporta mejor.

La figura 9.15 muestra el orden en el que se atenderían las distintas solicitudes de nuestro ejemplo con el algoritmo C-SCAN. En ella, la flecha con trazo discontinuo representa el salto de la cabeza de un extremo a otro sin servir peticiones.

Del mismo modo, existe un algoritmo C-LOOK que consiste en modificar la estrategia LOOK, tratando las pistas del disco como una lista circular, igual que en C-SCAN. La figura 9.16 muestra el comportamiento de este algoritmo en nuestro ejemplo.

Estas dos políticas suponen la existencia de una orden especial del dispositivo que mueve la cabeza de un extremo a otro en un breve espacio de tiempo, en relación al movimiento sobre cualquier otra pista.

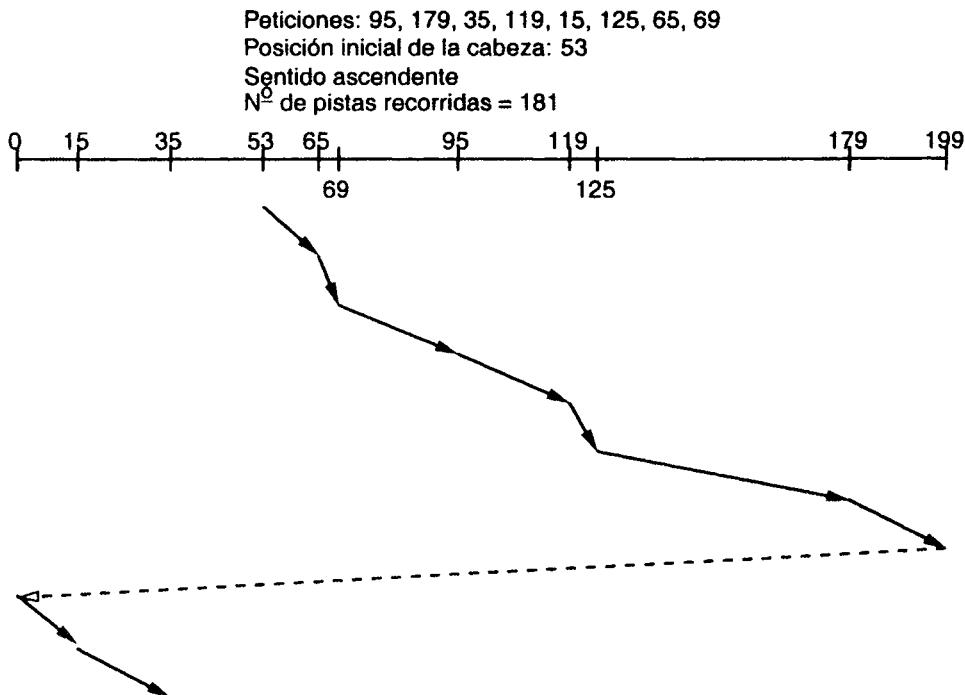


Figura 9.15: Algoritmo C-SCAN

### 9.5.3.7 Algoritmos N-SCAN y FSCAN

El algoritmo N-SCAN es una modificación interesante de SCAN. En ella la cola de peticiones se divide en varias de longitud  $N$ . Cada una se procesa independientemente usando SCAN. Mientras se está procesando una de ellas, las nuevas peticiones que van llegando se ponen en otra distinta. Si al terminar de procesar una cola hay menos de  $N$  peticiones disponibles, todas ellas se procesan a continuación.

El rendimiento de este algoritmo depende del valor de  $N$ . Así, para valores grandes, su rendimiento se aproxima al de SCAN; si  $N = 1$ , el algoritmo se convierte en un FIFO.

El algoritmo de planificación FSCAN utiliza dos colas. Cuando comienza una exploración, todas las peticiones están en una de ellas, estando la otra vacía. El algoritmo que emplea para atender las peticiones es SCAN. Todas las peticiones nuevas que llegan durante su recorrido se introducen en la cola que estaba inicialmente vacía.

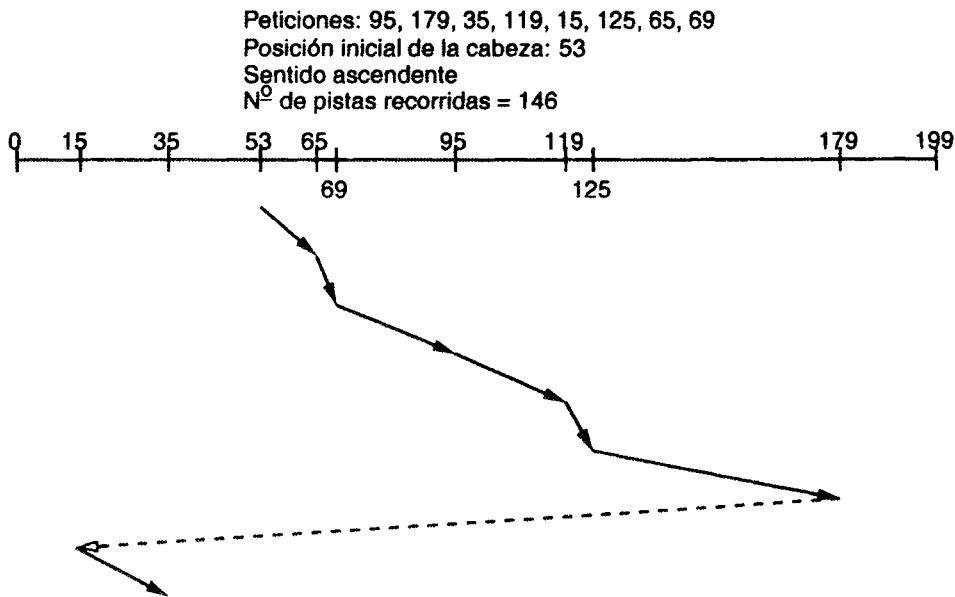


Figura 9.16: Algoritmo C-LOOK

## 9.6 E/S en LINUX

Una característica que diferencia a **LINUX** de otros sistemas operativos es el tratamiento de los dispositivos de E/S, puesto que éstos se tratan como ficheros, que se denominan **ficheros especiales**. Éstos se almacenan en el directorio **/dev**, de esta forma, al tener integrados los dispositivos de E/S en el sistema de ficheros, se logra una mayor flexibilidad de acceso.

Cada dispositivo se identifica por una clase (bloques o caracteres), un **número mayor** y un **número menor**. La clase permite acceder a la tabla de configuración de los dispositivos, que apunta a los manejadores de éstos. El número mayor indica el tipo de dispositivo, que se usa como índice en la tabla anterior para acceder al manejador correspondiente. El número menor se utiliza para identificar un dispositivo físico entre varios del mismo tipo.

### 9.6.1 Técnicas de optimización

Para los dispositivos de bloques, **LINUX** dispone de una caché de **buffers** situada entre los manejadores y el sistema de ficheros. Ésta es una tabla en el núcleo que contiene los bloques de uso reciente, y se utiliza tanto en las operaciones de

escritura como en las de lectura. El algoritmo de sustitución que emplea es el LRU. Periódicamente se realizan escrituras de los bloques que han sido modificados para minimizar las potenciales inconsistencias en los sistemas de ficheros.

El tamaño de la caché de *buffers* puede ser un parámetro importante para la eficiencia de un sistema, ya que, si es suficientemente grande, el porcentaje de éxitos en caché puede ser grande y el número de transferencias de E/S bajo.

Otro aspecto de optimización a tener en cuenta es la planificación de peticiones al disco. LINUX emplea el algoritmo LOOK.

Los dispositivos de caracteres no usan la caché de *buffers* porque trabajan con flujos de caracteres para pasar la información entre el dispositivo y la memoria. Algunos de ellos utilizan estructuras de datos especiales llamadas **listas-C**<sup>3</sup>, que son *buffers* circulares.

Cada lista-C está formada por un contador y una lista de bloques de tamaño variable. Al llegar los caracteres de las terminales y otros dispositivos de caracteres, éstos se almacenan en la lista. Permiten la manipulación de datos carácter a carácter o en grupos. Es útil para la transmisión de pequeños volúmenes de datos típicos de dispositivos lentos tales como terminales.

### 9.6.2 Llamadas al sistema

La E/S en un sistema LINUX se maneja principalmente usando las llamadas al sistema que aparecen en la tabla 9.1.

Llamada	Descripción
open	Solicitar un dispositivo.
close	Liberar un dispositivo.
read	Leer de un dispositivo.
write	Escribir en un dispositivo.
lseek	Posicionarse en un dispositivo.
ioctl	Establecer opciones al dispositivo.

Tabla 9.1: Llamadas al sistema relacionadas con los dispositivos

<sup>3</sup> C-list: character list.

## 9.7 Resumen

El sistema de E/S es el componente del sistema operativo encargado de la realización de operaciones de E/S. Sus objetivos principales son el rendimiento y la uniformidad.

Los dispositivos de E/S permiten el intercambio de información entre el sistema y el mundo exterior. Éstos se conectan al sistema a través de las controladoras, que actúan como intermediarias. El objetivo de éstas es superar las incompatibilidades de velocidad, señalización, traducción de órdenes de E/S genéricas a controles específicos del dispositivo, etc.

El sistema operativo, a través de los manejadores de dispositivos, efectúa las operaciones de E/S escribiendo las órdenes oportunas en los registros de la controladora. Dado que existe una gran variedad de dispositivos, para conseguir la uniformidad a la hora de realizar una operación de E/S, se implementa una interfaz común para todos ellos, la API. De este modo, añadir un nuevo dispositivo consiste en agregar su manejador respetando la interfaz existente con los usuarios.

Para realizar las operaciones de E/S existen tres técnicas, la E/S controlada por programa, la E/S controlada por interrupciones y el acceso directo a memoria. En la primera, la CPU es la encargada de realizar las transferencias entre la memoria y los registros de datos de la controladora. En concreto, es el manejador del dispositivo el que copia los datos del área del espacio de usuario a los registros de la controladora y viceversa. El problema es la existencia de una espera activa mientras se realiza una operación de E/S, debido a la necesidad de interrogar de forma repetida el estado del dispositivo.

Una alternativa a este enfoque es la E/S por interrupciones. En este caso, la controladora notificará al sistema operativo de forma automática el fin de la operación de E/S mediante una interrupción. De este modo, el procesador tras emitir una orden de E/S a la controladora puede continuar con la ejecución de otro proceso. Al finalizar la operación de E/S, la controladora producirá una interrupción para realizar la transferencia de datos entre ella y el sistema.

Por último, el acceso directo a memoria permite la realización de operaciones de E/S directamente con la memoria, sin necesidad de interrumpir al procesador para realizar la transferencia. Esta técnica es más eficiente que las anteriores, sobre todo en grandes volúmenes de datos.

Otro aspecto importante del sistema de E/S es el rendimiento de las operaciones, debido a la diferencia en velocidad entre el procesador y los dispositivos. Así, para los dispositivos de caracteres se puede aplicar una técnica de *buffering* para simultanejar la realización de una operación de E/S con una de cálculo. En

el caso de dispositivos de bloques se dispone de la caché de disco, que consiste en tener en memoria los últimos bloques transferidos. De este modo, cuando se necesite un bloque del dispositivo, es posible que se encuentre en la caché, por lo que no hay que realizar transferencia alguna.

Para los discos, podemos realizar una planificación de las peticiones con objeto de minimizar el tiempo medio de acceso. Existen diversos algoritmos de planificación como FIFO, SSTF, LOOK, etc., aunque diferentes estudios realizados indican que la mejor política podría operar en dos etapas. Cuando la carga es baja, la política SCAN es la mejor. Para una carga media a alta, C-SCAN conduce a los mejores resultados.

## 9.8 Ejercicios

1. ¿Qué mecanismos para realizar una operación de E/S se basan en las interrupciones?
2. En un disco de 200 pistas nos encontramos situados en la pista número 53 realizando un movimiento ascendente. Supongamos la siguiente secuencia de peticiones y el algoritmo de planificación de discos correspondiente:
  - (a) SCAN y 54,55,56,1,57,58, ..., 199,200,199,198, ..., 2
  - (b) SSTF y 58,2,59,60,59,60,59,60, ..., 59,60,59,60, ...

Determine si existe algún problema; en tal caso, descríbalo y explique cómo se podría solucionar.

3. En el algoritmo de planificación de discos SCAN-N, ¿qué se puede decir si  $N$  tiene valor 1? ¿Y en el caso de que  $N$  sea muy grande? Razone las respuestas.
4. ¿Cuáles son las características de un dispositivo de bloques?
5. Es posible que mientras se atiende una solicitud de disco para una pista llegue otra solicitud para la misma pista. Algunos algoritmos de planificación de disco atenderán esta nueva solicitud inmediatamente después de procesar la solicitud actual. ¿Qué inconvenientes presenta? Razone la respuesta.
6. ¿Cuál es la utilidad de la tabla de referencias?
7. ¿Por qué en LINUX la caché de disco recibe el nombre de caché de *buffers*?
8. Un sistema realiza una técnica de *buffering* circular. ¿Qué diferencias y parecidos hay entre ésta y la caché de disco?

9. Suponga un disco duro de 512 bytes por sector, 96 sectores por pista, 110 pistas por superficie y ocho superficie útiles. El disco gira a 3600 rpm. Se quiere realizar una operación de E/S para leer un registro de 120 bytes, que se encuentra almacenado en un sector. El procesador lee un sector del disco mediante E/S dirigida por interrupciones, con una interrupción por cada byte. Si se tarda 2,5 microsegundos en procesar cada interrupción, ¿qué porcentaje del tiempo gastará el procesador en gestionar la E/S (no considere el tiempo de búsqueda)?
10. Repetir el problema anterior utilizando DMA y suponiendo que se produce una interrupción por sector.
11. Disponemos de un disco con 300 pistas, numeradas de 1 a 300. Nos encontramos en la pista número 189 en sentido ascendente. Supongamos la siguiente secuencia de peticiones:

25, 30, 80, 1, 289, 275, 298, 150, 96, 164, 3, 75, 122, 200, 202

Calcule el número de movimientos de la cabeza lectora para los siguientes algoritmos:

- (a) FIFO.
  - (b) SCAN.
  - (c) SCAN-N siendo N=5.
  - (d) LOOK-C.
  - (e) SSTF.
  - (f) El algoritmo empleado en LINUX.
12. Resuelva el problema anterior suponiendo que el sentido es descendente.

# Capítulo 10

## Sistemas de ficheros

---

El sistema operativo, a través del sistema de ficheros, proporciona al usuario un mecanismo para almacenar información de forma permanente. En este capítulo abordaremos la implementación de un sistema de ficheros, determinando cómo es la interfaz de éste con los usuarios mediante el estudio de los ficheros y directorios. Posteriormente veremos las distintas estrategias para la gestión del espacio libre, asignación de espacio a los ficheros y los métodos existentes para mejorar el rendimiento.

### 10.1 Introducción

El objetivo principal de un sistema es ejecutar programas. Éstos junto con los datos a los que acceden deben encontrarse en memoria principal durante la ejecución. En una situación ideal, nos gustaría que todos los programas y datos residieran permanentemente en la memoria principal. Esto no es posible por dos razones, la memoria es pequeña para contener todos los datos de forma permanente y, además, es un dispositivo de almacenamiento volátil, por lo que pierde su contenido al apagar el sistema.

No sólo existen los problemas de necesidad de almacenamiento, sino que también puede presentarse la necesidad de que varios procesos accedan a distintas partes de esa información al mismo tiempo. Si esta información está almacenada

dentro del espacio de direcciones de un proceso, sólo éste podría acceder a ella. La forma de resolver este problema es hacer que la información sea independiente de cualquier proceso.

Tenemos, por tanto, tres requisitos esenciales para el almacenamiento de información a largo plazo:

1. Almacenar una gran cantidad de información.
2. La información debe permanecer después de la terminación del proceso que la está usando.
3. Múltiples procesos deben ser capaces de acceder a la información concurrentemente.

La solución a estos problemas consiste en almacenar la información en medios externos (discos, cintas, CD-ROM, etc.), en unidades llamadas **ficheros**. Un fichero es una colección de datos con un nombre que suele residir en un dispositivo de almacenamiento secundario.

Los ficheros son manejados por el sistema operativo mediante el denominado **sistema de ficheros**. Éste es muy importante dentro del sistema operativo, ya que es la parte más visible por el usuario.

No se puede, ni se debe hacer una separación radical entre el capítulo anterior sobre gestión de dispositivos y el que estamos estudiando. Aunque todas las operaciones de E/S que se realizan en un sistema no están relacionadas con los ficheros, si lo están una gran parte de ellas, por lo que es interesante ver el punto de contacto entre ambos capítulos. En la figura 10.1 se muestra cómo se relacionan estos componentes del sistema. Podemos ver cómo las operaciones en las que están implicados ficheros, tienen que ir a través del sistema de ficheros antes de acceder al dispositivo mediante su manejador.

## 10.2 Funciones del sistema de ficheros

Para comprender mejor las funciones que realiza el sistema de ficheros, veamos qué operaciones se realizan cuando un usuario da una orden sobre un fichero.

Los usuarios y los programas de aplicación interaccionan con el sistema de ficheros por medio de órdenes para la creación, borrado y la realización de operaciones sobre ficheros. Antes de realizar cualquier operación, el sistema de ficheros debe identificar y localizar el fichero seleccionado. Esto requiere el uso de alguna

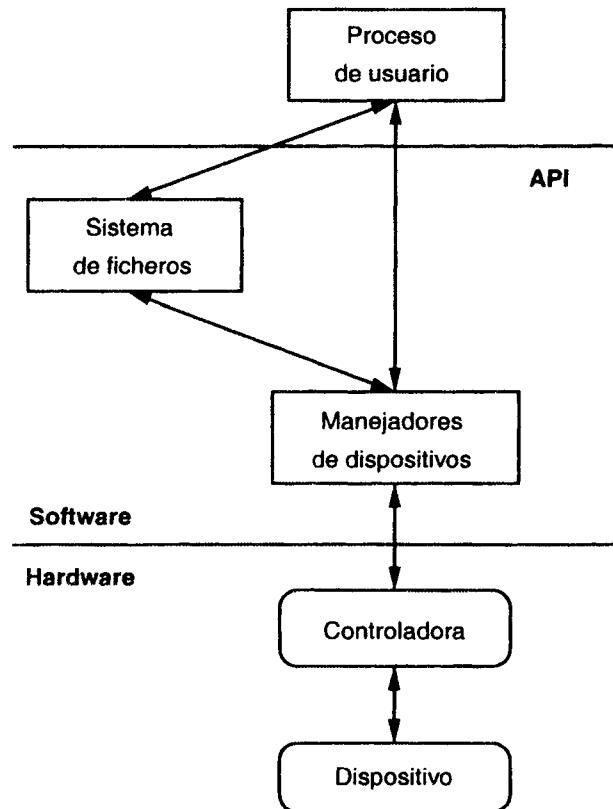


Figura 10.1: Relación entre el sistema de ficheros y el de E/S

estructura que describa la localización de todos los ficheros más sus atributos. Ésta se suele denominar **directorio**. Además, la mayor parte de los sistemas de tiempo compartido tienen un sistema de control de acceso de los usuarios a los ficheros, así sólo pueden acceder a un fichero los usuarios autorizados y de una determinada forma.

El usuario o aplicación ve el fichero con una estructura de registros. De esta forma, las operaciones básicas que se pueden realizar sobre un fichero son llevadas a cabo sobre los registros individuales. Así, para traducir órdenes de usuario en órdenes específicas de manipulación de ficheros debe emplearse el método de acceso adecuado a la estructura del fichero.

Mientras que los usuarios y las aplicaciones están interesados en los registros, la E/S se hace por bloques. Así, para la realización de una operación de E/S sobre un fichero sus registros deben ser agrupados en bloques. Para el manejo de

los dispositivos de bloques el sistema debe disponer de técnicas para el manejo del espacio libre y para la asignación de bloques a los ficheros.

En la tabla 10.1 podemos ver las funciones del sistema de ficheros, donde se separan las que forman parte de la visión del usuario, el **sistema de ficheros lógico**, y las del sistema operativo, el **sistema de ficheros físico**. Por tanto, a la hora de diseñar el sistema de ficheros nos encontramos con dos problemas. El primero de ellos es definir cómo aparecerá éste ante el usuario, es decir, cómo se define un fichero, sus atributos, operaciones permitidas, y la estructura de los directorios. El otro son los algoritmos y estructuras de datos que deben crearse para establecer una correspondencia entre el sistema de ficheros lógico y el físico. En este capítulo estudiaremos ambas visiones del sistema de ficheros, su interfaz y su diseño, respectivamente.

<b>Sistema de ficheros lógico</b>	
Gestión de directorios	Creación de ficheros Borrado de ficheros Localización de ficheros
Gestión de ficheros	Método de acceso a la información
Control de acceso	Operaciones permitidas al usuario
<b>Sistema de ficheros físico</b>	
Agrupamiento de registros en bloques	Fijo Longitud variable extendido Longitud variable no extendido
Asignación de espacio del disco	Contigua Encadenada Indexada
Gestión del espacio libre	Lista enlazada Mapa de bits
Fiabilidad	Consistencia del sistema Copias de seguridad

**Tabla 10.1:** Funciones del sistema de ficheros

### 10.3 Interfaz del sistema de ficheros

Desde el punto de vista del usuario los aspectos a tener en cuenta en un sistema de ficheros son la estructura del fichero, la forma en que éstos se nombran y protegen, las operaciones que se permiten hacer con ellos, etc.

### 10.3.1 Ficheros

Los ficheros son un mecanismo de abstracción que proporcionan una forma de almacenar información en dispositivos de almacenamiento secundario. Éstos permiten ocultar al usuario los detalles de cómo y dónde está almacenada la información, así como la forma en que funcionan los dispositivos de almacenamiento.

Todos los sistemas operativos asocian cierta información a cada fichero, por ejemplo, su nombre, tipo, la fecha y la hora en que el fichero fue creado, su tamaño, su localización, etc. A estos datos se les llaman **atributos del fichero**. La lista de atributos puede variar considerablemente de un sistema a otro, así como los valores que éstos pueden tomar.

Los ficheros no suelen ser elementos estáticos, por el contrario, suelen sufrir numerosas modificaciones y manipulaciones durante su vida. Los sistemas operativos proporcionan servicios para realizar operaciones sobre los ficheros, que son independientes del dispositivo físico donde están almacenados. Las operaciones básicas que se pueden realizar son:

- Creación.
- Escritura.
- Lectura.
- Búsqueda.
- Borrado.

Estas operaciones son las mínimas que se deben proporcionar, y a partir de ellas, combinándolas, se pueden obtener otros tipos de operaciones comunes.

Muchos sistemas operativos soportan varios tipos de ficheros. LINUX, por ejemplo, distingue entre ficheros regulares, directorios y ficheros especiales de dispositivos. Los ficheros regulares son los que contienen información del usuario. Los directorios mantienen información sobre la estructura del sistema de ficheros, y los ficheros especiales están relacionados con la E/S.

El sistema operativo MS-DOS, que admite extensiones en los nombres de ficheros, utiliza éstas para distinguir diferentes tipos de ficheros y las operaciones que se pueden realizar con éstos. Por ejemplo, sólo un fichero con una extensión .com, .exe o .bat puede ser ejecutado.

### 10.3.1.1 Estructura de los ficheros

Al hablar de estructura de los ficheros debemos hacer una distinción entre **estructura lógica u organización del fichero**, es decir, la visión externa que se tiene de éste, y la **estructura física o visión interna del fichero**.

Un fichero es una colección de **registros**, que a su vez, suelen componerse de uno o varios **campos**. Por este motivo, un elemento fundamental en el diseño de un sistema de ficheros es la forma en la que los registros están organizados o estructurados. La organización de ficheros hace referencia a la organización lógica de los registros, es decir, la forma en que se guardan y ordenan los registros de un fichero. Por tanto, la colección de todos los registros que componen el fichero y la percepción que tiene el usuario de la forma en que están organizados constituye la estructura lógica del fichero, que coincide con la visión que tiene el usuario de éste.

La estructura física de un fichero es la colección de bloques físicos que lo componen y la forma en que éstos están almacenados en memoria secundaria.

Hay sistemas operativos que soportan múltiples estructuras lógicas, como por ejemplo VMS, que define tres. Sin embargo, otros sistemas consideran una sola estructura, como por ejemplo LINUX, que considera un fichero como una secuencia de bytes. Ambos enfoques tienen sus ventajas y sus desventajas. En el primer caso, el sistema operativo nos va a prestar más servicios, pero también va a ser mucho más complicado. En el segundo, vamos a tener un sistema más simple y flexible, aunque nos preste menos ayuda en algunos casos.

Existen distintos tipos de organizaciones de ficheros:

- Pila.
- Secuencial.
- Secuencial-indexada.
- Indexada.
- Directa o de dispersión.

Estrechamente relacionada con la organización, está la forma de acceder a la información que hay en los ficheros. Esto es lo que se conoce como **método de acceso**. Algunos sistemas sólo proporcionan un único método, mientras que otros proporcionan varios. Estos métodos pueden ser:

- Acceso secuencial.
- Acceso secuencial-indexado.
- Acceso directo o indexado.

En un sistema multiusuario además de tener en cuenta los aspectos anteriores, hemos de pensar que existe la necesidad de permitir a los usuarios compartir ficheros. Entonces aparecen dos cuestiones, derechos de acceso y acceso simultáneo. La primera hace referencia a las operaciones que pueden hacerse sobre los ficheros y la segunda a la posibilidad de que varios usuarios deseen actualizar el mismo fichero, haciendo necesario incorporar mecanismos que aseguren el acceso exclusivo al fichero.

### 10.3.2 Directorios

Todo sistema de ficheros necesita de una estructura que permita organizar de alguna forma el conjunto de ficheros. Los directorios son los que realizan esta función. Éstos son ficheros que contienen información sobre otros ficheros y se diferencian de ellos en el modo de acceso.

La principal tarea de un directorio es la de asociar los nombres de los ficheros con su ubicación física en el almacenamiento secundario. Para ello, los directorios se estructuran en entradas, una por fichero. Cada entrada incluirá el nombre del fichero y atributos de éste, como su ubicación física.

Sin embargo, la información que contienen los directorios acerca de los ficheros varía de un sistema a otro. En algunos contienen todos los atributos de los ficheros, y en otros simplemente un puntero a una estructura de control del fichero.

Dependiendo de la información que se almacene en cada entrada del directorio, su tamaño será mayor o menor. Los directorios normalmente se almacenan en memoria secundaria debido a que pueden resultar muy grandes. No obstante, se suele cargar una parte en memoria principal para mejorar la velocidad de acceso.

Las operaciones que se pueden realizar sobre un directorio son:

- Ver los atributos de un fichero.
- Establecer los atributos de un fichero.
- Renombrar, crear, borrar, copiar y mover ficheros.
- Crear y deshacer enlaces.

- Listar el contenido del directorio.

Los directorios se pueden organizar de distintas formas, a continuación se estudiarán las más comunes.

#### 10.3.2.1 Directorio con estructura de árbol

Esta estructura permite que un directorio contenga tanto ficheros como otros directorios (figura 10.2). Esto permite a los usuarios crear sus propios directorios y organizar sus ficheros. El sistema de ficheros de MS-DOS, por ejemplo, tiene estructura de árbol. El árbol tiene un **directorío raíz**, a partir del cuál se organizan todos los ficheros y directorios del sistema. Cada fichero en el sistema tiene un camino único, que es su ubicación dentro del árbol. Este camino puede ser **absoluto** o **relativo**, según venga expresado a partir del directorio raíz o a partir del directorio donde se encuentra situado el usuario, denominado **directorío de trabajo**, respectivamente.

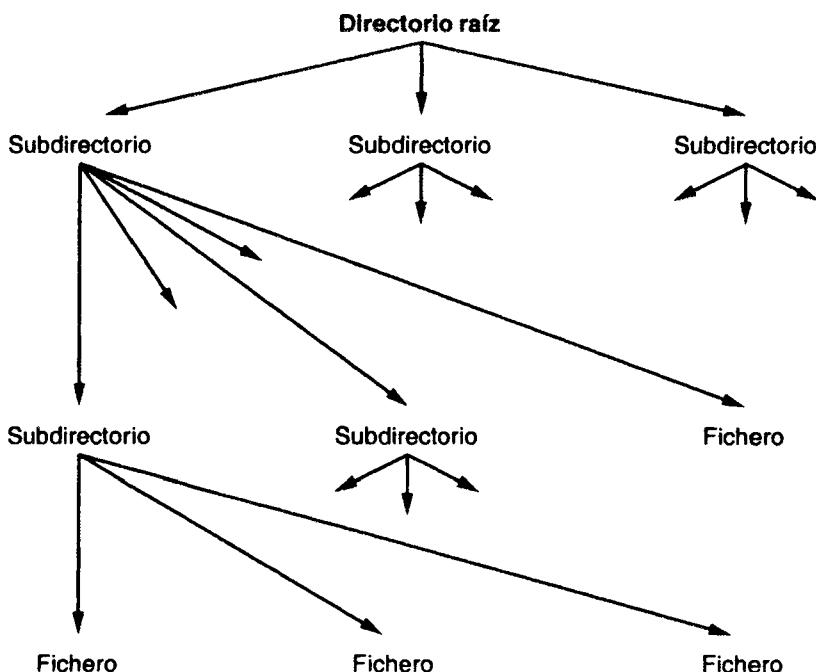


Figura 10.2: Directorio con estructura de árbol

### 10.3.2.2 Directorio con estructura de grafo acíclico

Una estructura de árbol no permite la compartición de ficheros o directorios. Un grafo acíclico permite que un directorio tenga ficheros compartidos, ya que un mismo fichero puede aparecer en dos directorios diferentes, como se muestra en la figura 10.3.

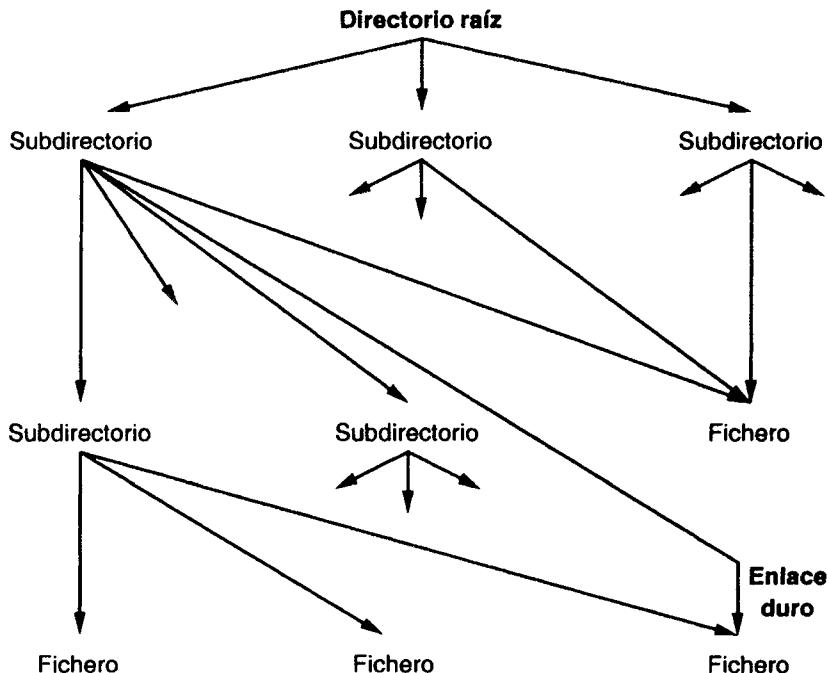


Figura 10.3: Directorio con estructura de grafo acíclico

Los ficheros compartidos pueden ser implementados de varias formas. Una forma común, usada por LINUX, es crear una nueva entrada de directorio llamada **enlace duro**, que duplica la entrada del fichero.

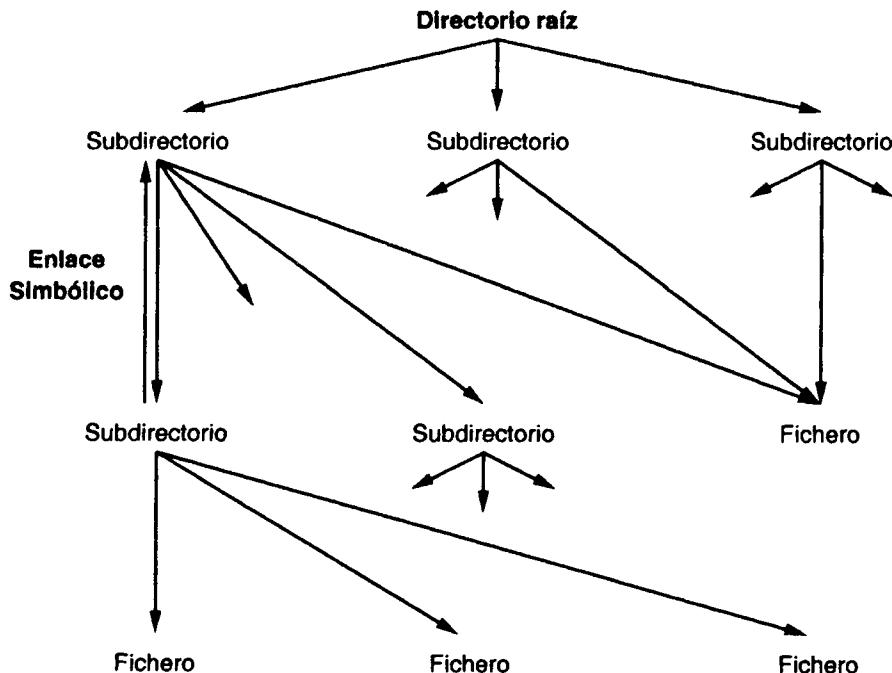
Una estructura de grafo acíclico es más flexible que una estructura de árbol, pero también, más compleja. Pueden aparecer varios problemas debido a que un fichero puede tener múltiples caminos. Así, cuando examinamos el sistema de ficheros completo (para encontrar un fichero, tomar datos estadísticos sobre los mismos, o hacer una copia de seguridad) este problema es significativo, puesto que no se debe recorrer las estructuras compartidas más de una vez.

Otro problema está relacionado con el borrado. ¿Cuándo se debe borrar la estructura del fichero? Sólo cuando se hayan eliminado todas las referencias, lo

que implica llevar un contador de éstas.

### 10.3.2.3 Directorio con estructura de grafo general

Es el caso más general de estructura de directorio. Se da cuando en el grafo acíclico se permite que un subdirectorio esté catalogado en más de un directorio. Esto se puede conseguir en los sistemas LINUX con los llamados **enlaces simbólicos**. Éste no es más que un puntero a otro fichero o subdirectorio. Cuando se hace referencia a un enlace simbólico, se busca en el directorio su entrada que está marcada como un enlace y proporciona el camino para localizar el fichero real. Los problemas que teníamos en el caso de un grafo acíclico se acentúan más.



**Figura 10.4:** Directorio con estructura de grafo general

## 10.4 Diseño del sistema de ficheros

En este apartado vamos a estudiar los sistemas de ficheros desde el punto de vista del diseñador del sistema operativo, cómo manejar el espacio en disco, cómo almacenar los ficheros y cómo hacer que todo funcione de forma eficiente y fiable.

### 10.4.1 Agrupamiento de registros

Los registros son las unidades lógicas de acceso a un fichero, mientras que los bloques son las unidades de E/S del disco. Como normalmente el tamaño del bloque y de los registros no coinciden es necesario realizar el agrupamiento de registros. Antes de estudiar éste haremos algunas consideraciones acerca del tamaño de los bloques.

Hay varios aspectos a considerar. Primero, ¿deberían ser los bloques de longitud variable o fija? En la mayoría de los sistemas los bloques son de longitud fija. Esto simplifica la E/S, la asignación de memoria intermedia en memoria principal, y la organización de los bloques en el almacenamiento secundario.

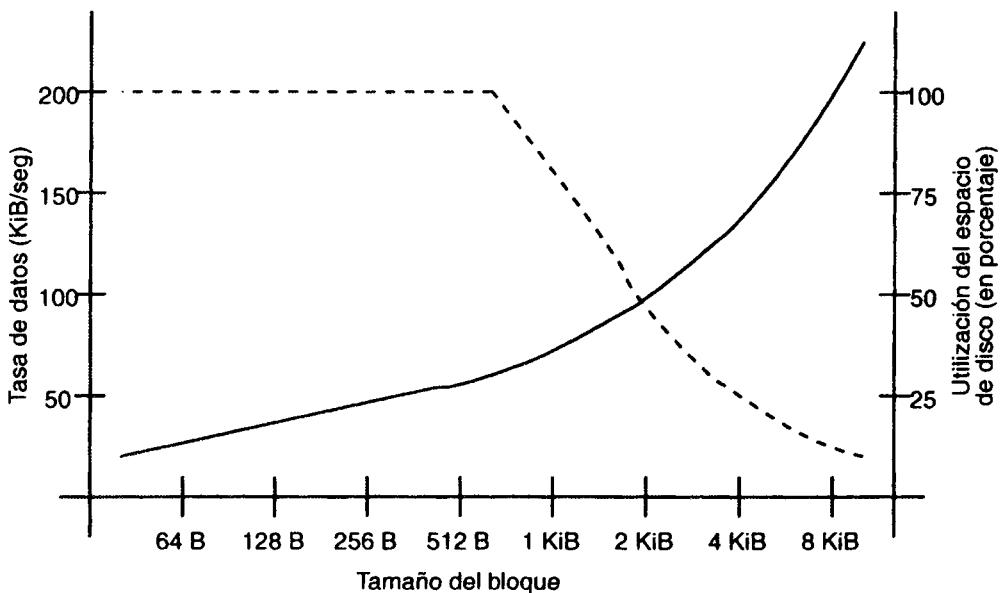


Figura 10.5: Efecto del tamaño de bloque

Otro aspecto a considerar es el tamaño relativo de un bloque con relación al tamaño promedio de los registros. La situación es la siguiente, cuanto más grande sea un bloque, más registros se pasan en una operación de E/S. Si un fichero está siendo procesado o se está realizando una búsqueda de forma secuencial, esto sería una ventaja porque el número de operaciones de E/S se ve reducido al usar bloques más grandes, acelerándose así el proceso. Por otro lado, si se va a acceder de forma aleatoria a los registros, entonces los bloques más grandes dan lugar a una transferencia innecesaria de registros que no se van a usar. Además, hemos de tener en cuenta que el bloque es la unidad mínima de asignación en la memoria.

secundaria, por lo que ficheros pequeños pueden desperdiciar espacio en el bloque. Estos dos factores se ven reflejados en la gráfica de la figura 10.5.

El tamaño del bloque puede colaborar con el hardware de la memoria virtual. En un entorno de memoria virtual paginada es deseable que la unidad básica de transferencia sea la página. De este modo, conviene formar bloques que equivalgan a una página o bien a un conjunto de ellas, para que en cada operación de E/S dispongamos de un conjunto de páginas.

Se suelen utilizar los métodos siguientes de agrupación de registros:

**Agrupamiento fijo** Se usan registros de longitud fija, y en cada bloque se almacena un número determinado y entero de registros, conocido como **factor de bloque**. Puede existir espacio no usado al final de cada bloque, es decir, fragmentación interna. Es el método más común para ficheros secuenciales con registros de longitud fija. El tamaño del registro está limitado al tamaño del bloque.

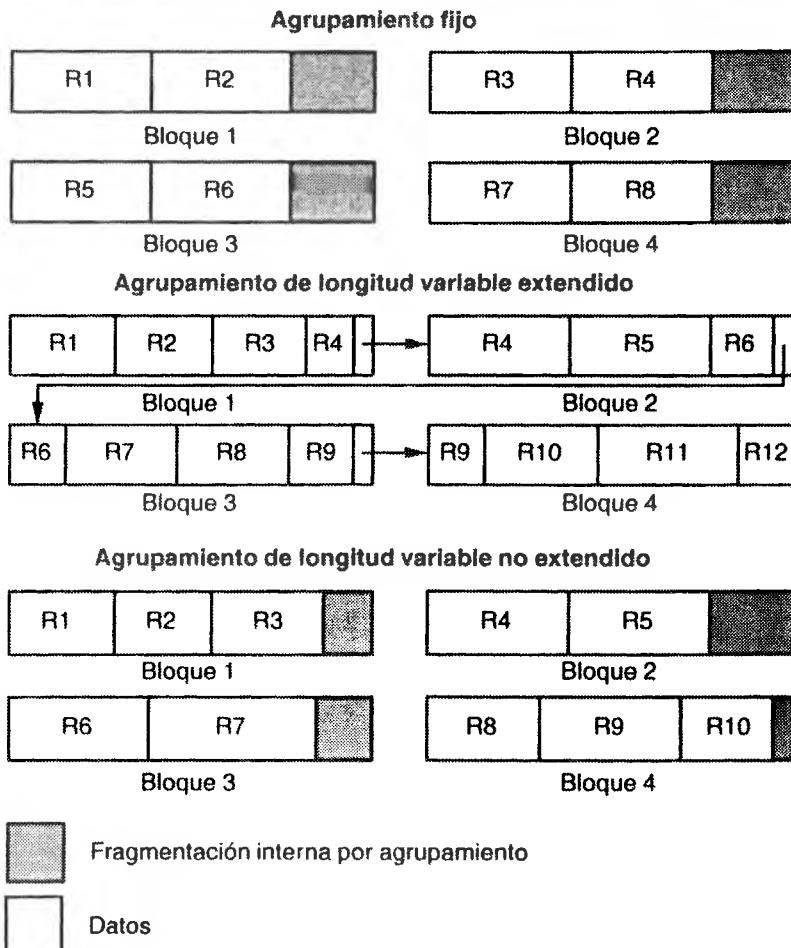
**Agrupamiento de longitud variable extendido** Se usan registros de longitud variable que son empaquetados en bloques sin dejar espacio libre. Así, algunos registros deben expandirse a dos o más bloques; en este caso, un puntero indica donde reside el resto del registro. Constituye un almacenamiento eficaz, pero es de difícil implementación, pues se pueden requerir dos operaciones de E/S para leer un registro, y además se hace difícil la actualización de los ficheros. Se conoce también como **agrupamiento de longitud variable por tramos**.

**Agrupamiento de longitud variable no extendido** También denominado **agrupamiento de longitud variable sin tramos**. Se usan registros de longitud variable, pero no se emplea la expansión. Hay espacio desperdigado en la mayoría de los bloques debido a la incapacidad para usar el resto de un bloque si el registro siguiente es mayor que el espacio sobrante no usado. También se limita el tamaño máximo de un registro al tamaño de un bloque.

La figura 10.6 ilustra estos métodos, considerando que un fichero está almacenado en bloques secuenciales en un disco. El efecto no cambiaría si se utilizara cualquier otro método de asignación del espacio del disco.

#### 10.4.2 Métodos de asignación de espacio a los ficheros

Se suelen utilizar tres métodos, **asignación contigua**, **enlazada** e **indexada**. Cada uno tiene sus ventajas y sus inconvenientes, por lo que algunos sistemas



**Figura 10.6:** Métodos de agrupamiento de registros

operativos proporcionan más de un método, aunque la mayoría sólo proporciona un método de asignación para todos los ficheros.

#### 10.4.2.1 Asignación contigua

En este caso se asigna al fichero un conjunto contiguo de bloques, en el momento de su creación, suficiente para almacenar su contenido actual y hacer frente a su posible crecimiento en un futuro. Por tanto es una estrategia de asignación previa. En este caso, la entrada del directorio necesita almacenar el bloque de comienzo, el tamaño máximo del fichero y su tamaño actual. Esto se puede ver

en la figura 10.7.

Este tipo de asignación permite acceso secuencial y directo a los ficheros. Para un acceso secuencial el sistema de ficheros recuerda la dirección de disco del último bloque referenciado y, cuando es necesario, lee el bloque siguiente. Para el acceso directo al bloque  $x$  de un fichero que comienza en el bloque  $i$ , podemos acceder inmediatamente al bloque  $i + x - 1$ .

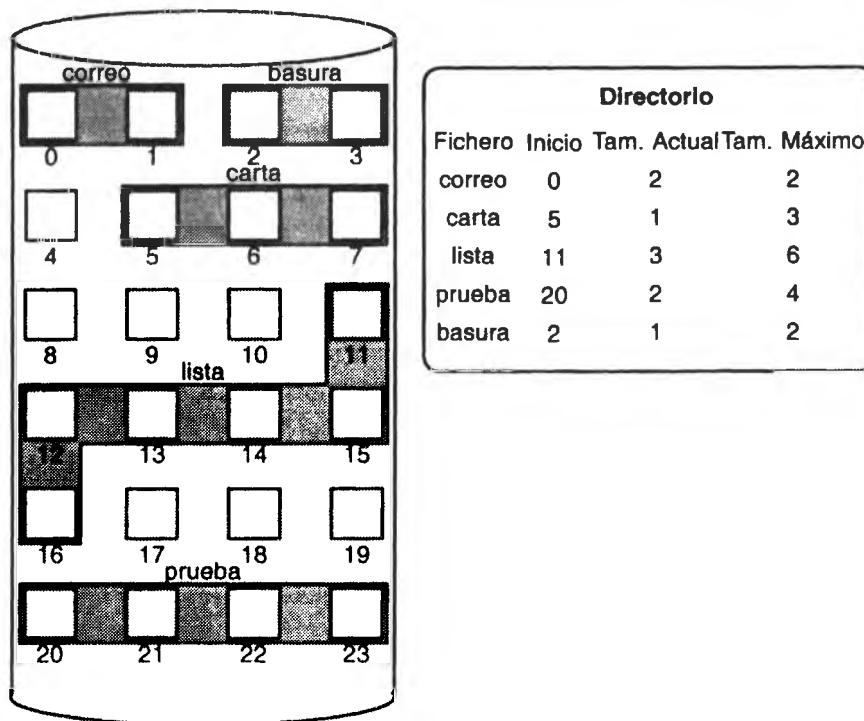


Figura 10.7: Asignación contigua

La asignación contigua presenta algunos problemas:

- Puede aparecer fragmentación externa, haciendo difícil encontrar bloques contiguos libres de suficiente longitud. Para reducirla es necesario la compactación del disco de forma periódica, pero esto es costoso en tiempo, obligando a detener el sistema mientras se realiza.
- Además, es necesario realizar una política de asignación del espacio libre como primer ajuste, mejor ajuste, peor ajuste, etc.
- Es necesario declarar el tamaño máximo de un fichero en el momento de

su creación (preasignación). Esto origina que inicialmente se produzca fragmentación interna, debido a que se solicita el tamaño máximo que tendrá el fichero, pudiendo ocuparlo o no.

- Dificultad en la realización de operaciones de borrado de registros, creación de nuevos registros, etc.

#### 10.4.2.2 Asignación enlazada

Otro método de asignación es la enlazada o encadenada (figura 10.8). En este caso, los bloques se asignan individualmente y cada uno de ellos contiene, además de una zona de datos, un puntero al siguiente bloque de la cadena. Cada entrada del directorio contendrá el primer y el último bloque del fichero. Se realiza una asignación dinámica de acuerdo a las necesidades del fichero y cualquier bloque puede serle asignado. Por tanto, no hay problemas de fragmentación externa.

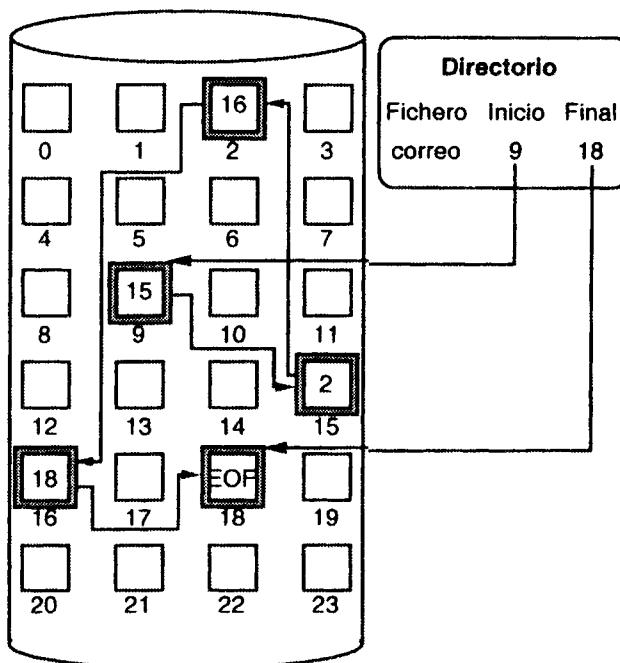


Figura 10.8: Asignación enlazada

Este tipo de organización sólo permite el acceso secuencial a los ficheros, ya que para acceder a un bloque determinado hay que leer los anteriores. Dado que

los bloques pueden estar dispersos por todo el dispositivo, se puede realizar una desfragmentación para agrupar los bloques del fichero y acelerar el acceso.

Un problema más grave es la fiabilidad. Dado que los bloques de un fichero están enlazados mediante punteros, si uno de ellos se pierde, perderíamos el resto de la información. Una solución parcial es la utilización de listas doblemente enlazadas o almacenar el nombre del fichero y el número de bloque relativo dentro de cada bloque; estos métodos requieren aún más sobrecarga para cada fichero.

Los sistemas operativos MS-DOS y OS/2 utilizan una variante del método de asignación enlazada. En ellos, en vez de guardarse la información sobre cuál es el siguiente bloque del fichero en los propios bloques de datos se construye lo que se llama una **tabla de asignación de ficheros** (FAT). En cada partición del disco se reserva un espacio para contener esta tabla. Ésta tiene una entrada por cada bloque del disco, y está indexada por el número de bloque.

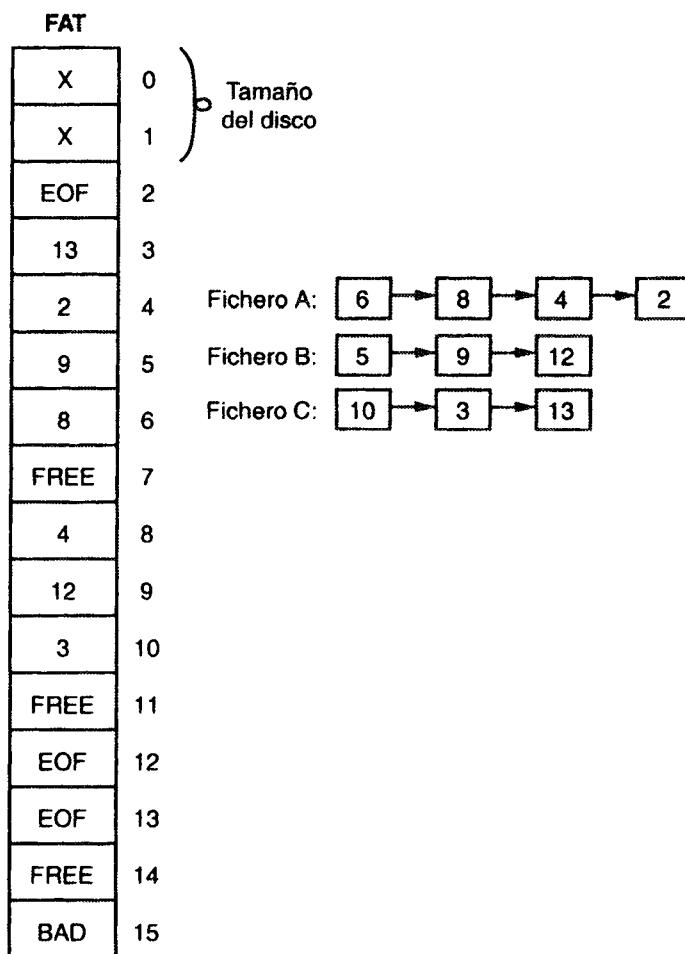
Las dos primeras entradas de la FAT están reservadas para contener el tamaño del disco; el resto describe la utilización de los bloques. El contenido de cada entrada puede ser:

- Bloque disponible (FREE).
- Último bloque del fichero (EOF).
- Siguiente bloque del fichero (n).
- Bloque en mal estado (BAD).

Cada entrada del directorio contiene el número del primer bloque asignado al fichero; éste se utiliza como punto de entrada en la FAT. A partir de éste, cada entrada de la FAT contiene el número del siguiente bloque, hasta llegar al final del fichero. En el caso del fichero A (figura 10.9), éste comienza en el bloque 6. La entrada número 6 de la FAT contiene el número del siguiente bloque asignado al fichero, el 8; esta entrada apunta al siguiente bloque, el 4; la entrada número 4 almacena el número del siguiente bloque, el 2; como éste es el último bloque asignado al fichero, la entrada número 2 de la FAT contiene el código de fin de fichero (EOF).

El problema de la FAT es que la información sobre la ubicación de los ficheros en el disco se combina al azar en la misma tabla. Esto quiere decir que se necesita toda la FAT para abrir un fichero.

La asignación enlazada resuelve los problemas de la fragmentación externa y la declaración del tamaño máximo del fichero que conlleva la asignación contigua. Sin embargo, no soporta de forma eficiente el acceso directo a los ficheros.



**Figura 10.9:** Tabla de asignación de ficheros

#### 10.4.2.3 Asignación indexada

La asignación indexada resuelve los problemas de los esquemas anteriores, es decir, soporta el acceso directo sin sufrir fragmentación externa. Es la forma más utilizada actualmente de asignación de espacio a los ficheros.

Al igual que la asignación encadenada, ésta permite asignar dinámicamente los bloques a un fichero según sus necesidades. La información de qué bloques componen un fichero se mantiene en su **bloque índice**. La entrada  $i$ -ésima del bloque índice apunta al  $i$ -ésimo bloque del fichero. La figura 10.10 muestra este esquema de asignación.

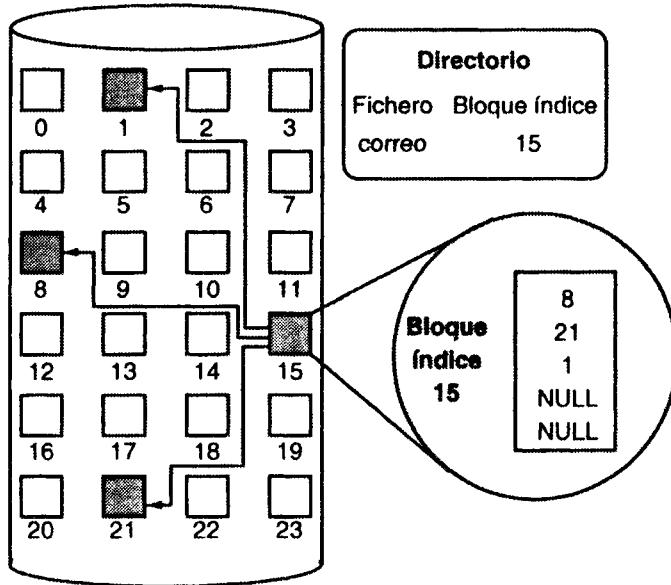


Figura 10.10: Asignación indexada

Este tipo de asignación permite tanto el acceso secuencial como el directo. Para acelerar los accesos, el bloque índice se almacena en memoria principal en el momento en que se abre el fichero, manteniéndose en ella hasta que se cierra. De este modo, al trabajar con un fichero disponemos en memoria principal de cuál es la localización de los distintos bloques de datos que posee, sin necesidad de requerir dos accesos a disco para localizar un bloque de datos.

Normalmente los bloques índices no forman parte del directorio, sino que éste mantiene un puntero al mismo. Con esta estrategia, el fichero tiene tantos bloques de datos como necesite, además cada uno es usado íntegramente para almacenar datos, sin necesidad de almacenar un puntero como en el caso de la asignación enlazada.

Un problema que plantea la asignación indexada es el gasto extra de espacio para mantener los bloques índice. El caso más extremo se da cuando tenemos un fichero que sólo ocupa un bloque de datos; con este método de asignación necesitaremos dos bloques, el de datos y el índice.

Otra cuestión es cómo de grande puede ser el bloque índice. Puesto que cada fichero tiene su propio bloque índice, nos interesaría que éste sea lo más pequeño posible, para no desperdiciar mucho espacio. Sin embargo, si el bloque índice es demasiado pequeño, no será capaz de mantener todos los punteros necesarios en

el caso de que tengamos ficheros grandes. Por tanto, habrá que diseñar algún tipo de mecanismo para tratar este problema. Veamos algunas posibilidades:

**Esquema enlazado** Normalmente un bloque índice es un bloque de disco. Si tenemos un fichero grande que necesita más de un bloque índice para mantener las direcciones de los bloques que lo componen, los bloques índices se enlazan entre sí. Es decir, para ficheros pequeños sólo se utilizará un bloque índice, para ficheros grandes se utilizarán tantos como sean necesarios, todos ellos enlazados entre sí a través de la última entrada de cada bloque índice.

**Índice multinivel** Una variante del método anterior es usar un bloque índice separado para apuntar a los bloques índice. Para acceder a un bloque, el sistema operativo utiliza el índice de primer nivel para encontrar el índice de segundo nivel, y éste para encontrar los bloques de datos. Este esquema se podría continuar con un tercer o cuarto nivel, dependiendo del tamaño máximo de los ficheros que se quiera manejar.

**Esquema combinado** Otra alternativa es utilizar un bloque en el que se mantienen punteros directos a bloques de datos y punteros indirectos. Éstos apuntan a bloques que contienen punteros a bloques de datos.

### 10.4.3 Gestión del espacio libre

El sistema de gestión de ficheros debe llevar el control de qué bloques están libres y cuáles ocupados. Normalmente se utilizan dos técnicas, **mapas de bits** y **listas enlazadas**.

#### 10.4.3.1 Mapa de bits

Frecuentemente, el control del espacio libre se lleva a cabo mediante un **mapa de bits**, donde cada bloque se representa con un bit. Si el bloque está libre, el bit está a 0, y si está asignado, el bit estará a 1. La figura 10.11 ilustra esta técnica. El tamaño del mapa dependerá del número de bloques del disco. Así, para un disco de 1 GiB, con bloques de 4 KiB, serán necesarios 262144 bits. Por tanto, se necesitarían 8 bloques para almacenar el mapa de bits.

La principal ventaja de este método es que es relativamente simple y eficiente para encontrar  $n$  bloques libres consecutivos. Sin embargo, los mapas de bits son ineficientes a menos que se mantenga el mapa completo en memoria principal.

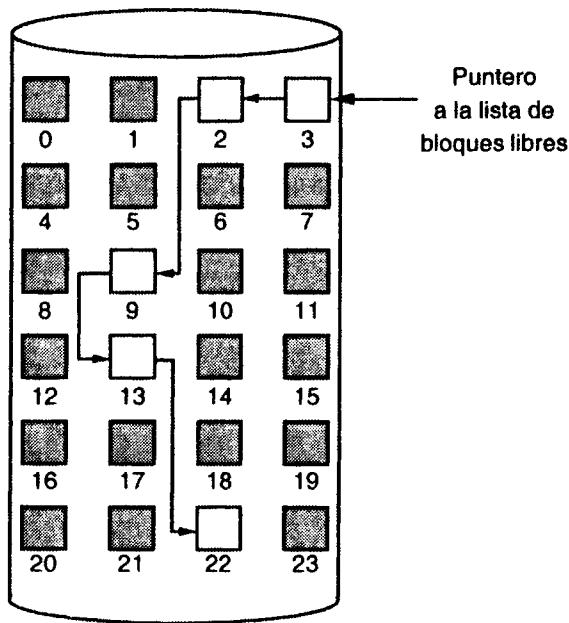
Bloques libres: 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, 27

Mapa de bits: 11000011000000111001111100011111...

**Figura 10.11:** Mapa de bits para la gestión del espacio libre

#### 10.4.3.2 Lista enlazada

Otra posibilidad es enlazar todos los bloques libres del disco usando punteros. Este método no gasta mucho espacio porque no necesita una tabla de asignación del disco, sino simplemente un puntero al primer bloque libre. El resto de los punteros se almacenan en los propios bloques libres (figura 10.12).



**Figura 10.12:** Lista de espacio libre enlazada

Este método no es muy eficiente ya que para recorrer la lista tenemos que leer cada bloque, lo que requiere una operación de E/S. Por este motivo, para optimizar el tiempo de búsqueda de un conjunto de bloques libres, se han realizado diversas implementaciones. Una primera forma es el **agrupamiento**. En este caso se usa un primer bloque que almacena las direcciones de  $n$  bloques libres (tantos como quepan en él), donde la última dirección apunta al siguiente bloque que contiene direcciones de bloques libres, y así sucesivamente. De este modo, se

hallan rápidamente las direcciones de un gran número de bloques libres.

El inconveniente es el espacio que requiere esta técnica. Supongamos que la implementamos en el mismo disco visto en el mapa de bits, es decir, un disco de 1 GiB, con bloques de 4 KiB, donde cada dirección requiere 32 bits. Cada bloque puede contener un total de 1024 direcciones de bloques. Luego para poder contener los 262144 bloques del disco, serán necesarios 257 bloques.

Otra posibilidad consiste en aprovechar los bloques libres contiguos. Es lo que se denomina **recuento**. Esta técnica almacena la dirección de un bloque libre junto con el número de bloques libres contiguos. De este modo, cada entrada requiere más espacio que antes, pero la lista será más corta, dado que el contador generalmente será mayor que uno.

#### 10.4.4 Implementación de directorios

Un directorio es un tipo de fichero especial, que contiene información de la ubicación de los distintos ficheros que se catalogan en él. La implementación de los directorios es muy importante en el sistema de gestión de ficheros, debido a que toda operación que se quiera realizar sobre un fichero requiere acceder previamente al directorio.

El método más sencillo de implementación de un directorio es el de una lista lineal de sus entradas. Así, crear un fichero consiste en añadir una entrada al final de la lista. Borrar un fichero consiste en eliminar su entrada de la lista; ese espacio puede ser posteriormente utilizado por un nuevo fichero, o bien pasar la última entrada a dicha posición para reducir el tamaño del directorio. El problema que plantea esta implementación es la operación de búsqueda de un fichero, ya que al no estar la lista ordenada, tenemos que hacer una búsqueda secuencial en el directorio. El mantenerla ordenada para facilitar la búsqueda nos complicaría los métodos de inserción y borrado de ficheros.

Una alternativa que permite mejorar los tiempos de búsqueda sin perjudicar las operaciones de inserción y borrado, sería la utilización de una tabla de dispersión. El inconveniente de esta implementación es que la tabla de dispersión presenta un tamaño fijo, y la función de dispersión depende de éste.

Con independencia del tipo de implementación que tenga el directorio, los sistemas operativos actuales suelen mantener en memoria principal, mediante la caché de disco, la información de los directorios de uso más reciente.

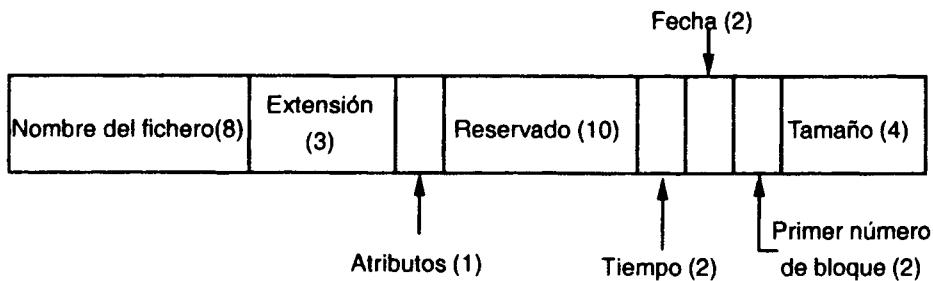
Otro de los aspectos a considerar es la estructura de las distintas entradas. Esta depende del tipo de asignación empleado (apartado 10.4.2) pues determina

la mínima información que debe residir en ellas. Sin embargo, la estructura de éstas varía mucho de un sistema a otro.

En la figura 10.13 se muestra una entrada de un directorio en MS-DOS, indicándose entre paréntesis el número de bytes de cada componente. Cada entrada tiene 32 bytes que se emplean para guardar la siguiente información:

- Nombre del fichero.
- Extensión.
- Atributos.
- Fecha y hora de creación o de última modificación.
- Número del primer bloque del fichero.
- Tamaño.

El número del primer bloque del fichero se emplea como índice en la FAT para conocer el resto de los bloques que forman el fichero. Los directorios en MS-DOS son ficheros y pueden contener un número arbitrario de entradas. La estructura de la entrada de un directorio en LINUX se verá en el apartado 10.7.4.



**Figura 10.13:** Entrada de un directorio en MS-DOS

## 10.5 Fiabilidad del sistema de ficheros

El sistema de ficheros, además de llevar el control del espacio libre y la asignación de bloques a los ficheros, debe proporcionar mecanismos de protección de la información almacenada en ellos. Estas técnicas no evitan la destrucción física de los datos, pero pueden permitir recuperarlos, cuando se origina su pérdida,

y ayudarnos a mantener la consistencia de la información almacenada. En este apartado vamos a analizar algunos de los aspectos implicados en la protección del sistema de ficheros.

### 10.5.1 Copias de seguridad

Es importante hacer copias de seguridad de los discos con frecuencia. El medio empleado para realizar la copia dependerá del volumen de información a guardar, así se podrán emplear discos flexibles, discos duros, CD-ROM, cintas magnéticas, etc.

Las copias de seguridad pueden ser de distinto tipo dependiendo de la información que se copie; se pueden hacer copias de toda la información que contiene el disco, o bien sólo de aquellos ficheros que han sido modificados desde la última copia de seguridad; éstas son las llamadas **copias incrementales**. Para poder realizarlas, el sistema debe guardar para cada fichero la fecha en que se realizó la última copia.

### 10.5.2 Consistencia del sistema de ficheros

Muchos sistemas de ficheros cuando realizan operaciones sobre los ficheros modifican la copia existente en memoria principal y posteriormente actualizan la memoria secundaria. Si antes de realizar esta actualización se produce un fallo en el sistema, la información en memoria secundaria puede quedar en un estado inconsistente ya que no coincide con las modificaciones hechas por el usuario. Este problema se agrava si afecta a las estructuras de control del sistema de ficheros.

Un sistema de ficheros se dice que es **consistente** cuando todas las modificaciones realizadas por el usuario se reflejan en el disco.

El sistema operativo puede proporcionar alguna utilidad que nos permita verificar la consistencia del sistema de ficheros. En el apartado 10.7.8 se estudia la herramienta que incorpora el sistema operativo **LINUX**.

## 10.6 Rendimiento del sistema de ficheros

Al estudiar cómo se puede mejorar el rendimiento de un sistema de ficheros vamos a tener en cuenta dos aspectos fundamentales que influyen en la mejora de éste:

- Reducción del número de accesos al disco, que se consigue mediante la caché

de disco (apartado 9.5.2).

- Reducción del movimiento del brazo del disco, esto lo podemos conseguir por medio de la planificación de las peticiones de acceso al disco (apartado 9.5.3) y agrupando los bloques que forman parte de un fichero.

La forma de realizar la agrupación de bloques va a depender tanto del esquema de asignación como de la forma en que se lleva el control del espacio libre. Si se emplea una asignación contigua, el propio esquema nos proporciona el agrupamiento de los bloques. Las otras estrategias de asignación, enlazada o indexada, presentan problemas de dispersión de bloques, por lo que si el fichero necesita más bloques, hemos de intentar asignárselos cercanos al resto. Por tanto, el objetivo es evitar la desfragmentación de los ficheros por todo el disco. Esto dependerá del mecanismo empleado para gestionar el espacio libre. Si se emplea un mapa de bits, es fácil la elección de un bloque libre que esté lo más cerca posible del anterior. Si se utiliza una lista libre, el agrupamiento es más complejo.

Otro obstáculo para el rendimiento en los sistemas de asignación indexada es que se requiere un acceso extra al disco para leer el bloque índice. La ubicación de los bloques índice con respecto a los bloques de datos va a influir en el rendimiento. Si se sitúan en un extremo del disco, como se muestra en la figura 10.14(a), la distancia media entre el bloque índice y los bloques de datos será la mitad del número de pistas, lo que requiere desplazamientos largos.

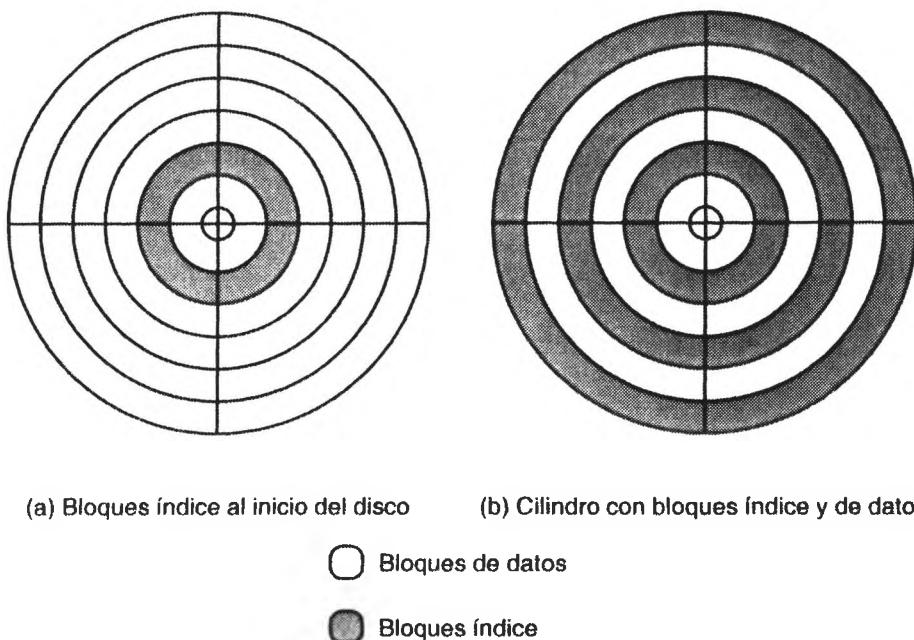
Si el disco se divide en grupos de cilindros, de forma que cada uno disponga de un conjunto de bloques índice y de datos, se puede intentar asignar a un fichero los bloques de datos y el bloque índice en el mismo grupo de cilindros. Si no fuese posible, se elegiría del grupo de cilindros más cercano. Esto es lo que se muestra en la figura 10.14(b).

## 10.7 Sistema de ficheros en LINUX

En 1993 Remy Card y Wayne Davison diseñaron el **Segundo Sistema de Ficheros Extendido** o **ext2**, que se convirtió rápidamente en el sistema de ficheros nativo de **LINUX**, y será el objeto de estudio de este apartado.

### 10.7.1 Estructura lógica

Como la mayoría de los sistemas operativos modernos, **LINUX** organiza su sistema de ficheros como una jerarquía de directorios, que se denomina **árbol de directorios**. Como se muestra en la figura 10.15, existe un directorio especial, llamado



**Figura 10.14:** Optimización del rendimiento en asignación indexada

directorio **raíz**, que está situado en la parte más alta de la jerarquía. Cada directorio del sistema de ficheros **ext2** puede contener tres clases de ficheros:

**Ficheros ordinarios** Contienen datos.

**Ficheros especiales** Proporcionan acceso a los dispositivos de E/S.

**Directarios** Contienen información acerca de conjuntos de ficheros y se utilizan para localizar un fichero a partir de su nombre.

Esa jerarquía con un único directorio raíz puede estar formada por varios sistemas de ficheros residentes en distintas particiones de disco duro u otros dispositivos de bloque. La forma de conseguir esa estructura lógica de un único árbol de directorios es a través de las operaciones de **montaje**<sup>1</sup> de sistemas de ficheros. De este modo:

- Existe un único sistema de ficheros raíz que se monta cuando el sistema arranca.

<sup>1</sup> Montar (*mount*): Integrar en la jerarquía de directorios.

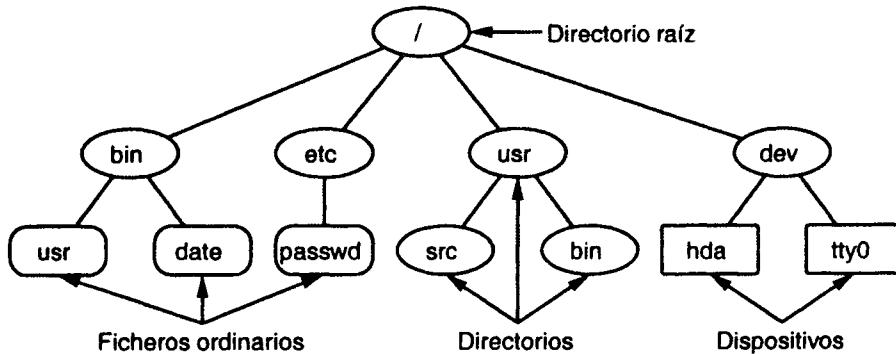


Figura 10.15: Jerarquía de directorios y ficheros

- Los demás sistemas de ficheros se montan en directorios que deben existir previamente en la jerarquía. Los directorios donde se montan los sistemas de ficheros se denominan **puntos de montaje**. Éste es el único modo que tiene el usuario para acceder a los ficheros que hay en una partición de disco duro, CD-ROM, disco flexible, cinta, etc. Si el punto de montaje elegido tiene ya un contenido, éste queda oculto hasta que se desmonte el sistema de ficheros.

Por tanto, el usuario ve un único sistema de ficheros, pero éste está realmente compuesto por varios sistemas de ficheros físicos, cada uno de ellos situado en un dispositivo o partición diferente, pudiendo ser incluso sistemas distintos. La figura 10.16 muestra esta situación. El sistema VFS<sup>2</sup> es el que se encarga de crear ese árbol de directorios, de forma que cuando se accede a un fichero, el sistema VFS transmite la petición del usuario al sistema de ficheros físico correspondiente. Podemos decir, que el sistema VFS se comporta como un sistema de sistemas de ficheros.

Cuando se para el sistema se desmontan todos los sistemas de ficheros. Del mismo modo, si se dispone de un sistema montado en un dispositivo extraíble (disquete o CD-ROM) y queremos extraerlo, es necesario desmontarlo previamente.

<sup>2</sup> Virtual File System.

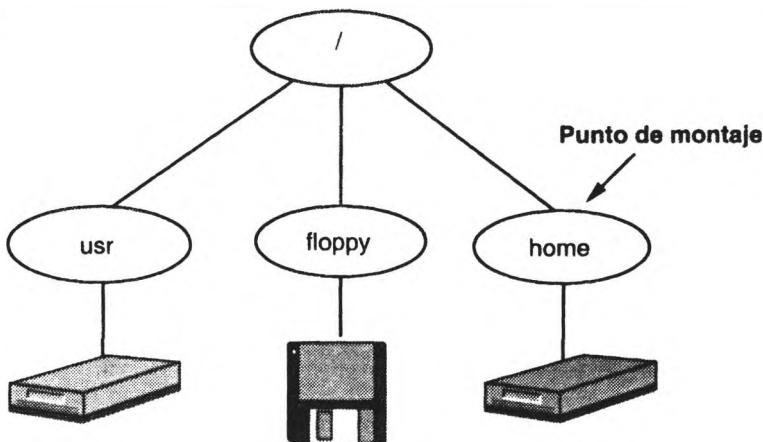


Figura 10.16: Sistema de ficheros constituido por tres dispositivos

### 10.7.2 Estructura física del fichero

El sistema de ficheros **ext2** presenta un esquema de asignación indexada, por lo que los bloques de datos de un fichero pueden estar dispersos por todo el disco. Para llevar el control de cuáles son los bloques que pertenecen a un fichero, se emplea una estructura de control que recibe el nombre de **nodo índice** o **nodo-i**. Éste contiene información de control del fichero, así como sobre los bloques que ocupa éste. Está diseñado como muestra la figura 10.17, ocupando 128 bytes.

Entre la información de control que posee sobre el fichero se encuentra:

- Identificación del propietario del fichero, y el grupo de usuarios al que pertenece.
- Tipo de fichero (fichero ordinario, especial o directorio).
- Bits de protección.
- Fechas de creación, último acceso y última modificación.
- Número de enlaces duros al fichero.
- Tamaño del fichero en bytes y en bloques.

Además, el nodo índice incluye la **tabla de contenidos** utilizada en la localización de los datos de un fichero en el disco. Su diseño permite mantener una estructura del nodo índice pequeña y, a la vez, permitir ficheros grandes.

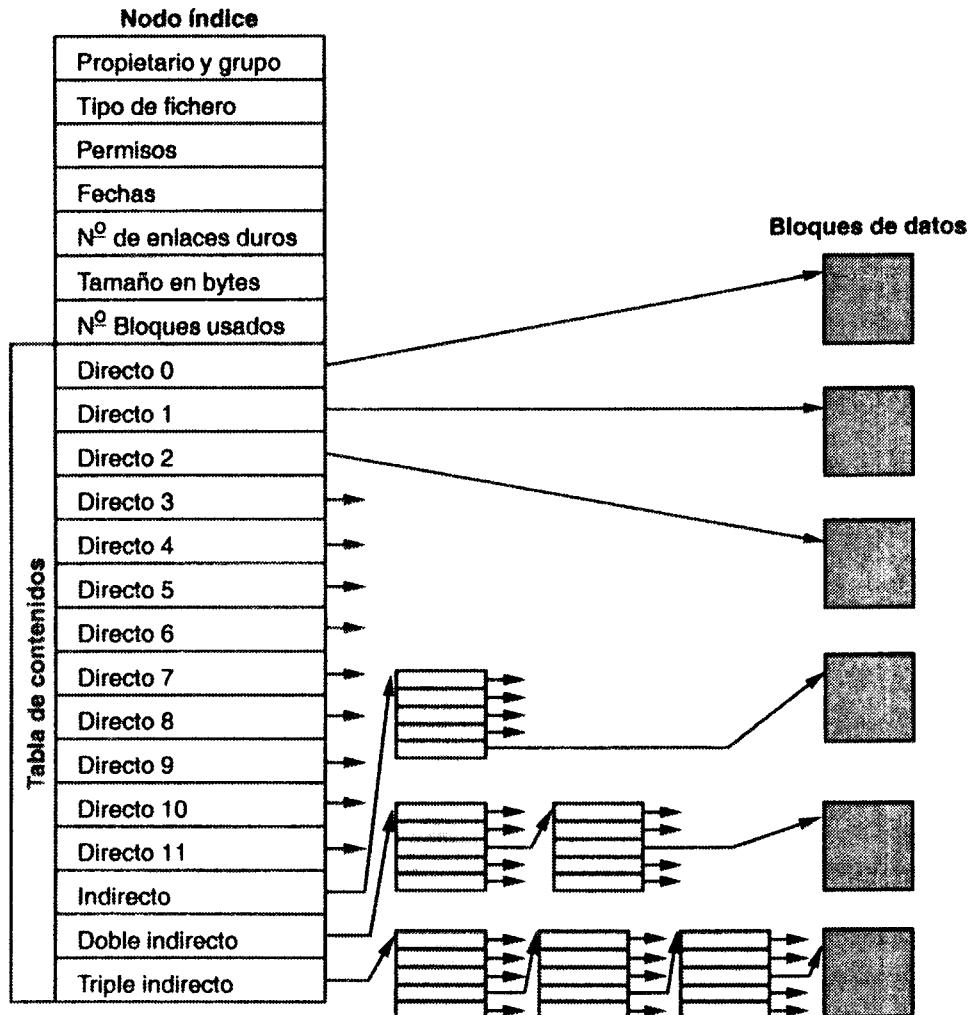


Figura 10.17: Estructura del nodo índice del sistema de ficheros ext2

La tabla de contenidos dispone de 15 punteros, de los cuales 12 de ellos apuntan directamente a bloques de datos del fichero. Así, para ficheros pequeños (no más de 12 bloques) sólo se necesita del nodo índice. Para ficheros de mayor tamaño, las tres últimas entradas de la tabla de contenidos apuntan a bloques de direcciones de bloques de datos. Así, el puntero indirecto simple apunta a un bloque que contiene direcciones de bloques de datos, el puntero indirecto doble apunta a un bloque que contiene direcciones de otros bloques, que a su vez contienen direcciones de bloques de datos y así sucesivamente.

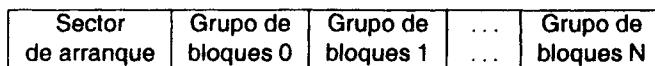
El número de direcciones que hay dentro de un bloque depende del tamaño del bloque. Por ejemplo, si se dispone de bloques de 4 KiB, dado que las direcciones de éstos son de 32 bits, un bloque podría apuntar a otros 1024 bloques. De este modo, el tamaño máximo de un fichero sería:

$$(12 + 1024 + 1024^2 + 1024^3) \times 4 \text{ KiB} \approx 4 \text{ TiB}$$

Aunque es posible direccionar más de 4 TiB, dado que el nodo índice almacena la longitud del fichero en bytes, éste se limita a 4 TiB.

### 10.7.3 Estructura física de un sistema de ficheros

El sistema de ficheros **ext2** divide la partición o dispositivo de bloques donde reside en **grupos de bloques** o **cilindros**, como se muestra en la figura 10.18.



**Figura 10.18:** Estructura física del sistema de ficheros **ext2**

El **sector de arranque** ocupa el principio de la partición o dispositivo de bloques y puede contener el código de arranque (*bootstrap*) que es leído por la máquina para iniciar el sistema operativo.

La estructura de un grupo de bloques se representa en la figura 10.19. En ella se observa que cada uno contiene:

**Superbloque** Estructura de control del sistema de ficheros.

**Descriptores de grupos** Estructuras de control de los grupos de bloques.

**Mapas de bits** Permiten una gestión rápida tanto de los bloques de datos como de los nodos-i disponibles en el sistema de ficheros.

**Tabla de nodos-i** En el sistema de ficheros **ext2** cada fichero se representa por un nodo-i. Esta tabla contiene los nodos-i que se almacenan en el grupo de bloques.

**Bloques de datos** Contienen los datos de los ficheros.

Superbloque	Descriptores de grupos	Mapa de bits de bloques	Mapa de bits de nodos-i	Tabla de nodos-i	Bloques de datos
-------------	------------------------	-------------------------	-------------------------	------------------	------------------

Figura 10.19: Estructura de un grupo de bloques

Podemos comprobar que aunque cada grupo presenta la misma estructura, sólo los grupos de bloques impares son los que contienen una copia redundante de las estructuras de control del sistema de ficheros (superbloque y descriptores de grupos). De todas ellas, el núcleo sólo utiliza la que se encuentra en el grupo 1, las otras sólo se emplean como copias de seguridad, permitiendo aumentar la fiabilidad.

Además, el sistema de ficheros se encuentra repartido entre los distintos grupos de bloques, de forma que cada uno contiene una parte de los nodos-i y bloques de datos del total disponible.

#### 10.7.3.1 El superbloque

El sistema de ficheros se representa globalmente mediante el superbloque. Esta estructura contiene toda la información necesaria para que el núcleo pueda llevar el control del sistema de ficheros. La información contenida en el superbloque se puede agrupar en tres tipos:

- Parámetros fijos que se determinan durante la creación del sistema de ficheros y no pueden variar.
- Parámetros ajustables que se pueden modificar una vez creado el sistema de ficheros.
- Información del estado actual del sistema de ficheros.

La tabla 10.2 muestra los distintos tipos de información que se puede encontrar en el superbloque.

Con respecto a los parámetros fijos, el número mágico se emplea para identificar el tipo de sistema de ficheros en las operaciones de montaje. En el caso del sistema **ext2** es el **0xEF53**.

El tamaño del bloque puede ser 1, 2 o 4 KiB. La elección de un valor u otro dependerá del tamaño del sistema de ficheros. Este parámetro es el valor clave para la determinación de la estructura física del sistema de ficheros. La tabla 10.3

**Parámetros fijos**

- Número mágico.
- Tamaño del bloque.
- Número de nodos-i y de bloques de datos en el sistema de ficheros.
- Número de nodos-i y de bloques de datos por grupo.
- Primer nodo-i.

**Parámetros ajustables**

- Número de bloques reservados para el administrador.
- Identificación del usuario y grupo de usuarios que pueden hacer uso de los bloques reservados al administrador.
- Número de veces que se tiene que montar para realizar una comprobación de la consistencia del sistema de ficheros.
- Número de segundos que puede transcurrir sin realizarse una comprobación de la consistencia del sistema de ficheros.
- Comportamiento del núcleo cuando se encuentra un error en el sistema de ficheros.

**Estado actual del sistema de ficheros**

- Número de bloques libres.
- Número de nodos-i libres.
- Fecha del último cambio efectuado en el sistema de ficheros.
- Fecha en la que se montó por última vez.
- Número de veces que ha sido montado.
- Fecha de la última comprobación de consistencia del sistema.
- Tipo de error detectado en el sistema de ficheros.

---

**Tabla 10.2:** Información del superbloque

muestra la relación existente entre el tamaño del bloque, el número de bloques por grupo y por nodo índice.

Tamaño del bloque	Nº bloques/grupo	Nº bloques/nodo-í
1 KiB	8192	4
2 KiB	16384	3
4 KiB	32768	2

**Tabla 10.3:** Influencia del tamaño del bloque en la estructura física del sistema `ext2`

Un conjunto de información importante es la que emplea la orden `e2fsck`, encargada de comprobar la consistencia del sistema de ficheros. Así, cada vez que el sistema se monta se incrementa una variable, de forma que cuando llegue a un número máximo se realiza una comprobación del sistema. Este límite de operaciones de montaje se establece por omisión a 20. Con objeto de evitar que éstas se hagan muy tardías, en el superbloque se almacena también el número de segundos que debe transcurrir como máximo entre dos comprobaciones consecutivas.

Un aspecto importante en el sistema `ext2` es la posibilidad de reservar una serie de bloques del disco para el administrador del sistema. Por omisión se reserva el 5% de los bloques totales del sistema de ficheros. Esto permite que el sistema no se llene al 100%, puesto que esto podría provocar problemas en su funcionamiento. Además, es posible especificar qué usuario o grupo de usuarios pueden hacer uso de esos bloques.

#### 10.7.3.2 Descriptores de grupo

Cada grupo de bloques se representa en el sistema por el **descriptor de grupo**. Todos los descriptores de grupo de un sistema de ficheros forman una **tabla de descriptores**. Ésta se almacena, al igual que el superbloque, en los grupos de bloques impares por motivos de fiabilidad, aunque sólo se emplea la copia del grupo 1. La información que se encuentra en un descriptor de grupo se puede agrupar en dos tipos:

- Parámetros fijos que se determinan durante la creación del sistema de ficheros y no pueden variar.
- Estadísticas de uso del grupo de bloques.

La tabla 10.4 muestra la información que se puede encontrar en un descriptor. Las estadísticas sobre el uso de los recursos de un grupo de bloques son empleadas

**Parámetros fijos**

- Número de bloques ocupados por el mapa de bits de bloques.
- Número de bloques del mapa de bits de nodos-i.
- El número del bloque donde comienza la tabla de nodos-i para el grupo de bloques.

**Estadísticas de uso**

- Número de bloques y nodos-i libres en el grupo de bloques.
- Número de directorios existentes en el grupo de bloques.

---

**Tabla 10.4:** Información del descriptor de grupo

por el núcleo para distribuir de forma homogénea los ficheros y directorios entre los distintos grupos de bloques.

### 10.7.3.3 Mapas de bits

La gestión de los recursos disponibles del sistema de ficheros, nodos índice y bloques de datos, se realiza mediante mapas de bits. Cada grupo de bloques posee un mapa de bits para los nodos índice y otro para los bloques de datos que se encuentran en él.

El tamaño del mapa de bits viene especificado en el descriptor del grupo. Normalmente se suele emplear un bloque, puesto que si disponemos de bloques de 1 KiB, el mapa de bits puede gestionar hasta  $1024 \times 8 = 8192$  bloques o nodos índice en el grupo de bloques.

### 10.7.4 Directorios

Es un fichero cuyos datos son una secuencia de entradas de longitud variable, que se maneja como una lista enlazada. La estructura de cada entrada se muestra en la figura 10.20.

Número de nodo-i	Longitud de la entrada	Longitud del identificador	Identificador de fichero
------------------	------------------------	----------------------------	--------------------------

Figura 10.20: Entrada de un directorio del sistema de ficheros **ext2**

El identificador de fichero le da un nombre a éste dentro de un directorio. El número máximo de caracteres que puede tener un identificador de fichero es de 255.

Con objeto de facilitar la gestión de la lista de entradas, el tamaño de éstas debe ser múltiplo de 4 bytes, rellenándose de «ceros» los espacios libres. Por este motivo, cada entrada almacena tanto la longitud del identificador como la de la propia entrada.

Las entradas de un directorio se almacenan igual que los datos de cualquier fichero ordinario, usando la estructura de nodo-i y los niveles de bloques directos e indirectos.

Las dos primeras entradas de un directorio son siempre el directorio actual y el padre, denotados por . y .., respectivamente. Estas entradas nunca se pueden eliminar de un directorio.

### 10.7.5 Enlaces

Una entrada de un directorio es un enlace a un fichero, dado que cada entrada apunta a un nodo-i. En el sistema de ficheros **ext2** puede haber múltiples enlaces a un mismo fichero, es decir, podemos tener un fichero con varios nombres. Todos los enlaces a un mismo fichero tienen el mismo número-i. Los enlaces pueden residir en directorios diferentes (figura 10.21(a)). Como sólo hay una copia del fichero, cualquier cambio que se haga en éste a través de cualquiera de sus enlaces es visible para todos los demás.

Estos enlaces se suelen denominar **enlaces duros** y presentan dos restricciones; no se pueden crear enlaces duros entre directorios, ni entre dos ficheros que estén en sistemas de ficheros diferentes. La razón para esta última restricción es que cada sistema de ficheros tiene su propio conjunto de nodos-i, por tanto, los números-i sólo son únicos dentro de un mismo sistema de ficheros.

El sistema de ficheros **ext2** proporciona otra forma de enlace, el simbólico. Cuando se crea un **enlace simbólico** a un fichero se está creando un nuevo fichero, cuyo contenido es el camino del fichero enlazado. Así, cuando hacemos referencia al enlace simbólico el sistema averigua el camino del fichero al que

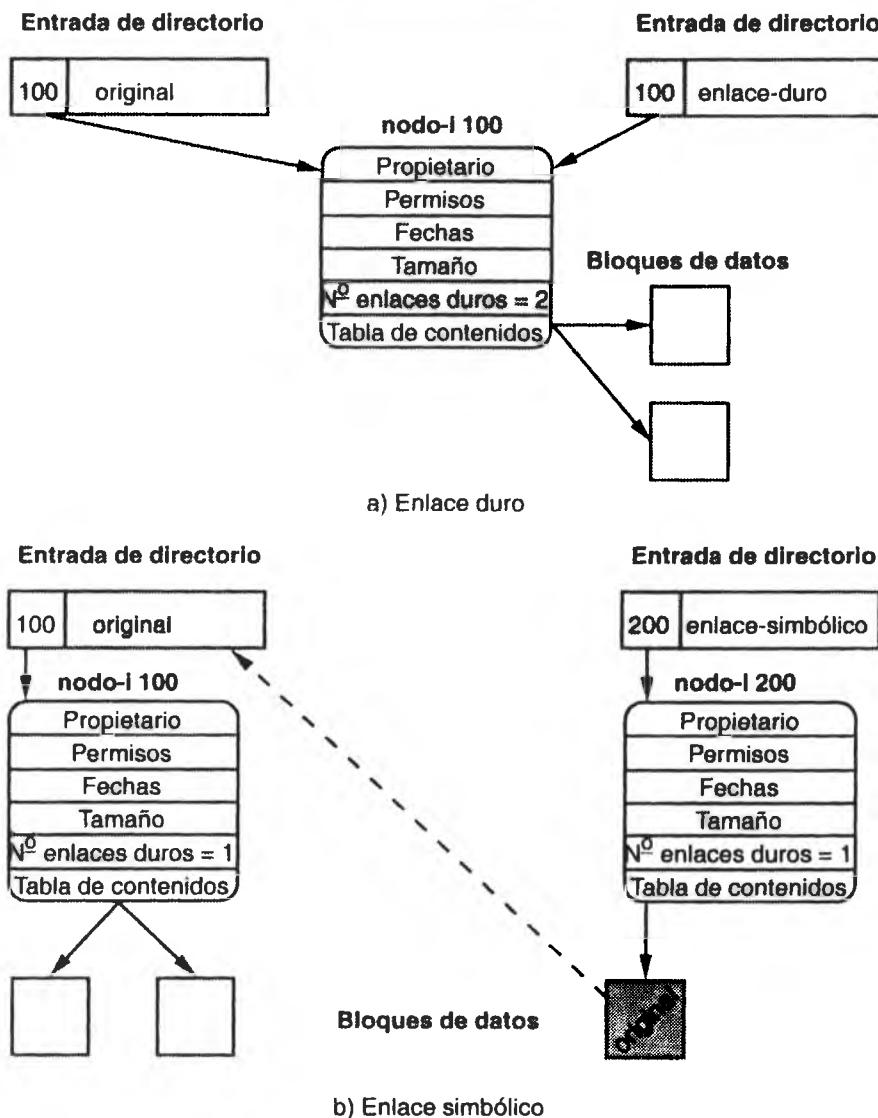


Figura 10.21: Enlaces duros y simbólicos en el sistema de ficheros ext2

realmente vamos a acceder (figura 10.21(b)).

El sistema de ficheros ext2 proporciona dos tipos de enlaces simbólicos: lento y rápido. El enlace lento crea una entrada de directorio, y almacena el camino en un bloque de datos, apuntado por el nodo-i.

El enlace rápido consiste en que el nodo-i en lugar de tener una tabla de

contenidos almacena el camino del enlace en esa tabla. El nombre del fichero enlazado ha de tener una longitud menor de 60 caracteres, que es el espacio disponible en la tabla de contenidos (15 entradas de 4 bytes cada una).

Los enlaces simbólicos no presentan las restricciones de los enlaces duros a la hora de crearlos. Se pueden establecer entre ficheros que estén en sistemas de ficheros diferentes, y también entre directorios, lo cual permite a los administradores de sistemas mover parte de la jerarquía de directorios de un sitio a otro de forma transparente para los usuarios.

El inconveniente de los enlaces simbólicos es que consumen más espacio que los duros, puesto que necesitan un nodo-i y pueden necesitar un bloque de datos.

#### 10.7.6 Búsqueda de un fichero

Un nombre de fichero en **ext2** es una sucesión de identificadores de ficheros separados por el carácter **/**. Cuando se le proporciona un nombre de fichero, el sistema debe localizar a partir de éste su nodo-i para poder acceder a sus bloques de datos. Para ello debe realizar una búsqueda a lo largo de todos los directorios que componen el nombre del fichero.

Supongamos que le damos el nombre absoluto **/home/linux/ext2**. El primer nodo-i que necesita es el del directorio raíz, el cuál se encuentra en el superbloque. A partir de éste accede a sus bloques de datos donde se encuentran almacenadas sus entradas.

A continuación busca la entrada **home** en el directorio raíz con el fin de hallar su nodo-i y localizar las entradas del directorio para poder buscar el siguiente componente del nombre, **linux**. Cuando encuentra la entrada **linux**, ésta tiene el nodo-i del directorio **/home/linux**. A partir de este nodo índice se puede hallar el contenido del directorio y buscar **ext2**. Con su nodo-i podemos acceder a los bloques del fichero deseado. Este proceso de búsqueda se ilustra en la figura 10.22.

Los nombres de ruta relativos se buscan de la misma forma que los absolutos, sólo que comenzando desde el directorio de trabajo y no desde el directorio raíz. Todo directorio tiene entradas para los ficheros **.** y **..** que se colocan ahí cuando se crea el directorio. La entrada **.** tiene el número de nodo-i del directorio actual y la entrada **..** tiene el número de nodo-i del directorio padre. Por tanto, un procedimiento que busca a **../fps/prog.c** simplemente busca a **..** en el directorio de trabajo, halla el número de nodo-i del directorio padre y rastrea ese directorio hasta encontrar **fps**.

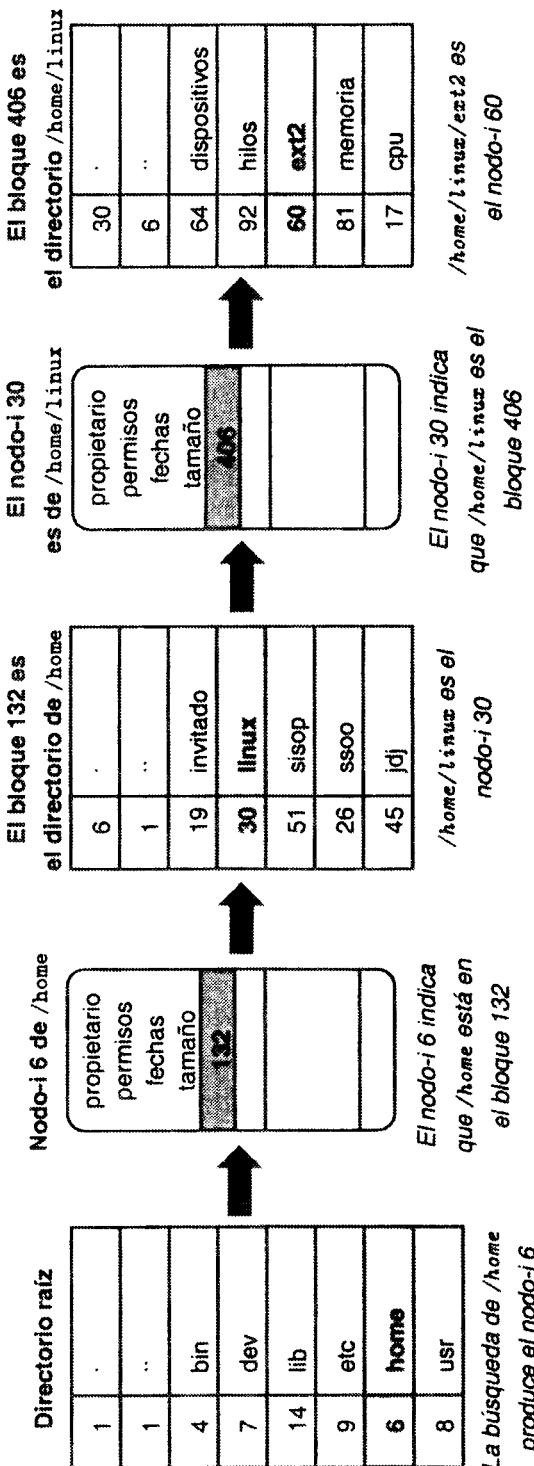


Figura 10.22: Las etapas en la búsqueda de /home/linux/ext2

### 10.7.7 Optimización del sistema de ficheros

El sistema de ficheros **ext2** utiliza un esquema de asignación indexada, lo que puede producir que los bloques de un fichero se encuentren muy dispersos en el disco, con el consiguiente incremento del tiempo de acceso debido al mayor desplazamiento de la cabeza lectora.

Con objeto de reducir los tiempos de acceso y, por consiguiente, mejorar el rendimiento, se realiza una optimización que consiste en intentar almacenar en el mismo grupo de bloques el nodo índice y los bloques de datos de un mismo fichero. Así, si se van a asignar los primeros bloques a un fichero se inicia la búsqueda dentro del mismo grupo donde se encuentra su nodo índice, y si se va a ampliar el tamaño del fichero, se inicia la búsqueda a partir del último bloque asignado al fichero. De este modo, cuando se intenta acceder al fichero el movimiento de la cabeza lectora es el menor posible.

El sistema **ext2** también realiza una preasignación de bloques. Para ello aprovecha la estructura del mapa de bits para la gestión de bloques libres. Dado que un byte del mapa lleva el control de 8 bloques, cuando se busca un bloque libre para un fichero se intenta asignar 8 bloques libres consecutivos, de forma que en el mapa de bits de bloques se representa con un byte de 8 bits libres.

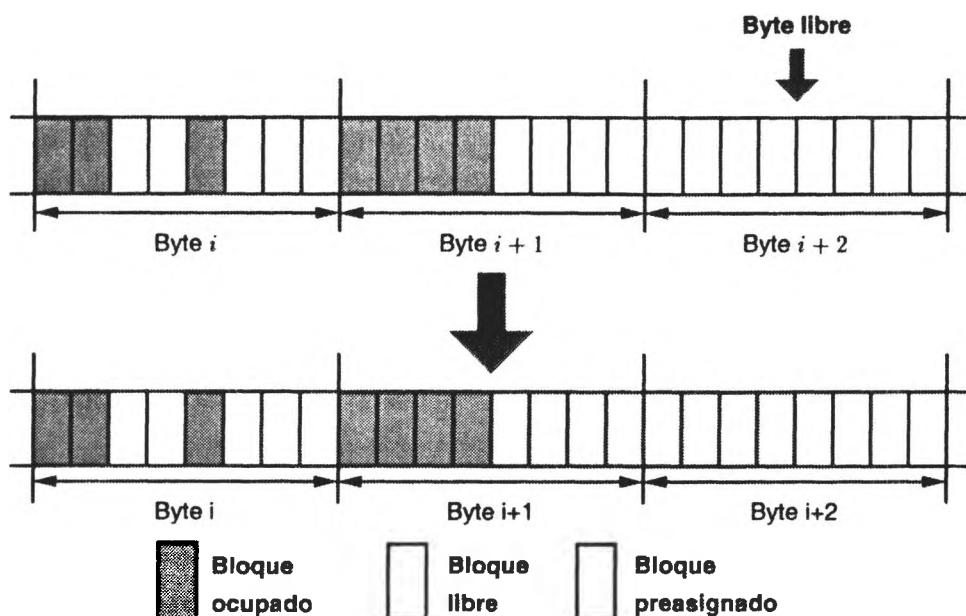


Figura 10.23: Preasignación de bloques

La búsqueda de bloques libres se realiza en dos fases. Primero se busca un byte libre completo en el mapa de bits. Si se encuentra, se realiza una búsqueda hacia atrás para no dejar espacio libre entre el byte y el último bloque asignado. Esta búsqueda en bytes permite que el fichero disponga de bloques adyacentes acelerando el acceso secuencial al fichero. Si no se encuentra un byte libre se buscan 8 bits libres en el mapa de bits para asignárselos al fichero.

La figura 10.23 muestra el resultado de una búsqueda de un bloque libre. De los tres bytes del mapa de bits que se ilustran, el byte  $i + 2$  es el que se encuentra libre. Una vez que lo localiza, retrocede hacia atrás aprovechando el espacio libre anterior.

Esta preasignación es simplemente una reserva de bloques, puesto que cuando el fichero se cierra los bloques preasignados se devuelven al mapa de bits como libres. De este modo, cualquier fichero si necesita más bloques de datos puede ocuparlos.

#### 10.7.8 Consistencia del sistema de ficheros

Para verificar la consistencia, el sistema **ext2** proporciona la orden **e2fsck** cuyo funcionamiento se describe a continuación.

Permite verificar tanto los bloques como los nodos-i. Para verificar la consistencia de un bloque tenemos que ver, en primer lugar, en qué estado se puede encontrar. Un bloque puede estar libre o bien pertenecer a algún fichero. En el caso de no encontrarse en ninguna de esas situaciones excluyentes, estamos en un sistema inconsistente.

Para probar la consistencia del sistema se crea una tabla con dos contadores por bloque, valiendo ambos inicialmente cero. El primer contador lleva la cuenta de las veces que un bloque está presente en un fichero, es decir, los que están ocupados; el segundo registra la frecuencia con la cual está presente en el mapa de bits, es decir, los que están libres.

Después el programa lee todos los nodos-i. Comenzando desde un nodo-i, es posible construir una lista de todos los números de bloque que se utilizan en el fichero correspondiente. Conforme se lee cada número de bloque, se incrementa el primer contador. Después el programa examina el mapa de bits, con objeto de encontrar todos los bloques que no están en uso, incrementando el segundo contador.

Una vez finalizado, se pueden presentar las siguientes situaciones:

- Cada bloque tendrá un 1 en un único contador, como se ilustra en la figu-

ra 10.24(a). En este caso el sistema de ficheros es consistente.

- Sin embargo, si ha ocurrido un fallo, la tabla podría aparecer como en la figura 10.24(b), donde el bloque 5 no aparece en ninguna de ellas. Estos bloques que faltan no hacen un daño real, pero desperdician espacio y, por lo tanto, reducen la capacidad del disco. La solución a este problema es fácil, simplemente el programa `e2fsck` lo marca como libre.
- El error más grave que puede aparecer es que un bloque de datos pertenezca a dos o más ficheros, como se muestra en la figura 10.24(c) con el bloque 7. Si alguno de estos ficheros se borra, el bloque 7 se liberará, lo cual nos llevará a una situación en la que el bloque está libre y en uso al mismo tiempo. La acción que debe llevar a cabo el programa `e2fsck` consiste en copiar el contenido del bloque 7 a un bloque libre y asignar un bloque a cada fichero. Los propietarios de los ficheros deberán ser informados del error para que inspeccionen el daño causado.

Bloque	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Uso	1	1	0	1	0	1	1	1	1	0	0	1	1	1	0	0
Libre	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

(a) Estado consistente

Bloque	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Uso	1	1	0	1	0	0	1	1	1	0	0	1	1	1	0	0
Libre	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

(b) Falta un bloque

Bloque	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Uso	1	1	0	1	0	1	1	2	1	0	0	1	1	1	0	0
Libre	0	0	1	0	1	0	0	0	0	1	1	0	0	0	1	1

(c) Bloque de datos duplicado

Figura 10.24: Estados de consistencia de un sistema de ficheros ext2

Además de revisar los bloques, se debe verificar el sistema de directorios. En este caso, también se utilizan dos contadores, pero por nodo-i, en vez de por bloques. El programa `e2fsck` realiza un recorrido recursivo de todos los directorios del sistema de ficheros, comenzando en el directorio raíz. Cada vez que encuentra un fichero en un directorio incrementa el primer contador y almacena en el segundo el número de enlaces duros que aparece en su nodo-i.

Cuando termina compara los dos contadores, que si el sistema es consistente deben coincidir. Pueden aparecer dos tipos de errores, el contador de enlaces en el nodo-i puede ser mayor o menor que el número de entradas en los directorios.

Si el número de enlaces es mayor, puede provocar un desperdicio de espacio ya que si se borran todos los enlaces al fichero, tanto el nodo-i como los bloques de datos seguirían marcados como ocupados. Si el número de enlaces es menor el error es más grave, puesto que podría desaparecer el fichero sin borrar todos sus enlaces. La solución en ambos casos consiste en almacenar en el número de enlaces del nodo-i el valor del contador que hemos obtenido.

Estas dos operaciones, la verificación de bloques y la de directorios, a menudo se integran por razones de eficiencia (es decir, sólo se requiere un paso por los nodos-i).

### 10.7.9 Llamadas al sistema

La tabla 10.5 muestra las distintas llamadas al sistema relacionadas con el sistema de ficheros en LINUX.

Llamada	Descripción
creat	Crea un nuevo fichero.
open	Abre fichero.
read	Leer de un fichero.
write	Escribir en un fichero.
lseek	Cambiar la posición.
close	Cerrar un fichero.
mknod	Crea un fichero especial.
chdir	Cambio de directorio.
chroot	Cambiar de directorio raíz.
chown, fchown	Cambiar el dueño.
chmod, fchmod	Cambiar los permisos.
mount	Montar un sistema de ficheros.
umount	Desmontar un sistema de ficheros.
stat, fstat, lstat	Obtener información del nodo-i.
link	Crea un enlace duro.
unlink	Elimina un enlace.

Tabla 10.5: Llamadas al sistema relacionadas con el sistema de ficheros

## 10.8 Resumen

Los sistemas operativos proporcionan una abstracción muy útil para tratar con la información que reside en los dispositivos de almacenamiento, los ficheros. Un fichero es una colección de datos con un nombre que suele residir en un dispositivo de almacenamiento secundario como un disco o una cinta. La parte del

sistema operativo que los trata se denomina sistema de ficheros, y proporciona a los usuarios y a las aplicaciones servicios relacionados con el uso de éstos.

En los sistemas de ficheros podemos encontrar dos elementos fundamentales, los ficheros y los directorios. Estos últimos son también ficheros que contienen información sobre otros catalogados en ellos.

Se pueden distinguir dos aspectos fundamentales de los sistemas de ficheros, la forma en que éstos aparecen ante los usuarios o interfaz del sistema de ficheros, y los algoritmos y estructuras de datos que deben crearse para establecer la correspondencia entre el sistema de ficheros que ve el usuario y el real.

Para los usuarios los puntos de interés de un sistema de ficheros son la forma de nombrarlos, las operaciones que pueden hacerse sobre ellos, cómo se protegen, su estructura lógica y la forma de acceso. Con respecto a los directorios les interesa la información que almacenan, las operaciones que pueden realizarse y la forma en que pueden organizarse los ficheros que se catalogan en ellos.

Otros aspectos con los que los usuarios no tratan directamente, pero que son de gran importancia para el rendimiento del sistema son: el método de asignación de espacio a los ficheros, la forma de llevar el control del espacio libre y el método empleado para implementar los directorios.

El sistema de ficheros debe proporcionar mecanismos para asegurar la consistencia de los datos almacenados en él, en este capítulo se estudia el proporcionado por el sistema LINUX.

## 10.9 Ejercicios

1. ¿Por qué es más rápido el tratamiento secuencial de los registros lógicos de un fichero con una asignación encadenada que en una indexada?
2. ¿Qué tipo de fragmentación existe en una asignación contigua y en una asignación encadenada?
3. Un sistema de ficheros **ext2** en un disquete de 1,44 MiB tiene 926 bloques de 1 KiB libres, y 344 nodos-i libres. ¿Cuántos ficheros podrá crear como máximo? ¿Cuál es el tamaño máximo de un fichero en bloques? Razone la respuesta.
4. La gestión del espacio libre se puede llevar mediante un mapa de bits y una lista de bloques libres. Suponga un disco con  $B$  bloques,  $L$  de los cuáles están libres. La lista de bloques se va a implementar como una estructura

independiente en memoria, donde cada elemento contiene el puntero al bloque libre y el puntero al siguiente elemento de la lista. Todas las direcciones requieren  $D$  bits. ¿Cuál es la condición para que la lista de bloques libres ocupe menos espacio que el mapa de bits?

5. ¿Qué métodos de agrupamiento de registros tienen limitado el tamaño del registro al tamaño del bloque? Razona la respuesta.
6. Determine el número de accesos a disco necesarios para leer 20 bloques lógicos consecutivos en un sistema con una asignación encadenada, contigua e indexada.
7. ¿Qué ventajas/desventajas presenta que el tamaño de la página del sistema sea igual/distinta al tamaño del bloque físico? Razona las respuestas.
8. Indique de forma razonada qué tipo de asignación escogería para un sistema donde los ficheros se procesarán en un 90% de los casos de forma secuencial, y en un 10% de forma aleatoria. Además, la duración de un fichero en el sistema suele ser corta, presentando muchas actualizaciones.
9. El sistema MS-DOS emplea para llevar el control del espacio asignado a los ficheros una modificación de la asignación enlazada mediante la FAT. Cada entrada de ésta ocupa 16 bits. Supongamos que tenemos un disco de 1 GiB. ¿Sería aconsejable realizar más de una partición al disco, o tener una única partición con todo el disco? ¿Y si el disco fuera de 2 GiB? Razona las respuestas.
10. Supongamos un tamaño de bloque de 8 KiB. Disponemos de un fichero que inicialmente tendrá 16 KiB, y se estima ocupe un tamaño máximo de 80 KiB. Para direccionar un bloque son necesarios 16 bits.
  - (a) ¿Qué espacio ocupará el fichero en cada una de las siguientes asignaciones: contigua, encadenada e indexada?
  - (b) Determinar la fragmentación existente en cada tipo de asignación.
  - (c) Al cabo de dos meses el fichero ocupa 40 KiB. ¿Qué espacio ocupará en cada tipo de asignación?
  - (d) ¿Qué fragmentación existirá ahora?
  - (e) Tras cuatro meses el fichero ocupa 79 KiB. ¿Qué espacio ocupa y qué fragmentación existe en cada tipo de asignación?
11. Un disco tiene 1,2 GiB de capacidad con 2048 cilindros, 14 pistas por cilindro y 10 sectores por pista. Disponemos de un fichero con registros lógicos de 100 bytes, y un tamaño del bloque físico de 1 KiB. Se utiliza una técnica de agrupamiento fijo. Queremos leer los registros 20 al 30 del fichero.

- (a) Determinar qué direcciones físicas corresponden a esa operación, teniendo en cuenta que se realiza una asignación contigua, cuya entrada en el directorio es la siguiente:

Fichero	Bloque inicial	Longitud del fichero
F1	20	300

- (b) Igual pero con una asignación encadenada, donde la entrada en el directorio indica que su primer bloque es el número 9. Las diez primeras entradas de la FAT son las siguientes:

0	LIBRE
1	LIBRE
2	100
3	34
4	8
5	MALO
6	100
7	7
8	45
9	4

- (c) Igual pero con una asignación indexada, en la que el bloque índice del fichero es el 3, y su contenido es el siguiente:

347
284
1023
345
454
45
18
790
50
...

12. Resolver el problema anterior suponiendo que se utiliza una técnica de agrupamiento de longitud variable con extensión, aunque se emplean registros de tamaño fijo. Suponga que el tamaño del puntero que indica la continuación de un registro en otro bloque es 2 bytes.
13. Al realizar una comprobación de la consistencia de un sistema de ficheros **ext2** se obtienen los siguientes valores de contadores para los bloques de datos:

Uso	1	0	1	0	0	2	1	0	1	1	0	1	0	0	0	1
Libre	0	1	1	1	1	1	0	0	0	0	1	1	0	1	1	1

Determine si existen problemas, de qué tipo y cómo se resolverían.

14. Un disco de 1,2 GiB utiliza bloques de 2 KiB. Un análisis de la utilización del disco, demuestra que el espacio medio libre es de 600 KiB repartidos en 30 huecos. ¿Cuánto ocupará una lista enlazada normal, si dicha estructura se almacena en memoria, empleando punteros y enteros de 32 bits? ¿Qué ventaja representaría emplear una lista enlazada con agrupamiento? ¿Y con recuento?



---

# **APÉNDICES**

---



# Apéndice A

## Conceptos básicos sobre el hardware

---

Los sistemas operativos utilizan los recursos hardware de un sistema de computación para ofrecer servicios a los usuarios. De acuerdo con esto, es importante tener ciertos conocimientos sobre el hardware de un sistema para comprender el funcionamiento de los sistemas operativos. El objetivo de este apéndice es ofrecer esos conocimientos a aquellos lectores que no los posean, siendo conveniente hacer una lectura previa de éste antes de comenzar a leer los capítulos del libro. Si el lector ya conoce estos conceptos puede obviar su lectura.

### A.1 Elementos básicos del computador

A alto nivel, un computador consta de procesador, memoria, y componentes de E/S, pudiendo tener uno o más módulos de cada tipo. Estos componentes se encuentran interconectados para lograr la función principal del computador, que es la ejecución de programas. Por tanto, podemos distinguir cuatro elementos estructurales:

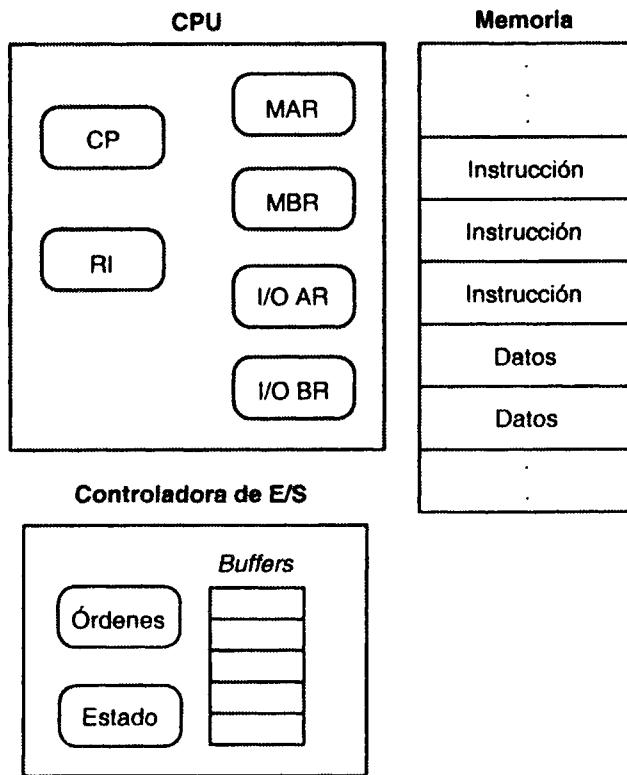
**Procesador** Controla la operación del computador y realiza sus funciones de procesamiento de datos.

**Memoria principal** Almacena programas y datos. Se suele conocer también

como memoria real o primaria y es de tipo volátil.

**Controladoras de E/S** Permiten el intercambio de datos entre el computador y el exterior.

**Sistema de interconexión** Son aquellas estructuras y mecanismos que permiten la comunicación entre los procesadores, la memoria principal y las controladoras de E/S.



**Figura A.1:** Visión a alto nivel de un computador

La figura A.1 muestra estos componentes. El procesador realiza funciones de control y de intercambio de información. Para ello dispone de una serie de registros internos cuya función específica veremos más adelante. Un módulo de memoria consta de un conjunto de localizaciones definidas por unas direcciones numeradas secuencialmente. Cada localización contiene un número binario que puede ser interpretado como un dato o una instrucción. Una controladora de E/S transfiere datos desde los dispositivos externos al procesador y a la memoria,

y viceversa. Contiene *buffers* que permiten mantener temporalmente los datos hasta que puedan ser enviados, así como unos registros de órdenes y de estado para dar las órdenes a los dispositivos, y conocer el estado de éstos.

## A.2 Registros del procesador

El procesador dispone de un conjunto de registros que proporcionan un nivel de memoria más rápida y de menor capacidad que la principal. Podemos clasificarlos en dos tipos:

**Registros visibles al usuario** El programador puede acceder a ellos directamente utilizando el lenguaje máquina o ensamblador, con objeto de minimizar las referencias a memoria principal. Esto permite optimizar la velocidad de ejecución del programa. En el caso de los lenguajes de alto nivel, un compilador que optimice intentará hacer una elección inteligente a la hora de decidir qué variables asignar a registros y cuáles a localizaciones de memoria principal. Algunos lenguajes, tales como C, permiten al programador sugerirle al compilador qué variables deberían ser mantenidas en registros.

**Registros de control y estado** El procesador los utiliza para controlar su funcionamiento, y las rutinas privilegiadas del sistema operativo para controlar la ejecución de los programas.

### A.2.1 Registros visibles al usuario

Un registro visible al usuario puede ser referenciado por medio del lenguaje máquina que ejecuta el procesador y normalmente está disponible para todos los programas, tanto los de aplicación como los del sistema. Los registros que entran normalmente dentro de esta categoría son los de datos, de dirección y de código de condición.

A los **registros de datos** les pueden ser asignadas varias funciones por parte del programador. En algunos casos son de propósito general y pueden ser usados con cualquier instrucción de la máquina que realice operaciones sobre los datos. Sin embargo, a menudo hay restricciones. Por ejemplo, puede haber registros dedicados para las operaciones de punto flotante.

Los **registros de dirección** contienen direcciones de memoria principal de datos o instrucciones. Pueden ser de propósito general o estar dedicados a un modo de direccionamiento particular.

Por último, un registro que es, al menos parcialmente, visible al usuario es el que mantiene los **códigos de condición** (o *flags*). Éstos son bits establecidos por el hardware del procesador como resultado de ciertas operaciones. Por ejemplo, una operación aritmética puede producir un resultado positivo, negativo, cero o un desbordamiento. Además de registrarse el resultado en un registro o en memoria, también se suele establecer el código de condición correspondiente.

Los bits de código de condición están reunidos en uno o más registros. Usualmente forman parte de un registro de control. Generalmente, las instrucciones de la máquina permiten que estos bits puedan ser leídos pero no modificados por el programador.

### A.2.2 Registros de control y estado

Algunos registros del procesador se emplean para controlar su funcionamiento. En la mayoría de las máquinas, éstos no son visibles al usuario. En algunos casos se puede acceder a algunos de ellos mediante instrucciones máquina ejecutadas en un modo privilegiado.

Máquinas diferentes tendrán organizaciones de registros distintas y usarán diferente terminología. Aquí se proporciona una lista de los tipos de registros más importantes, junto con una breve descripción de éstos.

Una de las funciones del procesador es intercambiar datos con la memoria principal. Para este propósito, hace uso de dos registros internos:

**Registro de dirección de memoria (MAR)** Especifica la dirección de memoria donde se va a hacer la próxima operación de lectura o escritura.

**Registro de buffer de memoria (MBR)** Contiene los datos que se van a escribir en la memoria, o que se leen de ésta.

Para el intercambio de datos con las controladoras de E/S tenemos:

**Registro de dirección de E/S (I/O AR)** Especifica una controladora de E/S particular.

**Registro de buffer de E/S (I/O BR)** Se usa para intercambiar datos entre una controladora de E/S y el procesador.

Otros registros muy importantes son:

**Contador de programa (CP)** Contiene la dirección de la próxima instrucción a ser ejecutada.

**Registro de instrucción (RI)** Contiene la instrucción que se está ejecutando.

Todos los procesadores también incluyen un registro o conjunto de registros, conocido a menudo como palabra de estado del programa (PSW), que contiene información de estado y códigos de condición. Los campos e indicadores que suele incluir son:

**Signo** Contiene el bit de signo de la última operación aritmética realizada.

**Cero** Se establece cuando el resultado de una operación aritmética es cero.

**Igual** Se activa si una comparación lógica resulta en igualdad.

**Desbordamiento** Se usa para indicar desbordamiento aritmético.

**Activar/Desactivar Interrupción** Se utiliza para activar o desactivar las interrupciones.

**Modo** Indica el modo de ejecución del procesador, usuario o supervisor.

### A.3 Ejecución de una instrucción

La función básica que realiza un computador es la ejecución de programas. Éstos constan de un conjunto de instrucciones y datos almacenados en memoria.

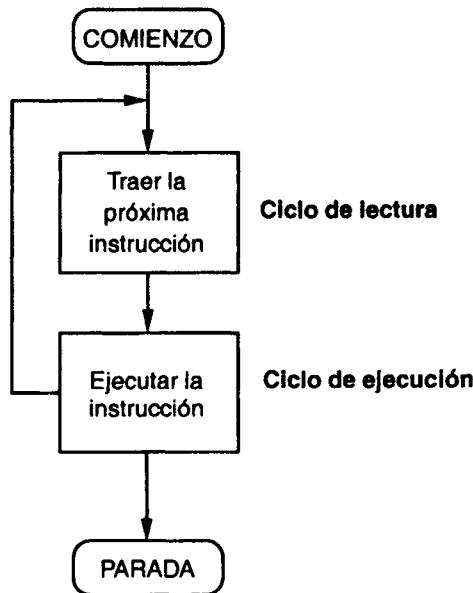
A continuación se va a analizar con detalle la ejecución de un programa. De una forma simple podemos considerar que el procesamiento de una instrucción, denominado **ciclo de instrucción**, consta de dos pasos (ver figura A.2):

**Ciclo de lectura** El procesador trae la instrucción de la memoria.

**Ciclo de ejecución** El procesador ejecuta esa instrucción.

La ejecución del programa consiste en la repetición de estos pasos tantas veces como sea necesario. Los dos pasos enumerados se pueden subdividir en varias acciones cada uno de ellos.

Al comienzo de cada ciclo, el procesador lee una instrucción de la memoria. ¿Cómo sabe éste qué instrucción es la que debe leer? Normalmente, el procesador



**Figura A.2:** Ciclo de instrucción básico

cuenta con un registro que se denomina contador de programa que le indica la próxima instrucción a leer. A menos que se le diga otra cosa, cada vez que el procesador carga una nueva instrucción incrementa el valor de este registro.

La instrucción leída se carga en el registro de instrucción. La instrucción está en código binario y especifica las acciones a realizar por el procesador. Éste interpretará la instrucción y ejecutará la acción requerida. En general, estas acciones se pueden clasificar en cuatro categorías:

**Procesador-Memoria** Los datos pueden ser transferidos desde el procesador a la memoria o desde la memoria al procesador.

**Procesador-E/S** Los datos pueden ser transferidos hacia o desde un periférico.

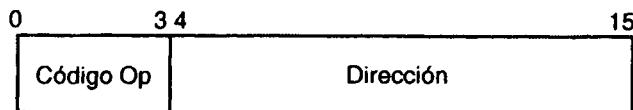
**Procesamiento de datos** El procesador puede realizar alguna operación aritmética o lógica sobre los datos.

**Control** Una instrucción puede especificar que la secuencia de ejecución sea alterada. Por ejemplo, el procesador puede leer una instrucción de la localización 357, que especifique que la próxima instrucción será la que se encuentra en la localización 419. Para recordarlo, éste almacenará en CP la dirección 419.

Así en el próximo ciclo de lectura, se tomará la instrucción de la localización 419 en vez de la 358.

La ejecución de una instrucción puede implicar una combinación de algunas de estas acciones. Consideremos un ejemplo simple usando una máquina hipotética que incluye las características listadas en la figura A.3.

El procesador contiene un registro de datos simple llamado acumulador (AC). Tanto las instrucciones como los datos son de 16 bits de longitud. El formato de la instrucción proporciona cuatro bits para el código de operación. Así podemos tener  $2^4$  códigos de operación diferentes, y  $2^{12}$  palabras de memoria que pueden ser directamente direccionadas.



(a) Formato de la instrucción



(b) Formato de entero

**Contador de Programa (CP)** = Dirección próxima instrucción

**Registro de instrucción (RI)** = Instrucción actual

**Acumulador (AC)** = Almacenamiento temporal

(c) Registros internos de la CPU

0001 = Cargar AC de la memoria

0010 = Almacenar AC en memoria

0101 = Añadir el contenido de la dirección de memoria a AC

(d) Lista parcial de códigos de operaciones

**Figura A.3:** Características de una máquina hipotética

La figura A.4 ilustra la ejecución parcial de un programa, donde se muestran la memoria y los registros del procesador implicados. El programa añade el contenido de la palabra de memoria en la dirección 725 al contenido de la palabra de memoria en la dirección 726 y almacena el resultado en esta última localización. Para ello se requieren tres instrucciones:

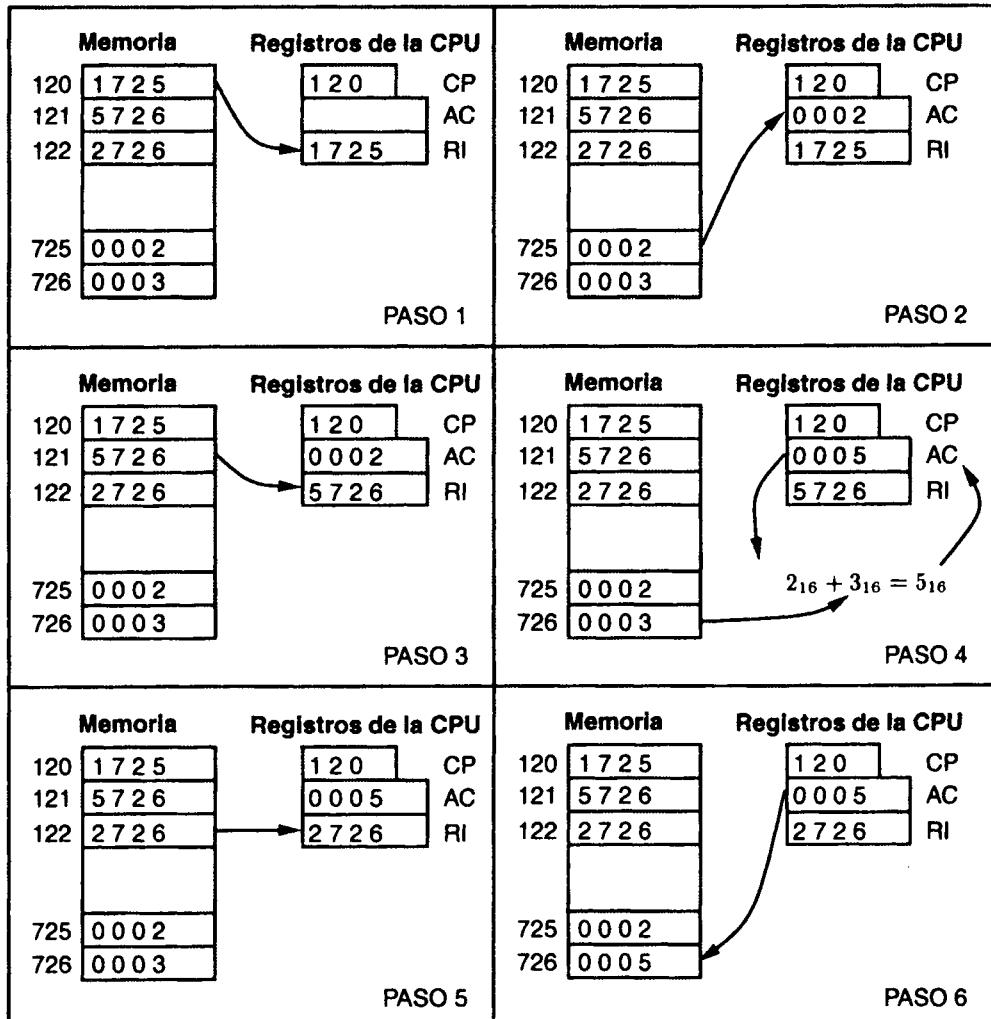


Figura A.4: Ejemplo de ejecución de un programa

1. El contador de programa contiene 120, la dirección de la primera instrucción. El contenido de la localización 120 se carga en el registro de instrucción.
2. Los cuatro primeros bits en el RI indican que se va a cargar el acumulador. Los restantes 12 bits especifican la dirección, que es 725.
3. Se incrementa el contador de programa, y se lee la próxima instrucción.
4. Se suman el contenido antiguo de AC y el contenido de la localización 726; el resultado se almacena en AC.

5. Se incrementa nuevamente el contador de programa, y se lee la próxima instrucción.
6. El contenido de AC se almacena en la localización 726.

En este ejemplo se han necesitado tres ciclos de instrucción para añadir el contenido de la localización 725 al de la 726. Disponiendo de un conjunto de instrucciones más complejas se podrían necesitar menos ciclos. Los procesadores más modernos incluyen instrucciones que contienen más de una dirección. Así, el ciclo de ejecución para una instrucción particular puede implicar más de una referencia a memoria. También, en vez de referencias a memoria, una instrucción puede especificar una operación de E/S.

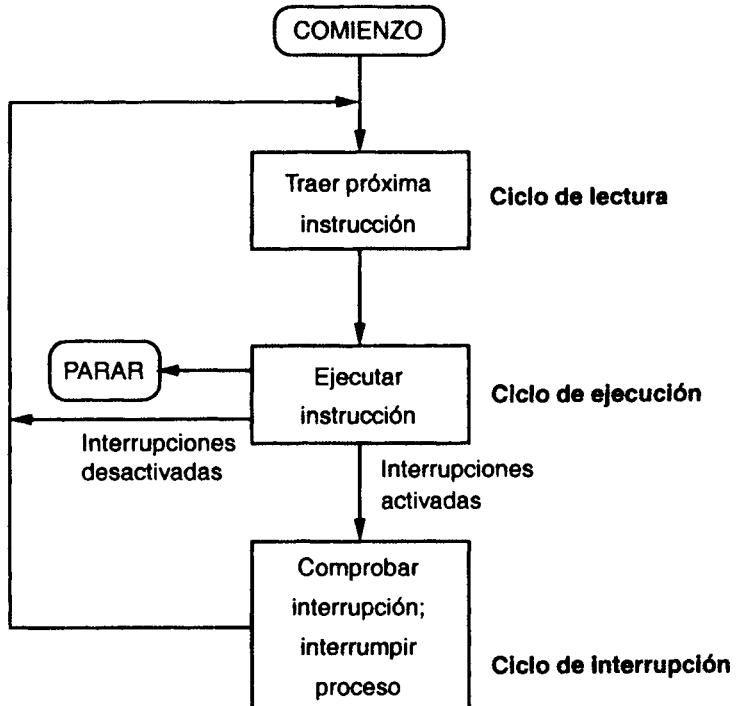
## A.4 Interrupciones

Todos los computadores proporcionan un mecanismo mediante el cuál otros módulos (dispositivos de E/S) pueden interrumpir el procesamiento normal del procesador.

Las interrupciones fueron proporcionadas inicialmente como una forma de mejorar la eficiencia del procesamiento. Sin la existencia de interrupciones, cada vez que se inicia una operación de E/S el procesador tendría que permanecer inactivo hasta que ésta terminara. Dado que los dispositivos de E/S son bastante más lentos que la CPU, esto supondría un desaprovechamiento considerable del procesador. Con las interrupciones, el procesador puede ejecutar otras instrucciones mientras se está realizando una operación de E/S.

Desde el punto de vista del programa de usuario, una interrupción sólo supone interrumpir la secuencia normal de ejecución. Cuando termina el procesamiento de la interrupción, la ejecución continúa. Por tanto, los programas de usuario no necesitan contener un código especial para tratar las interrupciones; el procesador y el sistema operativo son los responsables de suspenderlo y reanudarlo posteriormente en el mismo punto.

Para tratar las interrupciones se puede añadir un ciclo de interrupción al de instrucción, como se muestra en la figura A.5. En el ciclo de instrucción el procesador comprueba si ha ocurrido alguna interrupción; esto puede venir indicado por la presencia de alguna señal. Si no hay interrupciones pendientes, el procesador procede a la lectura de la próxima instrucción del programa actual. Si hay alguna interrupción pendiente, se suspende la ejecución del programa actual y se ejecuta la rutina manejadora de la interrupción. Generalmente, ésta forma parte del sistema operativo. Esta rutina determina la naturaleza de la interrupción



**Figura A.5:** Ciclo de instrucción con interrupciones

y realiza las acciones que se necesitan. Cuando finaliza su ejecución, el procesador puede reanudar la ejecución del programa de usuario en el punto de la interrupción.

Todo este proceso introduce una cierta carga adicional en el sistema. Deben ejecutarse instrucciones extras (el manejador de interrupciones) para determinar la naturaleza de la interrupción y decidir la acción apropiada. Sin embargo, dada la gran cantidad de tiempo que se podría gastar esperando en una operación de E/S, se obtiene un uso más eficiente del procesador con el uso de interrupciones.

#### A.4.1 Procesamiento de la interrupción

Una interrupción provoca una serie de eventos, que implican tanto al hardware como al software. La figura A.6 muestra una secuencia típica.

Cuando un dispositivo de E/S completa una operación, tienen lugar los siguientes eventos en los que está implicado el hardware:

1. La controladora emite una señal de interrupción al procesador.
2. El procesador finaliza la ejecución de la instrucción actual antes de responder a la interrupción.
3. El procesador comprueba si ha habido una interrupción, determina que se ha producido una, y envía una señal de reconocimiento a la controladora.
4. El procesador se prepara para transferir el control a la rutina de la interrupción. Para empezar, necesita guardar la información necesaria para volver a reanudar el programa actual en el punto de la interrupción. La información mínima requerida es la palabra de estado del programa (PSW) y la localización de la siguiente instrucción a ser ejecutada, que está contenida en el contador de programa. Es lo que se denomina el cambio de contexto de un proceso.
5. El procesador carga ahora en el contador de programa la localización de la rutina manejadora de la interrupción.

Una vez que el contador de programa ha sido cargado, el procesador procede al próximo ciclo de instrucción, que empieza con la lectura de una instrucción del manejador de la interrupción. La ejecución de éste da lugar a las siguientes operaciones:

1. Guarda el resto de la información del proceso que se estaba ejecutando para poder reanudarlo posteriormente. Si se interrumpe un proceso después de la instrucción localizada en  $N$ , se colocarán en la pila del núcleo el contenido de todos los registros más la dirección de la próxima instrucción ( $N + 1$ ). Se actualiza el puntero a la pila y el contador de programa apunta ahora al principio de la rutina de servicio de la interrupción.
2. El manejador de la interrupción comienza a procesarla. Esto puede incluir un examen de la información de estado correspondiente a la operación de E/S o de otro evento que haya causado la interrupción y puede implicar el envío de órdenes adicionales a la controladora de E/S.
3. Cuando finaliza el procesamiento de la interrupción, se recuperan los valores de los registros que fueron guardados anteriormente en la pila del núcleo, y se graban de nuevo en los registros.
4. Finalmente se recuperan los valores de PSW y del contador de programa de la pila. Como resultado de esto, la próxima instrucción que se ejecutará será la del programa previamente interrumpido.

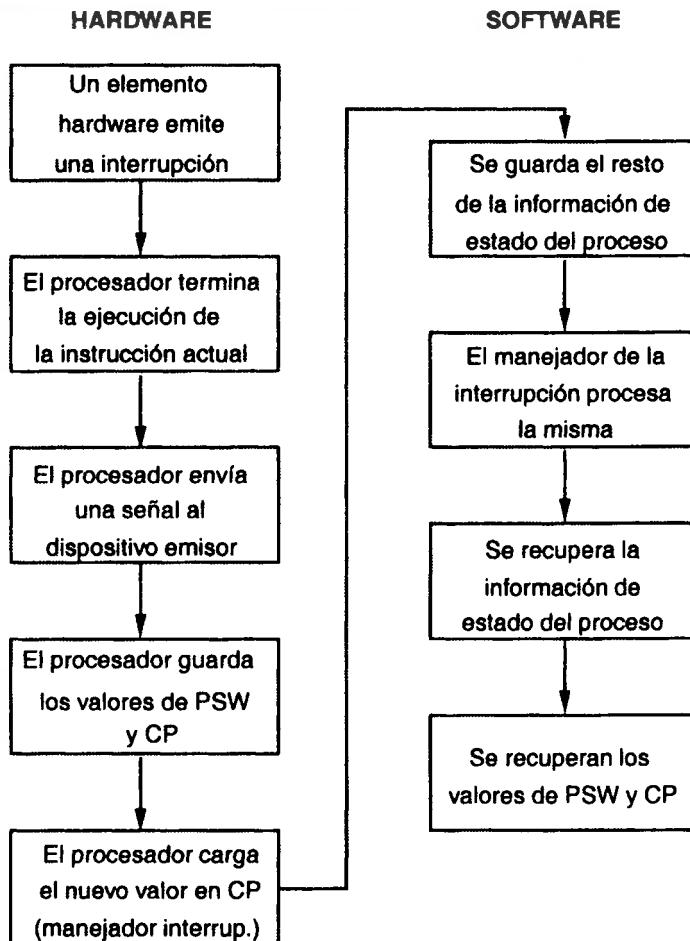


Figura A.6: Procesamiento de una interrupción

## A.5 Buses

Un computador consta de un conjunto de componentes (procesador, memoria, controladoras de E/S, etc.) que se comunican entre sí. En la figura A.7 se representan los tipos de intercambios que se necesitan, indicándose las formas principales de entrada y salida para cada tipo de módulo:

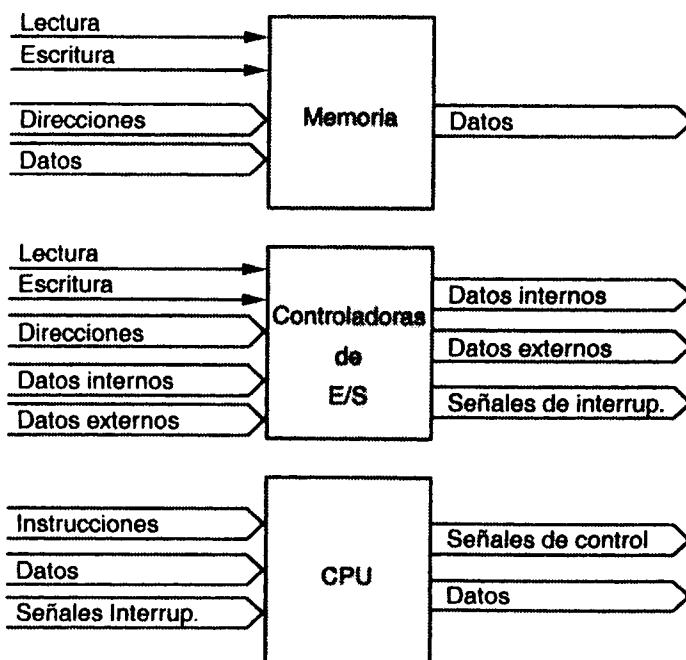
**Memoria** Un módulo de memoria suele constar de  $N$  palabras de igual longitud.

A cada palabra se le asigna una dirección numérica. Se pueden realizar operaciones de lectura o de escritura de palabras de datos. La localización

de la operación se especifica mediante una dirección.

**Controladora de E/S** Desde un punto de vista interno (al sistema de computación), la E/S es similar funcionalmente a la memoria. Se pueden realizar también dos operaciones, lectura y escritura. Una controladora de E/S puede de controlar más de un dispositivo externo, para referirnos a cada uno de ellos se le debe dar una dirección única. Además, hay caminos para los datos internos y externos, tanto para la entrada como para la salida de datos. Por último, una controladora de E/S debe ser capaz de mandar señales de interrupción al procesador.

**Procesador** El procesador lee instrucciones y datos, y posteriormente escribe los datos obtenidos como resultado de su procesamiento. También utiliza señales de control para controlar la operación global del sistema y recibe señales de interrupción.



**Figura A.7:** Módulos del computador

La estructura de interconexión que permite a cada componente comunicarse directamente con todos los demás, es el **bus**. Un bus es un medio de transmisión compartido que conecta dos o más módulos, que consta de varias líneas de comunicación. Cada línea es capaz de transmitir señales que representan un 1 ó un 0.

Dado que el bus es compartido, una señal transmitida por cualquier módulo o en cualquier línea está disponible para su recepción por cualquiera de los módulos unidos a ella. Por tanto, habrá que establecer algunas normas para que no haya problemas.

Aunque hay muchos diseños diferentes de buses, en cualquiera de ellos las líneas se suelen clasificar en tres grupos funcionales, líneas de datos, direcciones y control. Además puede haber líneas de distribución de potencia que suministren corriente a los módulos unidos a ellas.

Las líneas de datos proporcionan una vía para el movimiento de datos entre los distintos módulos del sistema. Estas se denominan de forma conjunta bus de datos. Los buses de datos suelen constar de 8, 16, o 32 líneas; este número es lo que se denomina anchura del bus de datos. Dado que cada línea sólo puede llevar 1 bit a un tiempo, el número de líneas determina cuantos bits pueden ser transferidos simultáneamente. La anchura del bus de datos es un factor clave para el rendimiento global del sistema. Por ejemplo, si el bus de datos es de 8 bits y cada instrucción tiene 16 bits, el procesador debe acceder al módulo de memoria dos veces durante cada ciclo de lectura de la instrucción.

Las líneas de direcciones se utilizan para designar la fuente o destino de los datos en el bus de datos. Por ejemplo, si el procesador ejecuta una instrucción que referencia una palabra (8, 16 o 32 bits) de datos que tiene que ser leída de la memoria, el procesador pone la dirección de la palabra deseada en las líneas de dirección. La anchura del bus de dirección determina la capacidad de memoria máxima posible que puede tener el sistema. Las líneas de dirección también se suelen usar para direccionar los puertos de E/S.

Las líneas de control sirven para controlar el acceso y el uso de las líneas de datos y de direcciones, ya que éstas son compartidas por todos los componentes del sistema.

Cuando un módulo quiere enviar datos a otro debe hacer dos cosas, obtener el uso del bus y transferir los datos a través de éste. Si un módulo desea recibir datos de otro, en primer lugar deberá obtener el uso del bus y posteriormente transferirá la petición a través de las líneas de control y de dirección apropiadas.

## Apéndice B

# Prefijos para los múltiplos binarios

---

Este libro adopta las recomendaciones aprobadas en Diciembre de 1998 por el IEC (*International Electrotechnical Commission*) [IEC99] respecto a los nombres y símbolos para los prefijos de los múltiplos binarios que se utilizan en los campos del procesamiento y transmisión de datos. Los prefijos son los que aparecen en la tabla B.1.

Factor	Nombre	Símbolo	Origen	Derivación
$2^{10}$	kibi	Ki	kilobinario: $(2^{10})^1$	kilo: $(10^3)^1$
$2^{20}$	mebi	Mi	megabinario: $(2^{10})^2$	mega: $(10^3)^2$
$2^{30}$	gibi	Gi	gigabinario: $(2^{10})^3$	giga: $(10^3)^3$
$2^{40}$	tebi	Ti	terabinario: $(2^{10})^4$	tera: $(10^3)^4$
$2^{50}$	pebi	Pi	petabinario: $(2^{10})^5$	peta: $(10^3)^5$
$2^{60}$	exbi	Ei	exabinario: $(2^{10})^6$	exa: $(10^3)^6$

Tabla B.1: Prefijo para múltiplos binarios

Estos nuevos prefijos para los múltiplos binarios no son parte del Sistema Internacional de Unidades (SI), pero para facilitar su comprensión y memorización derivan de los prefijos del SI para las potencias positivas de diez. Como puede verse en la tabla B.1, el nombre de cada prefijo deriva del nombre de éste en el SI, se toman las dos primeras letras de éste, y se le añade “bi”, que recuerda a la palabra binario. De forma similar, el símbolo nuevo deriva del correspondiente del sistema internacional añadiéndole la letra “i”, para recordar la palabra binario.

de nuevo.

A continuación se muestran algunos ejemplos y se comparan con los prefijos del SI:

1 kibibit	1 Kibit= $2^{10}$ bit=1024 bit
1 kilobit	1 kbit= $10^3$ bit=1000 bit
1 mebibyte	1 MiB= $2^{20}$ B=1.048.576 B
1 megabyte	1 MB= $10^6$ B=1.000.000 B
1 gibibyte	1 GiB= $2^{30}$ B=1.073.741.824 B
1 gigabyte	1 GB= $10^9$ B=1.000.000.000 B

# Bibliografía

---

- [Bach86] Bach, M. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.
- [Baco93] Bacon, J. *Concurrent Systems*. Addison-Wesley, 1993.
- [Barb97] Barbakati, N. *Los secretos de LINUX*. Anaya Multimedia, 1997.
- [Beck90] Beck, L. L. *System Software*. Addison-Wesley, 1990.
- [Bela66] Belady, L. *A Study of Replacement Algorithms for a Virtual Storage Computer*. IBM Systems Journal, Vol. 5, Nº 2, pág. 78-101, 1966.
- [Ben82] Ben-Ari, M. *Principles of Concurrent Programming*. Prentice-Hall, 1982
- [Ben90] Ben-Ari, M. *Principles of Concurrent and Distributed Programming*. Prentice-Hall, 1990.
- [Bic88] Bic, L. & Shaw, A. C. *The Logical Design of Operating Systems*. Prentice-Hall, 1988. 2<sup>a</sup> Edición.
- [Brin73] Brinch-Hansen, P. *Operating System Principles*. Prentice-Hall, 1973.
- [Camp93] Campbell, R.; Islam. N.; Raila, D. & Madany, P. *Designing and Implementing Choices: An Object-Oriented System in C*. Communications of the ACM, Vol. 36, Nº 9, pág. 117-126, 1993.
- [Card90] Card, R.; Ts'o, T. & Tweedie, S. *Design and implementation of the Second Extended Filesystem*. Proceedings of the First Dutch International Symposium on LINUX, 1990.

- [Carr84] Carr, R. *Virtual Memory Management*. Ann Arbor, MI: UMI Research Press, 1984.
- [Carr01] Carretero, J.; García, F.; De Miguel, P. & Pérez, F. *Sistemas Operativos. Una visión aplicada*. McGraw-Hill, 2001.
- [Chan96] Channon, D.; Koch, D. & Hannaford, M. *TSF: An object oriented address translation simulation framework*. Proceedings of the 19th Australasian Computer Science Conference (ACSC), Melbourne, Australia, 1996.
- [Coff71] Coffman, E. G.; Elphick, M. J. & Shoshani, A. *System Deadlocks*. ACM Computing Surveys, Vol. 3, pag. 67-78, junio 1971.
- [Coff73] Coffman, E. G. & Denning, P. J. *Operating Systems Theory*. Prentice-Hall, 1973.
- [Come84] Comer, D. *Operating Systems Design: the Xinu Approach*. Prentice-Hall, 1984.
- [Corn97] Cornes, P. *The LINUX A-Z*. Prentice-Hall, 1997.
- [Deit93] Deitel, H. M. *Sistemas Operativos*. Addison-Wesley, 1993. 2<sup>a</sup> Edición.
- [Denn67] Denning, P. J. *Effects of Scheduling on File Memory Operations*. Proceedings of AFIPS, SJCC, Vol. 30, pág. 9–21, 1967.
- [Denn68] Denning, P. J. *The Working Set Model for Program Behavior*. Communications of the ACM, Vol. 11, pág. 323–333, 1968.
- [Denn70] Denning, P. J. *Virtual Memory*. ACM Computing Surveys, Vol. 2, pág. 153–189, septiembre 1970.
- [Denn80] Denning, P. J. *Working Sets Past and Present*. IEEE Trans. on Software Engineering, Vol. SE-6, pág. 64–84, enero 1980.
- [Denn89] Denning, P. J.; Commer, D. E.; Gries, D.; Mulder, M. C.; Tucker, A. B.; Turner, A. J. & Young, P. R. *Computing as a Discipline*. Communications of the ACM, Vol. 32, Nº 1, pág. 9–23, enero 1989.
- [Dijk65] Dijkstra, E. *Cooperating Sequential Processes*. Technological University, Eindhoven, Holanda, 1965. (Reimpreso en *Programming Languages*, Ed. F. Genuys, Academic Press, Nueva York, 1968.)

- [Domi98] Domínguez Jiménez, J. J. & Estero Botaro, A. LINUX. *Aspectos internos*. Servicio de publicaciones del Departamento de Lenguajes y Sistemas Informáticos de la UCA, 1998.
- [Elph94] Elphinstone, K.; Russell, S. & Heiser, G. *Issues in Implementing Virtual Memory*. Technical Report of School of Computer Science and Engineering. University of NSW 2052, Australia. UNSW-CSE-TR-9411, septiembre 1994.
- [Fink88] Finkel, R. *An Operating Systems Vade Mecum*. Prentice-Hall, 1988.
- [Flyn91] Flynn, I. M. & McIver, M. A. *Understanding Operating Systems*. Brooks/Cole, 1991.
- [Foth61] Fotheringham, J. *Dynamic storage allocation in the Atlas Computer, including an automatic use of a backing store*. Communications of the ACM, Vol.4, pág. 435-436, 1961.
- [Ganc95] Gancarz, M. *The UNIX Philosophy*. Digital Press, 1995.
- [Gold95] Goldt, S. *The LINUX Programmer's Guide*. Linux Systems Lab., 1995.
- [Gros86] Grosshans, D. *File Systems. Design and Implementation*. Prentice-Hall, 1986.
- [Harb88] Harbron, T. R. *File Systems*. Prentice-Hall, 1988.
- [Have68] Havender, J. W. *Avoiding Deadlock in Multitasking Systems* IBM Systems Journal, Vol. 7, pág. 74-84, 1968.
- [Hoar74] Hoare, C. *Monitors, An Operating System Structuring Concept*. Communications of the ACM, Vol. 17, pág. 549-557, octubre 1974.
- [Hoar85] Hoare, C. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [Holt72] Holt, R. C. *Some Deadlock Properties of Computer Systems*. ACM Computing Surveys, Vol. 4, pág. 179-196, septiembre 1972.
- [IEC99] International Electrotechnical Commission. *Letter symbols to be used in electrical technology - Part 2: Telecommunications and electronics*. Amendment 2 to IEC International Standard IEC 60027-2, enero 1999.
- [John95] Jhonson, M. K. *The LINUX Kernel Hacker's Guide*. Linux Systems Lab., 1995.

- [Khal93] Khalidi, Y. A.; Talluri, M.; Nelson, M. N. & Williams, D. *Virtual Memory Support for Multiple Pages*. Technical Report, Sun Microsystems Labs Inc. SMLI TR-93-17, septiembre 1993.
- [Kilb62] Kilburn, T.; Edwards, D.; Lanigan, M. & Sumner, F. *One Level Storage System*. IRE Transactions, abril 1962.
- [Knut73] Knuth, D. E. *The Art of Computer Programming, Volume:1 Fundamental Algorithms*. Addison-Wesley, 1973. 2<sup>a</sup> Edición.
- [Kold92] Koldinger, E.J.; Chase, J.S. & Eggers, S.J. *Architectural Support for Single Address Space Operating Systems*. Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, pág. 175-186, Boston, United States, octubre 1992.
- [Krak88] Krakowiak, S. *Principles of Operating Systems*. MIT Press, 1988.
- [Leff89] Leffler, S. J.; McKusick, M. K.; Karels, M. J. & Quaterman, J. S. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.
- [Lero76] Leroudier, J. & Potier, D. *Principles of Optimality for Multiprogramming*. Proceedings International Symposium on Computer Performance Modeling, Measurement and Evaluation, marzo 1976.
- [Lewi96] Lewis, B., & Berg, D. *Threads Primer*. Prentice-Hall, 1996.
- [List93] Lister, A. & Eager, R. *Fundamentals of Operating Systems*. Springer-Verlag, 1993.
- [Liva90] Livadas, P. E. *File Structures. Theory and Practice*. Prentice-Hall, 1990.
- [Maek87] Maekawa, M.; Oldehoeft, A. E. & Oldehoeft, R. R. *Operating Systems. Advanced Concepts*. The Benjamin/Cummings Publishing Company, 1987.
- [Mile94] Milenkovic, M. *Sistemas Operativos. Conceptos y Diseño*. McGraw-Hill, 1994. 2<sup>a</sup> Edición.
- [Morr72] Morris, J. B. *Demand paging through utilization of working sets on the Maniac II*. Communications of the ACM, Vol. 15, Nº 10, pág. 867-872, 1972.

- [Nutt00] Nutt, G. *Operating Systems. A Modern Perspective.* Addison-Wesley, 2000. 2<sup>a</sup> Edición.
- [Nutt01] Nutt, G. *Kernel projects for LINUX.* Addison-Wesley, 2001.
- [Oxma95] Oxman, G. *The extended-2 filesystem overview.* Documento Web, 1995.  
[www.nondot.org/sabre/os/files/FileSystems/Ext2fs-overview-0.1.pdf](http://www.nondot.org/sabre/os/files/FileSystems/Ext2fs-overview-0.1.pdf)
- [Patt94] Patterson, D. & Hennessy, J. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kauffman, 1994.
- [Pink89] Pinkert, J. & Wear, L. *Operating Systems: Concepts, Policies, and Mechanisms.* Prentice-Hall, 1989.
- [Purc96] Editado por Purcell, J. & Robinson A. *LINUX. The complete reference.* Linux Systems Lab, 1996. 4<sup>a</sup> Edición.
- [Rash87] Rashid, R.; Tevanian, A.; Young, M. & Golub, D. *Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures.* Second Symposium on Architectural Support for Programming Languages and Operating Systems, ACM, octubre 1987.
- [Robb97] Robbins, K. A. & Robbins, S. *UNIX Programación práctica. Guía para la Conurrencia la Comunicación y los Multihilos.* Prentice-Hall, 1997.
- [Rose92] Rosenberg, J.; Keedy, J.L. & Abramson, D. *Addressing Mechanisms for Large Virtual Memories.* The Computer Journal, Vol. 35, Nº 4, 1992.
- [Rusl98] Rusling, D. A. *The LINUX Kernel.* Documento Web, 1998.  
[www.cs.us.es/archive/ldp/LDP/tlk/tlk.html](http://www.cs.us.es/archive/ldp/LDP/tlk/tlk.html)  
Traducción al español  
[lucas.hispalinux.es/htmls/manuales.html/enl-0.8.2-0.2.tar.gz](http://lucas.hispalinux.es/htmls/manuales.html/enl-0.8.2-0.2.tar.gz)
- [Schl89] Schleicher, D.L. & Taylor, R.L. *System overview of the Application System/400.* IBM Systems Journal, Vol. 28, Nº 3, 1989.
- [Silb01] Silberschatz, A.; Galvin, P. B. & Gagne, G. *Operating System Concepts.* John Wiley & Sons, Inc. 2001. 6<sup>a</sup> Edición.
- [Sing94] Singhal, M. & Shivaratri, N. *Advanced Concepts in Operating Systems.* McGraw-Hill, 1994.

- [Solt97] Soltis, F. *Inside the AS/400*. Duke Press, Loveland, CO. 1997. 2<sup>a</sup> Edición.
- [Stal96] Stallings, W. *Computer Organization and Architecture*. Prentice-Hall, 1996. 4<sup>a</sup> Edición.
- [Stal01] Stallings, William. *Sistemas Operativos*. Prentice Hall, 2001. 4<sup>a</sup> Edición.
- [Swit93] Switzer, R. W. *Operating Systems. A Practical Approach*. Prentice-Hall, 1993.
- [Tane93] Tanenbaum, A. S. *Sistemas Operativos Modernos*. Prentice-Hall Hispanoamericana, 1993.
- [Tane96] Tanenbaum, A. S. *Sistemas Operativos Distribuidos*. Prentice-Hall Hispanoamericana, 1996.
- [Tane98] Tanenbaum, A. S. & Woodhull, A. S. *Sistemas Operativos. Diseño e implementación*. Prentice-Hall, 1998. 2<sup>a</sup> Edición.
- [Vaha96] Vahalia, U. *UNIX Internals: The New Frontiers*. Prentice-Hall, 1996.
- [Wils95] Wilson, P. R.; Johnstone, M. S.; Neely, M. & Boles, D. *Dynamic Storage Allocation: A survey and critical review* Proceedings 1995 International Workshop on Memory Management, Springer Verlag LNCS, 1995.
- [Wirz95] Wirzenius, L. *LINUX System Administrator's Guide*. Linux Systems Lab., 1995.

# Índice de Materias

---

## A

- abrazo mortal**, véase interbloqueo  
**acceso directo a memoria**, 293, 296–298  
    controladora de, 296  
**almacenamiento**  
    jerarquía del, 185–187  
    requisitos de, 316  
**almacenamiento temporal**, véase buffering  
**anomalía de Belady**, 250  
**API**, 290, 292  
**asignación de espacio a ficheros**  
    contigua, 327–329  
    enlazada, 329–330  
    indexada, 331–333  
        bloque índice, 331  
    esquemas, 333

## B

- banquero, algoritmo del**, 169–174  
    algoritmo de seguridad, 173  
    estado inseguro, 171  
    estado seguro, 171  
    implementación, 172  
    inconvenientes, 173–174  
**bit**  
    de bloqueo, 236  
    de modificación, 236  
    de modo, 21, 367  
    de presencia, 236  
    de referencia, 236, 251, 252  
    de sentido, 306

## bloque

- factor de, 326  
    tamaño del, 325–326

## bloqueo

- indefinido, véase inanición  
    mutuo, véase interbloqueo

## buffer, 299

- circular, 301  
    doble, 300  
    intercambio de, 300

## buffering, 12, 15, 299–301

- buses**, 374–376  
**buzón**, 150

## C

- caché de buffers**, 302  
**caché de disco**, 301–302, 338  
**cargador**, 9  
**colocación en memoria**, 194–195, 201–203  
    mejor ajuste, 195, 201  
    peor ajuste, 201  
    primer ajuste, 195, 201  
    siguiente ajuste, 201  
**concurrencia**, 116  
    exclusión mutua, véase exclusión mutua  
    inanición, véase inanición  
    interbloqueo, véase interbloqueo  
    relaciones entre procesos, 119–122  
**conjunto de trabajo**, 258–263  
    principio del, 261

**conjunto residente de un proceso,** 230  
**gestión,** 256–266

**contexto**  
 cambio de, 69–70, 373  
 de un proceso, 69

**control de la memoria**  
 listas enlazadas, 204–206  
 mapas de bits, 203–204

**D**

**demora rotacional,** véase tiempo, de latencia

**direcciones**  
 físicas, 187  
 lógicas, 187  
 relativas, 187

**directorio(s),** 321–324  
 estructura de una entrada, 336  
 estructuras  
   de grafo acíclico, 323–324  
   de grafo general, 324  
   en árbol, 322  
 implementación, 335  
 operaciones, 321  
 raíz, 322

**disco(s)**  
 componentes, 302  
 cilindro, 302  
 plato, 302  
 sector, 302  
 superficie, 302

factor de bloque, 326  
 planificación de, véase planificación de discos

tamaño del bloque, 325–326

**dispositivo(s),** 286–289  
 controladora, 287–290  
 estructura, 287  
 registros de datos, 287, 299  
 registros de estado, 287  
 registros de órdenes, 287  
 de almacenamiento, 286  
 de bloques, 286  
 de caracteres, 286  
 de comunicación, 286

manejador de, véase manejador, de dispositivos  
 tabla de estado, 295

**E****E/S**

aislada, 289  
 controlada por interrupciones, 293, 295–296  
 controlada por programa, 293–295  
 escrutinio, 293  
 mapeada en memoria, 289  
 optimización, 299–309  
 técnicas de, 293–298

**escrutación,** véase E/S, controlada por programa

**espera activa,** 124, 137, 138, 295  
**estado**

cambio de, 51, 70  
 de un hilo, 71  
 de un proceso, 51, 53

**excepciones,** 21, 68–70

**exclusión mutua,** 119, 120, 122–137  
 algoritmo de Dekker, 123–131  
 algoritmo de Lamport, 132–133  
 algoritmo de Peterson, 131–132  
 con semáforos, 140–142  
 condiciones, 122  
 instrucciones atómicas, 134–137  
 paso de mensajes, 152–153  
 sección crítica, 119, 121  
 soluciones software, 122–133

**F**

**fallo de página,** 232–234  
**LINUX,** 275–276

**FAT,** véase MS-DOS, FAT

**fichero(s),** 316, 319–321  
 atributos del, 319  
 estructura física, 320  
 estructura lógica, 320  
 métodos de acceso, 320  
 nombre absoluto, 322  
 nombre relativo, 322  
 operaciones sobre un, 319  
 organización de, véase fichero(s), estructura lógica

- registro, 320  
campo, 320  
tipos de, 319
- fragmentación**  
externa, 328  
interna, 326
- frecuencia de fallos de página**, 263–266
- G**
- grafo de asignación de recursos**, 163
- H**
- hilos**, 70–73
- hiperpaginación**, 256, 258, 267–269
- I**
- inanición**, 92, 93, 98, 100, 120, 121, 128–129, 137, 305
- instrucciones**  
atómicas, 134–137  
de E/S directa, 289  
privilegiadas, 21–22
- instrucción**  
ejecución, 367–371  
ciclo de ejecución, 367  
ciclo de instrucción, 367  
ciclo de lectura, 367
- interbloqueo**, 120, 121, 126, 131, 162–179  
condiciones, 162–163  
detección, 165, 174–178  
*livelock*, 128–130  
modelado, 163–165  
predicción, 165, 169–174  
prevención, 165–169  
métodos directos, 166  
métodos indirectos, 166  
recuperación, 178–179
- intercambio**, 57, 88
- interfaz**  
de programación de aplicaciones abstractas, 290, 292  
de órdenes, 28
- interrupciones**, 68–70, 371–373  
deshabilitación, 133–134  
generadas por el dispositivo, 295, 296  
generadas por el temporizador, 22, 89, 94  
generadas por software, 21  
procesamiento, 372–373
- intérprete de órdenes**, 28
- L**
- LINUX**  
caché de *buffers*, 302, 311  
colas de mensajes, 156–157  
conurrencia, 153–157  
configuración, 45  
demonios, 45  
de paginación, 276–277  
directorio, 347–348  
dispositivo(s)  
de bloques, 310–311  
de caracteres, 311  
directorio de, 310  
fichero especial de, 310  
número mayor de, 310  
número menor de, 310  
E/S, 310–311  
enlace(s), 348–350  
duro, 323  
rápido, 350  
simbólico, 324  
estructura del sistema, 43–45  
fichero(s)  
tipos, 339  
gestión de memoria, 269–277  
área de intercambio, 277  
control de la memoria, 274  
fallo de página, 275–276  
regiones virtuales, 270  
sistema compañero, 274–275  
tabla de marcos, 273  
tabla de páginas, 271–273  
traducción de direcciones, 270
- hilos, 78–82  
  `pthread_create`, 78–82  
  `pthread_join`, 78–82
- historia, 42–43
- interconexiones en un sentido, 155–156
- interconexiones FIFO, 156
- lista-C, 311

- llamadas al sistema, 45  
**brk**, 277  
**chdir**, 355  
**chmod**, 355  
**chown**, 355  
**chroot**, 355  
**clone**, 77  
**close**, 311, 355  
**creat**, 355  
**exec**, 77–78  
**exit**, 77–78  
**fchmod**, 355  
**fchown**, 355  
**fork**, 77–78  
**free**, 277  
**fstat**, 355  
**ioctl**, 311  
**kill**, 154  
**link**, 355  
**lseek**, 311  
**lstat**, 355  
**malloc**, 277  
**mknod**, 156, 355  
**mount**, 355  
**msgget**, 156  
**msgrcv**, 156  
**msgsnd**, 156  
**open**, 311, 355  
**pipe**, 155  
**read**, 311, 355  
**semget**, 153  
**semop**, 153  
**signal**, 154, 155  
**stat**, 355  
**swapoff**, 277  
**swapon**, 277  
**unlink**, 355  
**umount**, 355  
**wait**, 77–78  
**write**, 311, 355  
módulos, 45  
nodo índice, 341–343  
  tabla de contenidos, 341, 350  
planificación de discos, 311  
planificación de la CPU, 106–108  
  algoritmo, 107  
  de hilos, 106  
sistemas multiprocesadores, 108  
tiempo compartido, 106–108  
tiempo real, 108  
procesos, 73–82  
  bloque de control, 74–75  
  estados, 75–76  
  imagen, 73  
  tabla de, 75  
semáforos, 153–154  
señales, 154–155  
sistema de ficheros  
  consistencia, 353–355  
  e2fsck, 346, 353–355  
  ext2, 338–355  
  optimización, 352–353  
  VFS, 45, 340  
superbloque, 343–346  
**llamadas al sistema**, 30–34, 68–70  
  funcionamiento, 30
- ## M
- manejador**  
  de dispositivos, 7, 9, 289–292  
  instalación, 292  
  de dispositivos reconfigurable, 292  
  de interrupciones, 10, 68, 295, 372  
**máquina virtual**, 40–41  
  Java, 41  
  monitor de, 40  
  VM/370, 40  
  VMWare, 41  
**mecanismos de protección**, 21–22  
  instrucciones privilegiadas, 21–22  
  modo de operación dual, 21, véase  
    modos de ejecución  
  para la CPU, 22  
  para la E/S, 21  
  para la memoria, 21  
**memoria**  
  administrador de, 28, 190–191  
  asignación contigua, 191  
  asignación no contigua, 192  
  colocación, véase colocación en me-  
    moria  
  compactación, 200  
  fragmentación externa, 200

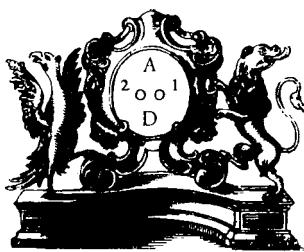
- fragmentación interna, 194  
listas enlazadas, véase control de la memoria, listas enlazadas  
mapas de bits, véase control de la memoria, mapas de bits  
monoprogramación, 192  
multiprogramación con particiones fijas, 193–197  
multiprogramación con particiones variables, 197–206  
paginación, véase paginación  
protección, 192, 196  
segmentación, véase segmentación  
sistema compañero, 206–208  
    LINUX, 274–275  
tipos  
    caché, 186  
    principal, 186  
    real, 230  
    secundaria, 186  
    virtual, 186
- memoria virtual**, 229–277  
    área de intercambio, 234–235  
        LINUX, 277  
    conjunto residente, 230  
    control de carga, 267–269  
    fallo de página, 232–234  
    funciones del gestor de, 243–269  
    gestión del conjunto residente, 256–266  
    paginación, 231–269  
    política de colocación, 245  
    política de lectura, 244  
    política de limpieza, 266–267  
    política de sustitución, 245–256  
        algoritmo del reloj, 251–252  
        algoritmo del reloj mejorado, 252–253  
        algoritmo FIFO, 248–250  
        algoritmo LRU, 250–251  
        algoritmo óptimo, 247–248  
        almacenamiento intermedio de páginas, 254–256  
    principio de localidad, 230–231  
    sustitución de página, 233
- mensaje**  
estructura, 149
- paso de, 30–34, 68–70, 121, 149–153  
direcccionamiento directo, 150  
direcccionamiento indirecto, 150, 152  
primitiva receive, 31, 149–152  
primitiva send, 31, 149–152  
sincronización, 150–152
- modelo cliente-servidor**, 20, 42
- modos de ejecución**  
modo núcleo, 4  
modo supervisor, 21  
modo usuario, 4, 21
- monitor**  
residente, 8–11  
    componentes, 9
- monitores**, 144–148  
estructura, 145  
variables de condición, 145  
    operación csignal, 145  
    operación cwait, 145  
ventajas sobre semáforos, 147–148
- MS-DOS**  
asignación de ficheros, 330  
estructura del sistema, 36  
estructura directorio, 336  
FAT, 330
- multiprogramación**, 15, 17, 50, 115–117  
grado de, 16, 87–88, 267–269
- multitarea**, 49
- N**
- núcleo**, 5  
    administrador de E/S, 28  
    administrador de ficheros, 28  
    administrador de memoria, 28, 190–191  
    administrador de procesos, 28  
    complejo, 33–34  
    dinámico, 45  
    mínimo, 33–34
- O**
- OS/2**  
estructura, 38–39
- P**
- paginación**, 208–217

- compartición, 215–216  
 fragmentación interna, 216  
 marcos, 208  
 por demanda, 244  
 prepaginación, 244–245  
 protección, 212–215  
 páginas, 208  
 tabla de marcos, 210–211, 235  
 tabla de páginas, *véase* tabla de páginas  
 tamaño de las páginas, 216–217, 246–247  
 traducción de direcciones, 211–212
- planificación**, 14, 86  
 a corto plazo, *véase* planificación de la CPU  
 a largo plazo, *véase* planificación de trabajos  
 a medio plazo, 86, 88  
 apropiativa, 88–89  
 de hilos, 105–106  
 evaluación de algoritmos, 102–105  
 implementación, 105  
 modelo de colas, 104–105  
 modelo determinista, 104  
 simulación, 105  
 función de selección, 88  
 modo de decisión, 88  
 no apropiativa, 88–89  
 regla de arbitraje, 88
- planificación de discos**, 302–309, 338  
 algoritmos  
   C-LOOK, 308  
   C-SCAN, 307–308  
   del ascensor, *véase* planificación de discos, algoritmos, LOOK  
   FIFO, 304  
   FSCAN, 309  
   LOOK, 307  
   N-SCAN, 309  
   SCAN, 306–308  
   SSTF, 304–306  
 criterios, 304
- planificación de la CPU**, 16, 17, 86, 88–103, 295  
 criterios, 102–103  
 FIFO, 89–91, 100
- HRRN, 93–94  
 tasa de respuesta, 93  
 multinivel, 98  
 multinivel realimentado, 99–102  
 prioridades, 96–98  
 envejecimiento, 98  
 RR, 94–96, 100  
 cuanto, 94–96, 100  
 SPN, 91–92  
 SRT, 92–93
- planificación de trabajos**, 16, 86–88
- problema del productor/consumidor**, 142, 301  
 con monitores, 147  
 con semáforos, 142–144
- procesamiento**  
 automático, 9, 10  
 concurrente, *véase* concurrencia  
 fuera de línea, 11, 15  
 por lotes, 8
- proceso**, 27, 49  
 administrador de, 28  
 bloque de control, 60–65, 67, 70  
 cambio de, 67–70  
 comunicación, 121–122  
 contexto del, 69–70  
 creación, 51, 66  
 estados suspendidos, 57–59  
 generación de, 52  
 hijo, 52  
 hilos, 70–73  
 identificador, 61, 63, 66  
 imagen, 59–63  
 ligero, 71  
 limitado por CPU, 13, 91  
 limitado por E/S, 13, 91  
 modelo de estados, 53–59  
 padre, 52  
 pesado, 71  
 terminación, 52  
 transición entre estados, 51  
 diagrama de, 54, 86
- unidad de despacho, 71  
 unidad propietaria de recursos, 71
- programa**, 50  
 carga, 50  
 compilación, 50

- de aplicación, 3  
del sistema, 3  
enlazado, 50  
**puerto**, 150  
**puerto de E/S**, 287
- R**
- recurso(s)**, 117–118  
  compartible, 118  
  compartición, 120–121  
    coherencia de datos, 120–121  
  competencia, 119–120  
  consumible, 118  
  crítico, 118, 119, 161  
  reutilizable, 118
- registro(s)**  
  base, 22, 65, 189  
  base a la tabla de páginas, 212  
  base de la tabla de segmentos, 219  
  bits de códigos de condición, 365  
  contador de programa, 62, 65, 69–70, 366  
  de control y estado, 365–367  
  de datos, 365  
  de dirección, 365  
  de estado, 62, 65  
  de instrucción, 367  
  longitud de la tabla de segmentos, 219  
  límite, 22, 65, 189, 192  
  PSW, 367  
  visibles al usuario, 62, 365–366
- rendimiento**, 86, 103, 115
- reubicación**, 191
- S**
- segmentación**, 217–222  
  c colocación de segmentos, 222  
  compartición, 219  
  fragmentación, 222  
  protección, 221  
  segmentos, 217  
    tabla de segmentos, 217–219  
    traducción de direcciones, 219
- segmentación paginada**, 222  
  traducción de direcciones, 222
- semáforo(s)**, 137–144
- binario, 139  
implementaciones, 139–141  
operación signal, 138–139  
operación wait, 138–139  
spinlock, 138
- señales**, 149
- sistema de E/S**, 285, 316  
  administrador del, 28  
  estructura, 289–290
- sistema de ficheros**, 316–338  
  administrador del, 28, 316–318  
  agrupamiento de registros, 326  
  diseño, 324–336  
  fiabilidad, 336–337  
    consistencia, 337  
    copias de seguridad, 337  
  físico, 318  
  fragmentación externa, 328  
  fragmentación interna, 326  
  interfaz del, 318–324  
  listas enlazadas, 333–335  
    con agrupamiento, 335  
    con recuento, 335  
  lógico, 318  
  mapas de bits, 333  
  métodos de asignación, véase asignación de espacio a ficheros  
  rendimiento, 337–338  
  tamaño del bloque, 325–326
- sistema de protección**, 28
- Sistema(s)**  
  batch, véase Sistema(s) operativo(s), por lotes  
  en red, 20–21  
  llamadas al, véase llamadas al sistema  
  multihilo, 71  
  multiprocesadores, 19–21, 106, 116  
  paralelos, 19–20  
    multiprocesamiento asimétrico, 20  
    multiprocesamiento simétrico, 20
- Sistema(s) operativo(s)**  
  abstracción del hardware, 5  
  como administrador de recursos, 5  
  concepto, 4  
  de tiempo compartido, 17–18, 94  
  de tiempo real, 18

- duros, 19
  - suaves, 19
  - distribuidos, 20–21, 28
    - middleware*, 20
  - estructura, 32–42
    - máquina virtual, *véase* máquina virtual
    - micronúcleo, 41–42
    - modular, 37–40
    - monolítica, 34–36
    - orientado a objetos, 39–40
    - por capas, 37–39
  - evolución histórica, 6
  - funciones, 27–29
  - interactivos, 6, 17
  - monoprogramados, 6, 18
  - monousuarios, 6, 18
  - multiusuarios, 18
  - núcleo del, *véase* núcleo
  - para computadores personales, 18
  - por lotes, 7–11, 17
    - por lotes multiprogramados, 15–17
    - servicios del, 30–34, 118, 121
  - spooling***, 13, 15
  - superposición**, 229
- T**
- tabla de páginas**, 210–211, 236–243
    - estructura, 236
    - invertida, 241–243
    - multinivel, 238–241
    - TLB, 211, 212
  - tabla de referencias indirectas**, 292
  - tablas**
    - de E/S, 65
    - de ficheros, 65
    - de memoria, 65
    - de procesos, 63–65
  - tarea**, 49
  - tarjetas de control**, 9
  - THE**
    - estructura, 38
  - tiempo**
    - de acceso, 303
    - de acceso total, 304
    - de búsqueda, 302
    - de espera, 90
  - de latencia, 303
  - de respuesta, 17, 90, 103
  - de retorno, 10, 90, 103
  - de retorno normalizado, 90
  - de servicio estimado, 92
  - de transferencia, 303
- trabajo**, 7–17
- traducción de direcciones**
- en sistemas de segmentación paginada, 222
  - en sistemas paginados, 211–212
  - en sistemas segmentados, 219
  - en tiempo de carga, 189
  - en tiempo de compilación, 188
  - en tiempo de ejecución, 189
  - LINUX, 270
- U**
- UNIX**
- demonios, 34, 36
  - estructura, 35–36





*Este libro se terminó de componer el día 12 de junio,  
en los talleres de Consegraf S.L.*



textos básicos  
UNI  
VER  
SITA  
RIOS



ISBN 978-84-7786-716-6

9 788477 867166