

PRÁCTICAS DE SISTEMAS OPERATIVOS

De la base al diseño

Jesús CARRETERO PÉREZ

Félix GARCÍA CARBALLEIRA

Fernando PÉREZ COSTOYA

Digitalización con Propósito Académico, realizado por:	
María A. González P.	16.337.220
Camilo E. Hernández A.	16.236.427
Edermary del J. Zabala G.	16.932.924

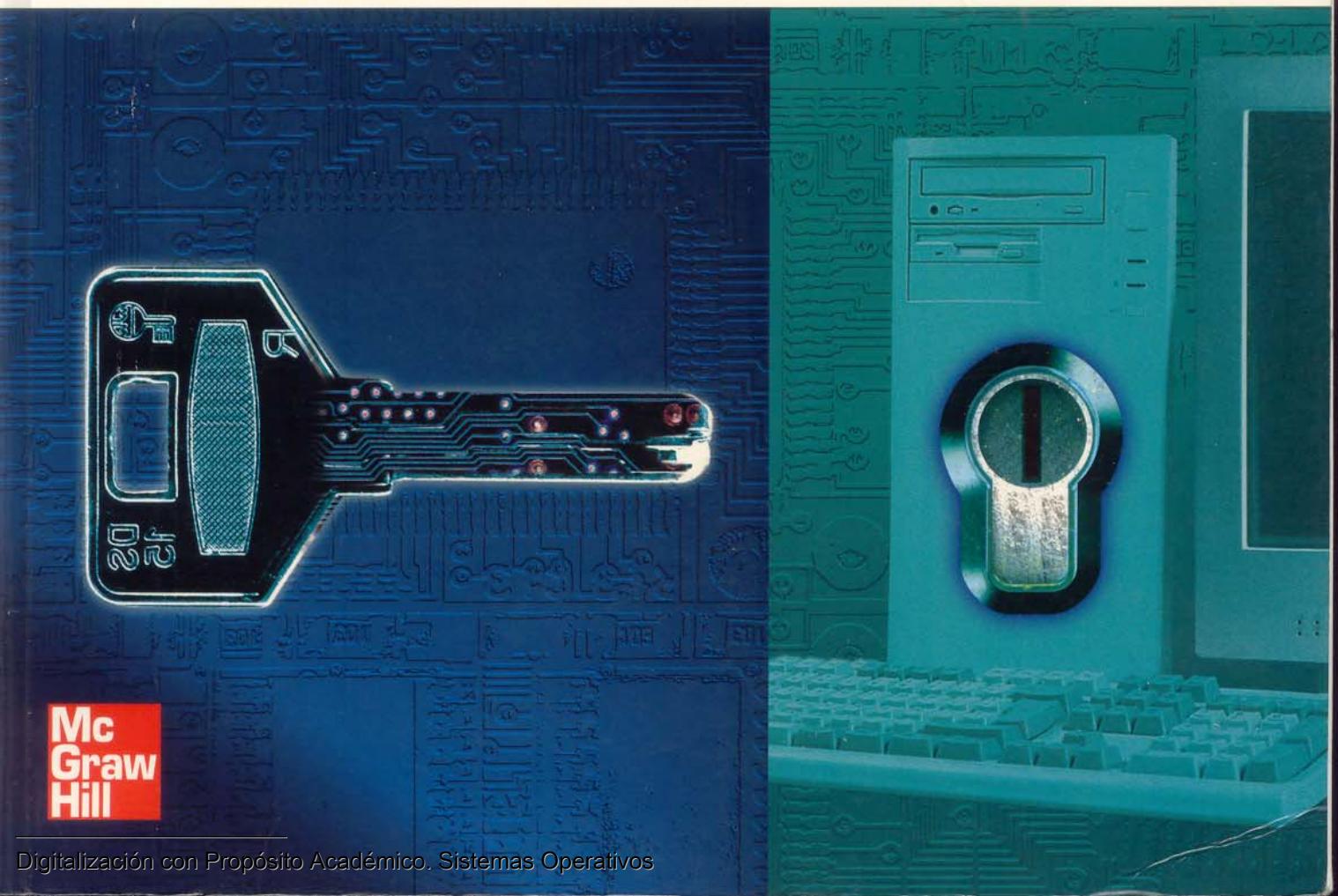


Cátedra: Sistemas Operativos

PRÁCTICAS DE SISTEMAS OPERATIVOS

De la base al diseño

Jesús CARRETERO PÉREZ
Félix GARCÍA CARBALLEIRA Fernando PÉREZ COSTOYA



**Mc
Graw
Hill**

LIBRO DE PRÁCTICAS DE SISTEMAS OPERATIVOS

LIBRO DE PRÁCTICAS DE SISTEMAS OPERATIVOS

Jesús Carretero Pérez

Catedrático del Departamento
de Informática
Escuela Politécnica Superior
Universidad Carlos III de Madrid

Félix García Carballeira

Profesor Titular del Departamento
de Informática
Escuela Politécnica Superior
Universidad Carlos III de Madrid

Fernando Pérez Costoya

Profesor Titular del Departamento de Arquitectura
y Tecnología de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid



MADRID • BUENOS AIRES • CARACAS • GUATEMALA • LISBOA • MÉXICO
NUEVA YORK • PANAMÁ • SAN JUAN • SANTAFÉ DE BOGOTÁ • SANTIAGO • SÃO PAULO
AUCKLAND • HAMBURGO • LONDRES • MILÁN • MONTREAL • NUEVA DELHI
PARÍS • SAN FRANCISCO • SIDNEY • SINGAPUR • SAN LUIS • TOKIO • TORONTO

Contenido

Prólogo	IV
1. INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS	1
1.1. Conceptos generales sobre sistemas operativos	2
1.1.1. Funciones del sistema operativo	2
1.1.2. Componentes del sistema operativo	2
1.1.3. Activación del sistema operativo	3
1.2. Práctica: Instalación del sistema operativo Linux	5
1.2.1. Pasos a seguir para la instalación de la distribución <i>Red Hat</i>	6
1.3. El <i>minikernel</i> : Un entorno para el desarrollo de prácticas de diseño de sistemas operativos	9
1.3.1. Características del <i>minikernel</i>	11
1.3.2. Descripción del entorno de desarrollo	14
1.3.3. Consideraciones generales sobre las prácticas basadas en el <i>minikernel</i>	24
1.3.4. Código fuente de apoyo	24
1.3.5. Recomendaciones generales	25
1.3.6. Entrega de documentación	26
1.3.7. Bibliografía	26
1.4. Práctica: Introducción de una nueva llamada al sistema en el <i>minikernel</i>	26
1.4.1. Objetivo de la práctica	26
1.4.2. Descripción de la funcionalidad que debe desarrollar el alumno	26
2. PROCESOS	29
2.1. Conceptos básicos de procesos	30
2.1.1. Procesos ligeros	32
2.1.2. Planificación	33
2.1.3. Señales	35
2.2. Servicios de procesos	36
2.2.1. Servicios POSIX para la gestión de procesos	36
2.2.2. Servicios POSIX de gestión de procesos ligeros	38
2.2.3. Servicios POSIX para la gestión de señales y temporizadores	40
2.3. Práctica: Monitorización de procesos en WINDOWS con Ntintern	41
2.3.1. Objetivos de la práctica	41

2.3.2.	Descripción de la funcionalidad que debe desarrollar el alumno	41
2.3.3.	Recomendaciones generales	42
2.3.4.	Entrega de documentación	43
2.3.5.	Bibliografía	43
2.4.	Práctica: Monitorización de procesos en LINUX	43
2.4.1.	Objetivos de la práctica	43
2.4.2.	Descripción de la funcionalidad que debe desarrollar el alumno	43
2.4.3.	Recomendaciones generales	46
2.4.4.	Entrega de documentación	46
2.4.5.	Bibliografía	46
2.5.	Práctica: Descubrir la jerarquía de procesos	46
2.5.1.	Objetivos de la práctica	46
2.5.2.	Descripción de la funcionalidad que debe desarrollar el alumno	46
2.5.3.	Recomendaciones generales	47
2.5.4.	Entrega de documentación	48
2.5.5.	Bibliografía	48
2.6.	Práctica: Gestión de señales en POSIX	48
2.6.1.	Objetivos de la práctica	48
2.6.2.	Descripción de la funcionalidad que debe desarrollar el alumno	48
2.6.3.	Código fuente de apoyo	49
2.6.4.	Recomendaciones generales	49
2.6.5.	Entrega de documentación	49
2.6.6.	Bibliografía	49
2.7.	Práctica: Intérprete de mandatos sencillo	50
2.7.1.	Objetivos de la práctica	50
2.7.2.	Descripción de la funcionalidad que debe desarrollar el alumno	50
2.7.3.	Código fuente de apoyo	53
2.7.4.	Entrega de documentación	54
2.7.5.	Bibliografía	54
2.8.	Gestión de procesos en el <i>minikernel</i>	54
2.8.1.	Los procesos en el <i>minikernel</i>	54
2.8.2.	Consideraciones sobre la implementación de procesos	55
2.9.	Práctica: Inclusión de una llamada bloqueante de temporización en el <i>minikernel</i>	60
2.9.1.	Objetivos de la práctica	60
2.9.2.	Descripción de la funcionalidad que debe desarrollar el alumno	60
2.10.	Práctica: Planificación <i>round-robin</i> en el <i>minikernel</i>	61
2.10.1.	Objetivos de la práctica	61
2.10.2.	Descripción de la funcionalidad que debe desarrollar el alumno	61
2.11.	Práctica: Planificación por prioridades en el <i>minikernel</i>	62
2.11.1.	Objetivos de la práctica	62
2.11.2.	Descripción de la funcionalidad que debe desarrollar el alumno	62
2.12.	Práctica: Planificación tipo Linux en el <i>minikernel</i>	63
2.12.1.	Objetivos de la práctica	63
2.12.2.	Descripción de la funcionalidad que debe desarrollar el alumno	63
2.13.	Práctica: Implementación de procesos ligeros en el <i>minikernel</i>	63
2.13.1.	Objetivos de la práctica	63
2.13.2.	Descripción de la funcionalidad que debe desarrollar el alumno	64

3. GESTIÓN DE MEMORIA.....	67
3.1. Conceptos básicos sobre gestión de memoria.....	68
3.1.1. Modelo de memoria de un proceso	68
3.1.2. Memoria virtual	71
3.1.3. Operaciones sobre las regiones de un proceso.....	74
3.1.4. Proyección de archivos en memoria	76
3.1.5. Servicios de gestión de memoria.....	76
3.2. Práctica: Análisis del mapa de memoria de los procesos	78
3.2.1. Objetivos de la práctica	78
3.2.2. El archivo /proc/pid/maps	78
3.2.3. Descripción de la funcionalidad que debe desarrollar el alumno	79
3.2.4. Entrega de documentación	81
3.2.5. Bibliografía	81
3.3. Práctica: Obtención de estadísticas sobre el mapa de memoria de los procesos	81
3.3.1. Objetivos de la práctica	81
3.3.2. Descripción de la funcionalidad que debe desarrollar el alumno	81
3.3.3. Recomendaciones generales	82
3.3.4. Entrega de documentación	83
3.3.5. Bibliografía	83
3.4. Práctica: Estudio de los fallos de página de un proceso en Linux.....	83
3.4.1. Objetivos de la práctica	83
3.4.2. Descripción de la funcionalidad que debe desarrollar el alumno	83
3.4.3. Material de apoyo	84
3.4.4. Entrega de documentación	84
3.4.5. Bibliografía	85
3.5. Práctica: Estudio de los fallos de página de un proceso en Windows	85
3.5.1. Objetivos de la práctica	85
3.5.2. Descripción de la funcionalidad que debe desarrollar el alumno	85
3.5.3. Recomendaciones generales	87
3.5.4. Material de apoyo	87
3.5.5. Entrega de documentación	87
3.5.6. Bibliografía	87
3.6. Práctica: Gestión de memoria dinámica	87
3.6.1. Objetivos de la práctica	87
3.6.2. Conceptos generales sobre la memoria dinámica	88
3.6.3. Descripción de la funcionalidad que debe desarrollar el alumno	89
3.6.4. Código fuente de apoyo	92
3.6.5. Recomendaciones generales	92
3.6.6. Entrega de documentación	92
3.6.7. Bibliografía	92
3.7. Práctica: Proyección de archivos en memoria	92
3.7.1. Objetivos de la práctica	92
3.7.2. Descripción de la funcionalidad que debe desarrollar el alumno	93
3.7.3. Código fuente de apoyo	95
3.7.4. Recomendaciones generales	95
3.7.5. Entrega de documentación	95
3.7.6. Bibliografía	95

3.8.	Práctica: Uso de bibliotecas dinámicas	96
3.8.1.	Objetivos de la práctica	96
3.8.2.	Descripción de la funcionalidad que debe desarrollar el alumno	96
3.8.3.	Código fuente de apoyo	100
3.8.4.	Entrega de documentación	100
3.8.5.	Bibliografía	100
3.9.	Práctica: Monitor del uso de memoria de un programa.....	100
3.9.1.	Objetivos de la práctica	100
3.9.2.	Descripción de la práctica	101
3.9.3.	Organización del <i>software</i> del monitor	103
3.9.4.	Descripción de la funcionalidad que debe desarrollar el alumno	106
3.9.5.	Código fuente de apoyo	109
3.9.6.	Recomendaciones generales	110
3.9.7.	Entrega de documentación	110
3.9.8.	Bibliografía	110
4. COMUNICACIÓN Y SINCRONIZACIÓN DE PROCESOS		111
4.1.	Conceptos básicos	112
4.1.1.	Problemas clásicos de comunicación y sincronización	112
4.1.2.	Mecanismos y servicios de comunicación	115
4.1.3.	Interbloqueos	124
4.2.	Práctica: Intérprete de mandatos con tuberías	125
4.2.1.	Descripción de la funcionalidad que debe desarrollar el alumno	125
4.2.2.	Desarrollos de la práctica	127
4.2.3.	Entrega de documentación	128
4.2.4.	Bibliografía	128
4.3.	Práctica: Productor-consumidor utilizando procesos ligeros	128
4.3.1.	Objetivo de la práctica	128
4.3.2.	Descripción de la funcionalidad que debe desarrollar el alumno	128
4.3.3.	Entrega de documentación	130
4.3.4.	Bibliografía	130
4.4.	Diseño e implementación de semáforos en el <i>minikernel</i>	130
4.4.1.	Objetivo de la práctica	130
4.4.2.	Descripción de la funcionalidad que debe desarrollar el alumno	130
4.4.3.	Entrega de documentación	132
4.4.4.	Bibliografía	132
4.5.	Diseño e implementación de <i>smtex</i> en el <i>minikernel</i>	132
4.5.1.	Objetivo de la práctica	132
4.5.2.	Descripción de la funcionalidad que debe desarrollar el alumno	132
4.5.3.	Entrega de documentación	134
4.5.4.	Bibliografía	134
4.6.	Detección de interbloqueos en los <i>smtex</i> del <i>minikernel</i>	134
4.6.1.	Objetivo de la práctica	134
4.6.2.	Interbloqueos para <i>smtex</i> frente a interbloqueos para semáforos	135
4.6.3.	Descripción de la funcionalidad que debe desarrollar el alumno	135
4.6.4.	Entrega de documentación	137
4.6.5.	Bibliografía	137

5. ENTRADA/SALIDA, ARCHIVOS Y DIRECTORIOS	139
5.1. Conceptos básicos de entrada/salida	140
5.2. Manejadores de dispositivos: el reloj y el terminal	141
5.2.1. El manejador del reloj	141
5.2.2. El manejador del terminal	143
5.3. Práctica: Monitorización de entrada/salida en Windows: Perlmon	144
5.3.1. Objetivos de la práctica	144
5.3.2. Descripción de la funcionalidad que debe desarrollar el alumno	145
5.3.3. Recomendaciones generales	146
5.3.4. Entrega de documentación	147
5.3.5. Bibliografía	147
5.4. Práctica: Implementación de un manejador de reloj en el <i>minikernel</i>	147
5.4.1. Objetivo de la práctica	147
5.4.2. Descripción de la funcionalidad que debe desarrollar el alumno	147
5.5. Práctica: Implementación de un manejador de terminal básico en el <i>minikernel</i>	149
5.5.1. Objetivo de la práctica	149
5.5.2. Descripción de la funcionalidad que debe desarrollar el alumno	149
5.6. Práctica: Diseño y programación de un manejador de terminal avanzado en el <i>minikernel</i>	150
5.6.1. Objetivo de la práctica	150
5.6.2. Descripción de la funcionalidad que debe desarrollar el alumno	150
5.7. Conceptos básicos de archivos	152
5.8. Conceptos básicos de directorios	154
5.9. Servicios de archivos y directorios	156
5.9.1. Servicios POSIX para archivos y directorios	156
5.9.2. Servicios Win32 para archivos y directorios	158
5.10. Práctica: Monitorización de archivos: Filermen	159
5.10.1. Objetivos de la práctica	159
5.10.2. Descripción de la funcionalidad que debe desarrollar el alumno	160
5.10.3. Recomendaciones generales	162
5.10.4. Entrega de documentación	162
5.10.5. Bibliografía	162
5.11. Práctica: Gestión de cuotas de disco usando <i>scripts</i> de UNIX	162
5.11.1. Objetivos de la práctica	162
5.11.2. Descripción de la funcionalidad que debe desarrollar el alumno	162
5.11.3. Recomendaciones generales	163
5.11.4. Entrega de documentación	163
5.11.5. Bibliografía	163
5.12. Práctica: Implementación del mandato du	163
5.12.1. Objetivos de la práctica	163
5.12.2. Descripción de la funcionalidad que debe desarrollar el alumno	164
5.12.3. Código fuente de apoyo	165
5.12.4. Recomendaciones generales	165
5.12.5. Entrega de documentación	165
5.12.6. Bibliografía	165
5.13. Práctica: Gestión de archivos usando <i>scripts</i> de UNIX	166
5.13.1. Objetivos de la práctica	166

5.13.2. Descripción de la funcionalidad que debe desarrollar el alumno	166
5.13.3. Recomendaciones generales	167
5.13.4. Entrega de documentación	167
5.13.5. Bibliografía	167
5.14. Práctica: Interpretación de mandatos con redirección y mandatos internos.....	167
5.14.1. Descripción de la funcionalidad que debe desarrollar el alumno	167
5.14.2. Código fuente de apoyo	168
5.14.3. Entrega de documentación	168
5.14.4. Bibliografía	168
5.15. Práctica: Llamadas al sistema para la gestión de archivos y directorios en POSIX: <code>ncp</code> y <code>n1s</code>	168
5.15.1. Objetivos de la práctica	168
5.15.2. Descripción de la funcionalidad que debe desarrollar el alumno	169
5.15.3. Código fuente de apoyo	170
5.15.4. Recomendaciones generales	171
5.15.5. Entrega de documentación	171
5.15.6. Bibliografía	171
5.16. Práctica: Llamadas al sistema para la gestión de archivos y directorios en Windows: <code>ncopy</code> y <code>ndir</code>	171
5.16.1. Objetivos de la práctica	171
5.16.2. Descripción de la funcionalidad que debe desarrollar el alumno	171
5.16.3. Recomendaciones generales	173
5.16.4. Entrega de documentación	173
5.16.5. Bibliografía	174
5.17. Conceptos de diseño de sistemas de archivos y directorios	174
5.17.1. Estructura del sistema de archivos	174
5.18. Práctica: Creación de un sistema de archivos: <code>mktfs</code>	177
5.18.1. Objetivos	177
5.18.2. Descripción de la funcionalidad que debe desarrollar el alumno	177
5.18.3. Código fuente de apoyo	179
5.18.4. Recomendaciones generales	179
5.18.5. Entrega de documentación	179
5.18.6. Bibliografía	180
5.19. Práctica: Diseño e implementación de archivos con bandas	180
5.19.1. Objetivos	180
5.19.2. Entorno de desarrollo de la práctica	180
5.19.3. Sistemas de archivos con bandas	180
5.19.4. Organización de la información	184
5.19.5. Descripción de la funcionalidad que debe desarrollar el alumno	184
5.19.6. Recomendaciones generales	187
5.19.7. Entrega de documentación	188
5.19.8. Bibliografía	188
6. INTRODUCCIÓN A LOS SISTEMAS DISTRIBUIDOS	189
6.1. Conceptos básicos	190
6.2. Protocolos de comunicación	191
6.3. Comunicación de procesos en sistemas distribuidos	192
6.3.1. <i>Sockets</i>	192

6.3.2.	Llamadas a procedimientos remotos	198
6.3.3.	Introducción a las RPC de SUN	199
6.4.	Práctica: Parámetros de configuración y monitorización de conexiones de red en Windows 2000	204
6.4.1.	Objetivos de la práctica	204
6.4.2.	Recomendaciones generales	206
6.4.3.	Entrega de documentación	206
6.4.4.	Bibliografía	206
6.5.	Práctica: Uso del mandato <code>rpcinfo</code> en Linux	207
6.5.1.	Objetivos de la práctica	207
6.5.2.	Descripción de la funcionalidad que debe desarrollar el alumno	207
6.5.3.	Entrega de documentación	207
6.5.4.	Bibliografía	207
6.6.	Práctica: Servicio de archivos remotos utilizando sockets	207
6.6.1.	Objetivos de la práctica	207
6.6.2.	Descripción de la funcionalidad que debe desarrollar el alumno	208
6.6.3.	Ejemplo de aplicación cliente	209
6.6.4.	Recomendaciones generales	210
6.6.5.	Entrega de documentación	210
6.6.6.	Bibliografía	210
6.7.	Práctica: Sistema de archivos remoto utilizando RPC	210
6.7.1.	Objetivos	210
6.7.2.	Descripción de la funcionalidad que debe desarrollar el alumno	211
6.7.3.	Ejemplo de aplicación cliente	211
6.7.4.	Recomendaciones generales	211
6.7.5.	Entrega de documentación	211
6.7.6.	Bibliografía	212
6.8.	Práctica: Servicio de bloques paralelo con tolerancia a fallos RAID5	212
6.8.1.	Objetivos de la práctica	212
6.8.2.	Dispositivos RAID	212
6.8.3.	Descripción de la funcionalidad que debe desarrollar el alumno	214
6.8.4.	Código fuente de apoyo	216
6.8.5.	Recomendaciones generales	216
6.8.6.	Entrega de documentación	217
6.8.7.	Bibliografía	217
6.9.	Práctica: Exclusión mutua utilizando un anillo con paso de testigo	217
6.9.1.	Objetivos de la práctica	217
6.9.2.	Exclusión mutua basada en paso de testigo	217
6.9.3.	Descripción de la funcionalidad que debe desarrollar el alumno	218
6.9.4.	Recomendaciones generales	220
6.9.5.	Entrega de documentación	220
6.9.6.	Bibliografía	220
A.	Guía del lenguaje de programación en C	221
A.1.	Características de C	222
A.2.	Primer programa en C	222
A.2.1.	¿Cómo compilar y ejecutar el programa?	223

A.3.	Características básicas de C	224
A.3.1.	Comentarios	224
A.3.2.	Palabras reservadas	224
A.3.3.	Tipos de datos elementales	224
A.3.4.	Constantes	225
A.3.5.	La función printf	225
A.3.6.	Variábles	226
A.3.7.	Expresiones y sentencias	227
A.3.8.	Operador de asignación	228
A.3.9.	Función scanf()	228
A.3.10.	Introducción a la directiva #define	229
A.4.	Operadores y expresiones	229
A.4.1.	Operadores aritméticos	230
A.4.2.	Operadores relacionales y lógicos	232
A.4.3.	Operadores lógicos	232
A.4.4.	Resumen de prioridades	234
A.4.5.	Operadores de asignación	234
A.4.6.	Operador condicional	235
A.5.	Sentencias de control	235
A.5.1.	Sentencia if	235
A.5.2.	Sentencia if-else	236
A.5.3.	Sentencia for	237
A.5.4.	Sentencia while	238
A.5.5.	Sentencia do-while	240
A.5.6.	Sentencia switch	241
A.5.7.	Bucles anidados	241
A.5.8.	Sentencias break y continue	242
A.6.	Funciones	243
A.6.1.	Definición de una función	244
A.6.2.	Declaración de funciones: prototipos	244
A.6.3.	Llamadas a funciones	245
A.6.4.	Recursividad	245
A.6.5.	Macros	246
A.7.	Punteros	247
A.7.1.	Definición de punteros	248
A.7.2.	Paso de punteros a una función	249
A.7.3.	El operador sizeof	250
A.8.	Ámbito de las variables y tipos de almacenamiento	251
A.8.1.	Variábles locales	251
A.8.2.	Variábles globales	252
A.8.3.	Variábles automáticas	252
A.8.4.	Variábles estáticas	253
A.8.5.	Variábles de tipo registro	253
A.8.6.	Variábles externas	253
A.9.	Cadenas de caracteres	254
A.9.1.	Definición de cadenas de caracteres	254
A.9.2.	Asignación de valores a cadenas de caracteres	255
A.9.3.	Lectura y escritura de cadenas de caracteres	256
A.9.4.	Paso de cadenas de caracteres a funciones	256

A.9.5.	Funciones de biblioteca para manejar cadenas	257
A.10.	Vectores y matrices	257
	A.10.1. Definición de un vector	258
	A.10.2. Procesamiento de un vector	258
	A.10.3. Paso de vectores a funciones	259
	A.10.4. Punteros y vectores	259
	A.10.5. Vectores y cadenas de caracteres	260
	A.10.6. Vectores multidimensionales	260
A.11.	Estructuras	261
	A.11.1. Procesamiento de una estructura	261
	A.11.2. Paso de estructuras a funciones	262
	A.11.3. Punteros a estructuras	262
	A.11.4. Vectores de estructuras	263
	A.11.5. Uniones	264
	A.11.6. Tipos enumerados	264
	A.11.7. Definición de tipos de datos (<code>typedef</code>)	265
A.12.	Entrada/salida	265
	A.12.1. Apertura y cierre de un archivo	265
	A.12.2. Lectura y escritura	266
A.13.	Aspectos avanzados	266
	A.13.1. Argumentos en la línea de mandatos	267
	A.13.2. Operadores de bits	268
	A.13.3. Máscaras	269
	A.13.4. Operadores de desplazamiento	269
	A.13.5. Campos de bits	270
	A.13.6. Punteros a funciones	270
	A.13.7. Funciones como argumentos	271
	A.13.8. Funciones con número variable de argumentos	272
	A.13.9. Compilación condicional	273
B.	Programación con lenguajes de script	275
B.1.	El <code>shell</code> de UNIX	276
B.2.	Estructura de los mandatos	276
B.3.	Agrupamiento de mandatos	277
	B.3.1. Lista con tuberías (<code>pipes</code>)	277
	B.3.2. Lista O-lógico (OR)	277
	B.3.3. Lista Y-lógico (AND)	278
	B.3.4. Lista secuencial	278
	B.3.5. Lista asíncrona (<code>background</code>)	278
B.4.	Mandatos compuestos y funciones	279
	B.4.1. El mandato condicional <code>if</code>	279
	B.4.2. El mandato condicional <code>case</code>	280
	B.4.3. El bucle <code>until</code>	280
	B.4.4. El bucle <code>while</code>	280
	B.4.5. El bucle <code>for</code>	280
	B.4.6. Funciones	281
B.5.	Redirecciones	282
	B.5.1. Redirección de salida	282
	B.5.2. Redirección de entrada	283

B.6.	<i>Quoting</i>	284
B.7.	Expansión de argumentos	284
B.7.1.	Expansión de tilde	284
B.7.2.	Expansión de variables	284
B.7.3.	Sustitución de mandatos	285
B.7.4.	Expansión aritmética	285
B.7.5.	Expansión de nombres de archivos	286
B.8.	Parámetros	286
B.8.1.	Parámetros posicionales	286
B.8.2.	Parámetros especiales	287
B.8.3.	Variabes	287
B.9.	<i>Shell scripts</i>	287
B.10.	Ejecución de un mandato	291
B.11.	Mandatos de UNIX	292
B.12.	Mandatos internos	292
B.13.	Mandatos externos	293
C.	Entorno de programación de sistemas operativos	295
C.1.	Introducción	296
C.2.	<i>Makefiles</i> de UNIX	296
C.2.1.	Estructura de un archivo <i>makefile</i>	296
C.2.2.	Gestor de bibliotecas	300
C.2.4.	Depuración de una aplicación en UNIX o Linux	301
C.3.	Entorno de programación del Visual C++ de Microsoft	302
C.3.1.	Creación de un espacio de trabajo	302
C.3.2.	Ejecución de una aplicación en Visual C++	305
C.3.3.	Depuración de una aplicación en Visual C++	306

Prólogo

Este libro está pensado como un texto general de prácticas de las asignaturas Sistemas Operativos y Diseño de Sistemas Operativos, pudiendo cubrir tanto la parte introductoria de los aspectos de programación de sistemas como aspectos avanzados de programación y diseño de sistemas operativos (programación de *shell scripts*, programación con llamadas al sistema, programación de módulos del sistema operativo, etc.). No se pretende llevar a cabo una presentación rigurosa y concisa de la teoría de sistemas operativos, para la que se puede recurrir a libros de teoría de la materia, como el escrito por los autores, sino complementar estos libros con prácticas resueltas y proyectos de prácticas que muestren el uso de los sistemas operativos y la programación de aplicaciones con llamadas al sistema sobre distintos sistemas operativos (Linux, UNIX, Windows...).

Obviamente, este libro resulta un complemento natural al libro *Sistemas operativos: una visión aplicada*, escrito por algunos de los autores de esta propuesta y ya publicado por McGraw-Hill. Si bien se puede usar de forma autónoma, complementándolo con cualquier otro libro de teoría de sistemas operativos.

MOTIVACIÓN

La **motivación** para llevar a cabo este trabajo surge de la inexistencia, en inglés o en español, de un libro que contenga un conjunto de prácticas de sistemas operativos resueltas y bien explicadas y que cubran todo el temario de teoría clásica de la materia, así como el de diseño de sistemas operativos. La mayoría de los libros de sistemas operativos existentes se limitan a describir la teoría sin desarrollar, en muchos casos, ejemplos prácticos de los casos estudiados y sin mostrar aspectos avanzados de los sistemas operativos. Por ello, en nuestra opinión, es necesario acudir al desarrollo de prácticas propias que cubran aspectos parciales de la teoría usando a veces versiones específicas, o reducidas, de los sistemas operativos. Esto supone un esfuerzo de cientos de horas de trabajo, por lo que suele resultar demasiado costoso si se quieren hacer prácticas realistas. La reducción de este esfuerzo, por cuestiones que no vienen al caso, suele dar como resultado proyectos prácticos que carecen de los aspectos pedagógicos adecuados para dichos cursos.

Existen algunos libros, escritos en inglés, que proponen proyectos prácticos de sistemas operativos. El libro *Operating Systems Projects for Windows NT*, de Gary Nutt, incluye prácticas sólo para Windows. El libro *Kernel Projects for Linux*, de Gary Nutt, tiene prácticas para Linux. Sin embargo, casi todos ellos tienen el problema de no cubrir todo el programa docente. Además, se ciñen a un sistema concreto y no proporcionan las soluciones de las prácticas, que es lo que realmente quieren los profesores que deben impartir las asignaturas. Frente a los anteriores, nues-

tro libro se distingue por su afán de enfatizar y cubrir todos los aspectos del sistema operativo reflejándolos con prácticas adecuadas. Además, se pretende hacer una descripción pedagógica de las prácticas usando para ello fichas que permitirán conocer su complejidad, objetivos, tiempo de realización estimado, etc.

Este libro incluye básicamente tres aspectos novedosos frente a los libros citados antes:

1. Una panoplia de prácticas que cubren todos los temas clásicos de la teoría. Se proponen varias prácticas por tema, lo que permitirá a los profesores cambiar o elegir las prácticas de forma cíclica.
2. Proyectos completos de prácticas, incluyendo enunciados, soluciones, etc.
3. Descripciones de las herramientas a usar para resolver las prácticas y enlaces web a los orígenes de las mismas.

En resumen, creemos que este libro está justificado porque tiene cosas distintivas y novedosas frente a los libros de sistemas operativos existentes actualmente. Comparado con otros libros, el nuestro tiene la ventaja de cubrir un espectro más amplio de alumnos y profesores, ya que cubre todos los temas de nivel introductorio, medio y de diseño de sistemas operativos.

Aplicación docente

Los sistemas operativos son una materia troncal en los planes de estudio de Informática, por lo que todos ellos incluyen uno o más cursos donde se estudian. La mayoría de libros del área usados en estos cursos incluye información teórica, pero muy pocos añaden aspectos prácticos. Sin embargo, en los planes de estudio se suele dedicar al menos un 35 por 100 a prácticas de laboratorio, cuya preparación y corrección cuesta a los profesores cientos de horas de preparación y cuya realización cuesta a cada alumno decenas de horas de trabajo, a veces usando una documentación deficiente.

Algunas de las asignaturas en que se puede usar este libro son:

- Introducción a los sistemas operativos.
- Sistemas operativos.
- Diseño de sistemas operativos.
- Laboratorio de sistemas operativos.
- Sistemas tolerantes a fallos.
- Redes de comunicaciones.

Además de los aspectos anteriores, más relacionados con la actividad de los profesores, este libro se puede usar en el ámbito de la educación a distancia y la autoenseñanza porque permite a los alumnos desarrollar las prácticas de forma muy autónoma y porque presenta una visión integral del sistema operativo desde la instalación hasta el desarrollo. Este libro está pensado como un libro de referencia para los alumnos, incluso en su futura vida profesional, puesto que cubre aspectos no tratados en los libros generales de sistemas operativos.

Aplicación profesional

Aunque más pensado para el mercado docente, este libro se puede aplicar muy bien al mercado profesional. Debido a la expansión de las computadoras, su uso actual se ha ampliado a todos los campos profesionales, desde ingeniería a sistemas de información, por lo que en muchos casos hay usuarios con poca formación que necesitan desarrollar aplicaciones sobre los sistemas opera-

tivos. Este libro, aun sin ser un vademécum de la materia, presenta casos concretos resueltos, por lo que se puede ajustar bien a los siguientes campos profesionales:

- Aplicaciones de sistemas que necesiten desarrollar códigos de bajo nivel con interfaz al *hardware* o a otros lenguajes, como ensamblador, FORTRAN o C++.
- Administración de sistemas operativos y de redes de computadoras.
- Monitorización de sistemas operativos y de redes de computadoras.
- Aplicaciones cliente-servidor en sistemas distribuidos.
- Sistemas distribuidos y de comunicaciones, como protocolos de comunicaciones.
- Sistemas de almacenamiento en archivos.
- Sistemas de tiempo real y tolerantes a fallos.
- Desarrollo de aplicaciones en *clusters* de computadoras.

Contenidos

El libro se estructura en seis capítulos, en cada uno de los cuales se presentan varios proyectos prácticos.

En cada capítulo se muestran:

- Los conceptos básicos necesarios para realizar las prácticas.
- Ejemplos de uso de los servicios y llamadas al sistema a utilizar.
- Enunciados de los proyectos prácticos.
- Material de apoyo que se proporciona para realizar las prácticas.
- Información sobre las herramientas necesarias para realizar las prácticas.
- Información complementaria para este tema y los proyectos descritos.

Las prácticas propuestas en el libro se dividen en tres niveles: introducción, avanzado y diseño. Esta graduación permite conocer qué prácticas son más adecuadas para cada tipo de curso.

En los apéndices se muestra una guía de referencia del lenguaje C, un apéndice sobre programación de *shell scripts* y el entorno de desarrollo del lenguaje en los sistemas operativos Linux y Windows.

En concreto, el libro incluye los siguientes capítulos:

1. Introducción a los sistemas operativos.
2. Procesos.
3. Gestión de memoria.
4. Comunicación y sincronización de procesos.
5. Entrada/salida, archivos y directorios.
6. Introducción a los sistemas distribuidos.

Página web

Este libro tiene asociada la página web:

<http://www.arcos.inf.uc3m.es/~ssoo-va/>

A través de esta página web se puede acceder a los materiales complementarios del libro, como:

- Material de ayuda para la realización de las prácticas.
- Más prácticas propuestas.
- Enlaces a compiladores gratuitos del lenguaje C.
- Etc.

Además, dicha página incluye una sección dedicada a los profesores que vayan a usar el libro con acceso restringido a los mismos mediante registro previo. Esta sección incluye:

- Una guía para el profesor con los métodos de solución de prácticas.
- Soluciones para las prácticas propuestas.
- Material de apoyo para la realización de las prácticas.
- Código fuente completo de las soluciones de las prácticas.
- Etc.

Jesús Carretero

Félix García
Departamento de Informática
Escuela Politécnica Superior
Universidad Carlos III de Madrid
Madrid, España

Fernando Pérez

Departamento de Arquitectura y Tecnología
de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid
Madrid, España

Este libro es el resultado de la experiencia docente en la enseñanza de la programación en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid y en la Facultad de Informática de la Universidad Politécnica de Madrid. Ambas universidades tienen una larga tradición en la formación de profesionales en informática y en la investigación en el campo de la programación. Los autores han desarrollado durante muchos años un currículum basado en la programación en C, que ha sido muy bien recibido por los estudiantes y profesores. El éxito de este currículum ha llevado a la creación de este libro, que pretende ser una herramienta útil para la enseñanza y aprendizaje de la programación en C.

Este libro se divide en tres partes principales: una introducción a la programación en C, un análisis de los fundamentos de la programación en C y una sección de prácticas. La introducción a la programación en C cubre los aspectos básicos de la programación en C, como la sintaxis, los tipos de datos y las estructuras de control. El análisis de los fundamentos de la programación en C cubre los aspectos más avanzados de la programación en C, como la memoria, la ejecución y la optimización. La sección de prácticas incluye una serie de ejercicios y problemas que permiten al lector practicar lo aprendido y aplicarlo a situaciones reales.

Este libro es destinado a los estudiantes de ingeniería de informática y a los profesionales que trabajan en el campo de la programación en C. Es un libro que combina teoría y práctica, y que pretende ser una herramienta útil para la enseñanza y aprendizaje de la programación en C. Los autores esperan que este libro sea de utilidad para todos aquellos que quieran aprender a programar en C y que quieran aplicar sus conocimientos a la resolución de problemas prácticos.

Este libro es el resultado de la experiencia docente en la enseñanza de la programación en C en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid y en la Facultad de Informática de la Universidad Politécnica de Madrid. Ambas universidades tienen una larga tradición en la formación de profesionales en informática y en la investigación en el campo de la programación. Los autores han desarrollado durante muchos años un currículum basado en la programación en C, que ha sido muy bien recibido por los estudiantes y profesores. El éxito de este currículum ha llevado a la creación de este libro, que pretende ser una herramienta útil para la enseñanza y aprendizaje de la programación en C.

Este libro se divide en tres partes principales: una introducción a la programación en C, un análisis de los fundamentos de la programación en C y una sección de prácticas. La introducción a la programación en C cubre los aspectos básicos de la programación en C, como la sintaxis, los tipos de datos y las estructuras de control. El análisis de los fundamentos de la programación en C cubre los aspectos más avanzados de la programación en C, como la memoria, la ejecución y la optimización. La sección de prácticas incluye una serie de ejercicios y problemas que permiten al lector practicar lo aprendido y aplicarlo a situaciones reales.

1

Introducción a los sistemas operativos

En este capítulo de introducción se presentan, en primer lugar, algunos conceptos generales de sistemas operativos de manera que sirvan de base para el resto del libro. A continuación, se plantea una práctica vinculada con la instalación de un sistema operativo, concretamente Linux. Por último, se presenta el minikernel, el entorno de prácticas que se utilizará en el libro para realizar prácticas de diseño de sistemas operativos.

1.1. CONCEPTOS GENERALES SOBRE SISTEMAS OPERATIVOS

En este apartado se presenta una serie de conceptos generales sobre sistemas operativos. En él no se pretende realizar una presentación exhaustiva de los mismos, puesto que este tipo de información se puede encontrar en cualquier libro general de sistemas operativos. El objetivo es recordar algunos conceptos básicos que permitan abordar las distintas prácticas planteadas a lo largo del libro. En primer lugar, se definen las funciones del sistema operativo para, a continuación, pasar a describir los componentes básicos del mismo. Por último, se explica cómo se produce la activación del sistema operativo y, concretamente, cómo se lleva a cabo una llamada al sistema.

1.1.1. Funciones del sistema operativo

Un *sistema operativo* (SO) es un programa que tiene encomendada una serie de diferentes funciones cuyo objetivo es simplificar el manejo y la utilización de la computadora haciéndolo seguro y eficiente. Las funciones clásicas del sistema operativo se pueden agrupar en las tres categorías siguientes:

- Gestión de los recursos de la computadora.
- Ejecución de servicios para los programas.
- Ejecución de los mandatos de los usuarios.

Con respecto a su faceta de gestor de recursos, hay que tener en cuenta que en una computadora actual suelen coexistir varios programas, del mismo o de varios usuarios, ejecutándose simultáneamente. Estos programas compiten por los recursos de la computadora, siendo el sistema operativo el encargado de arbitrar su asignación y uso. Como complemento a la gestión de recursos, el sistema operativo ha de garantizar la protección de unos programas frente a otros y ha de suministrar información sobre el uso que se hace de los recursos.

Por lo que se refiere al sistema operativo como máquina extendida, éste ofrece a los programas un conjunto de servicios o **llamadas al sistema** que pueden solicitar cuando lo necesiten, proporcionando a los programas una visión de máquina extendida. Los servicios se pueden agrupar en las cuatro clases siguientes: ejecución de programas, operaciones de E/S, operaciones sobre archivos y detección y tratamiento de errores.

Por último, el sistema operativo también es un elemento que proporciona la interfaz de usuario del sistema. El módulo del sistema operativo que permite que los usuarios dialoguen de forma interactiva con el sistema es el intérprete de mandatos o *shell*.

1.1.2. Componentes del sistema operativo

Como se muestra en la Figura 1.1, se suele considerar que un sistema operativo está formado por tres capas: el núcleo, los servicios y el intérprete de mandatos o *shell*.

El núcleo es la parte del sistema operativo que interacciona directamente con el *hardware* de la máquina. Las funciones del núcleo se centran en la gestión de recursos, como es el procesador, tratamiento de interrupciones y las funciones básicas de manipulación de memoria.

Los servicios se suelen agrupar según su funcionalidad en varios componentes, cada uno de los cuales se ocupa de las siguientes funciones:

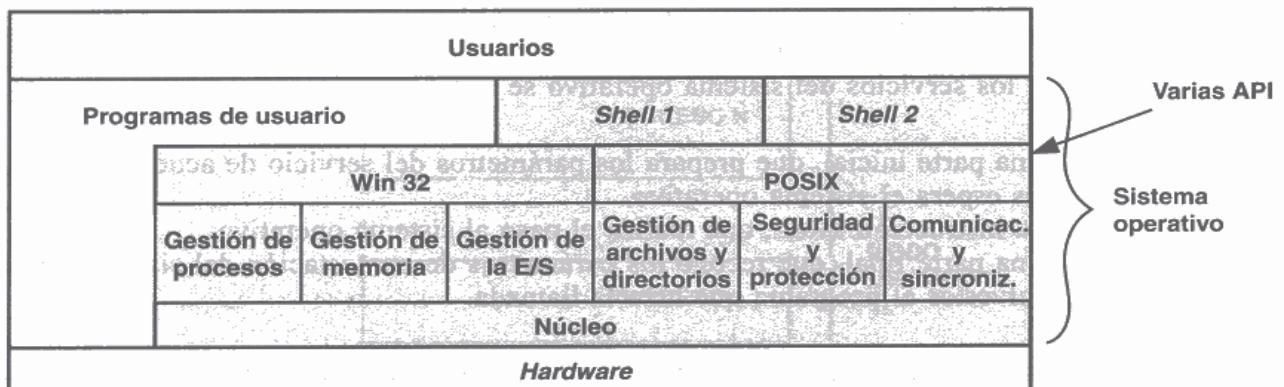


Figura 1.1. Componentes del sistema operativo.

- Gestión de procesos, encargada de la creación, planificación y destrucción de procesos.
- Gestión de memoria, componente encargada de saber qué partes de memoria están libres y cuáles ocupadas, así como de la asignación y liberación de memoria según la necesiten los procesos.
- Gestión de la E/S, se ocupa de facilitar el manejo de los dispositivos periféricos.
- Gestión de archivos y directorios, se encarga del manejo de archivos y directorios y de la administración del almacenamiento secundario.
- Comunicación y sincronización entre procesos, encargada de ofrecer mecanismos para que los procesos puedan comunicarse y sincronizarse.
- Seguridad y protección, este componente debe encargarse de garantizar la identidad de los usuarios y de definir lo que pueden hacer cada uno de ellos con los recursos del sistema.

Por lo que se refiere a la interfaz de usuario, el sistema operativo puede incluir varios intérpretes de mandatos, unos textuales y otros gráficos.

1.1.3. Activación del sistema operativo

Una vez presentadas las funciones y componentes del sistema operativo, es importante describir cuáles son las acciones que activan la ejecución del mismo. El sistema operativo es un servidor que está a la espera de que se le encargue trabajo. El trabajo del sistema operativo puede provenir de las siguientes fuentes:

- Llamadas al sistema emitidas por los programas.
- Interrupciones producidas por los periféricos.
- Condiciones de excepción o error del hardware.

Cuando es un proceso en ejecución el que desea un servicio del sistema operativo ha de utilizar una instrucción TRAP, que genera la pertinente interrupción. En los demás casos, será una interrupción interna (una excepción) o externa (proveniente de un dispositivo de entrada/salida) la que reclame la atención del sistema operativo.

Cuando se programa en un lenguaje de alto nivel, como el C, la solicitud de un servicio del sistema operativo se hace mediante una llamada a una función (por ejemplo, `fork()`, que es el servicio POSIX para la creación de un nuevo proceso). No hay que confundir esta llamada con el

servicio del sistema operativo. La función `fork` del lenguaje C no realiza el servicio `fork`, simplemente se limita a solicitar este servicio del sistema operativo. En general, estas funciones que solicitan los servicios del sistema operativo se componen de:

- Una parte inicial, que prepara los parámetros del servicio de acuerdo con la forma en que los espera el sistema operativo.
- La instrucción TRAP, que realiza el paso al sistema operativo.
- Una parte final, que recoge los parámetros de contestación del sistema operativo para devolverlos al programa que hizo la llamada.

Todo el conjunto de estas funciones se encuentran en una biblioteca del sistema y se incluyen en el código en el momento de su carga en memoria. Para completar la imagen de que se está llamando a una función, el sistema operativo devuelve un valor como una función real. Al programador le parece, por tanto, que invoca al sistema operativo como a una función. Sin embargo, esto no es así, puesto que lo que hace es invocar una función que realiza la solicitud al sistema operativo. El siguiente código muestra una hipotética implementación de la llamada al sistema `fork`.

```
int fork() {
    int r;
    LOAD R8, FORK_SYSTEM_CALL
    TRAP
    LOAD r, R9
    return(r);
}
```

El código anterior carga en uno de los registros de la computadora (el registro R8, por ejemplo) el número que identifica la llamada al sistema (en este caso, `FORK_SYSTEM_CALL`). En el caso de que la llamada incluyera parámetros, éstos se pasarían en otros registros o en la pila. A continuación, la función de biblioteca ejecuta la instrucción TRAP, con lo que se transfiere el control al sistema operativo, que accede al contenido del registro R8 para identificar la llamada a ejecutar y realizar el trabajo. Cuando el control se transfiere de nuevo al proceso que invocó la llamada `fork`, se accede al registro R9 para obtener el valor devuelto por la llamada y éste se retorna finalizando así la ejecución de la función de biblioteca.

La Figura 1.2 muestra todos los pasos involucrados en una llamada al sistema operativo indicando el código que interviene en cada uno de ellos.

Por ejemplo, si el programa quiere escribir datos en un archivo, el código del programa de usuario hace un CALL a la rutina en código máquina `WRITE`, con código similar al mostrado anteriormente para la llamada `fork`. Esta rutina prepara los parámetros de la operación de escritura y ejecuta la instrucción TRAP. El sistema operativo trata la interrupción, identifica que se trata de una solicitud de servicio y que el servicio solicitado es la escritura en un archivo. Seguidamente, ejecuta la rutina que lanza la pertinente operación de E/S. Por último, ejecuta el planificador y da paso a la ejecución de otro proceso.

Cuando el periférico termina la operación, su controlador genera una solicitud de interrupción que es tratada por el sistema operativo. Como resultado de la misma, el proceso anterior pasará a estar listo para su ejecución. En su momento, se seleccionará este proceso para que execute. En ese instante la rutina en código máquina `WRITE` recoge los parámetros que ha devuelto el sistema operativo y ejecuta un RET para volver al programa de usuario que la llamó.

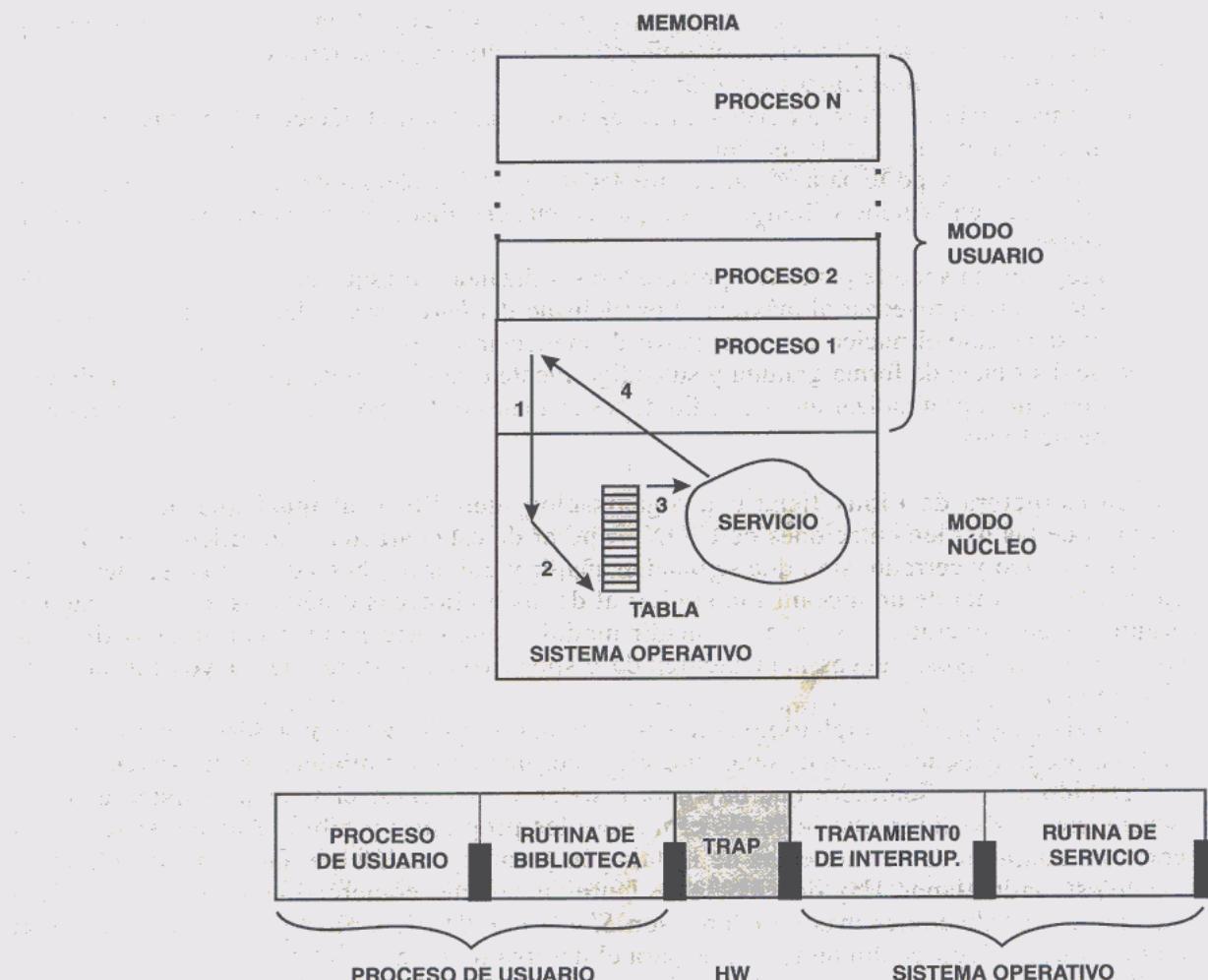


Figura 1.2. Pasos de la llamada al sistema operativo.

1.2. PRÁCTICA: INSTALACIÓN DEL SISTEMA OPERATIVO LINUX

El objetivo de esta práctica es doble. Por una parte, se pretende que el alumno aprenda a instalar un sistema operativo, de forma que entienda qué pasos hay que seguir a la hora de llevar a cabo dicha instalación. Por otro lado, el sistema operativo a instalar (Linux) se utilizará en numerosas prácticas descritas a lo largo del libro; por tanto, el alumno podrá disponer de una plataforma de trabajo desde el primer momento.

Linux es un sistema UNIX y, por tanto, posee las características típicas de los sistemas UNIX. Se trata de un sistema multiusuario y multitarea de propósito general. Algunas de sus características específicas más relevantes son las siguientes:

- Proporciona una interfaz POSIX.
- Tiene un código independiente del procesador en la medida de lo posible. Aunque inicialmente se desarrolló para procesadores Intel, se ha transportado a otras arquitecturas con un esfuerzo relativamente pequeño.

- Puede adaptarse a máquinas de muy diversas características. Como el desarrollo inicial se realizó en máquinas con recursos limitados, ha resultado un sistema que puede trabajar en máquinas con prestaciones muy diferentes.
- Permite incluir de forma dinámica nuevas funcionalidades al núcleo del sistema operativo gracias al mecanismo de módulos.
- Proporciona soporte para una gran variedad de tipos de sistemas de archivos, entre ellos los utilizados en Windows. También es capaz de manejar distintos formatos de archivos ejecutables.
- Proporciona soporte para multiprocesadores utilizando un esquema de multiproceso simétrico. Para aprovechar al máximo el paralelismo del *hardware*, se ha ido modificando progresivamente el núcleo con el objetivo de aumentar su concurrencia interna.
- Se distribuye de forma gratuita y su código fuente está disponible. Esto permite al alumno que quiera profundizar en el estudio de los sistemas operativos analizar la estructura interna de Linux.

La estructura de Linux tiene una organización monolítica, al igual que ocurre con la mayoría de las implementaciones de UNIX. A pesar de este carácter monolítico, el núcleo no es algo estático y cerrado, sino que se pueden añadir y quitar módulos de código en tiempo de ejecución. Se trata de un mecanismo similar al de las bibliotecas dinámicas, pero aplicado al propio sistema operativo. Se pueden añadir módulos que correspondan con nuevos tipos de sistemas de archivos, nuevos manejadores de dispositivos o gestores de nuevos formatos de ejecutables.

Un sistema Linux completo no sólo está formado por el núcleo monolítico, sino también incluye programas del sistema como, por ejemplo, demonios y bibliotecas del sistema.

Debido a las dificultades que hay para instalar y configurar el sistema, existen diversas distribuciones de Linux que incluyen el núcleo, los programas y bibliotecas del sistema, así como un conjunto de herramientas de instalación y configuración que facilitan considerablemente esta ardua labor. Hay distribuciones tanto de carácter comercial como gratuitas. Algunas de las distribuciones más populares son *Slackware*, *Debian*, *Suse* y *Red Hat*. Aunque el alumno puede elegir cualquiera de ellas para el desarrollo de las prácticas que se describen en el libro, en este apartado se va a describir los pasos básicos a seguir para instalar la distribución *Red Hat*.

1.2.1. Pasos a seguir para la instalación de la distribución *Red Hat*

Red Hat es una distribución muy utilizada y fácil de instalar. En la página web <http://www.redhat.com> se puede encontrar toda la información relativa a dicha distribución. Aunque existe la posibilidad de instalar esta distribución desde un servidor ftp, en esta sección sólo se van a describir los pasos básicos a seguir para instalar la distribución desde CDROM. Antes de comenzar el proceso de instalación, es fundamental que conozca la configuración de su computadora (monitor, teclado, discos, información relacionada con la red, etc.).

1. Inserte el CD con la distribución de *Red Hat* en su unidad de CDROM y arranque su computadora desde el CDROM.
2. Seleccione el lenguaje a utilizar durante el proceso de instalación, le aparecerá una pantalla como la que se muestra en la Figura 1.3.
3. Elija la configuración de su teclado, es decir, modelo y tipo de teclado (véase la Figura 1.4).

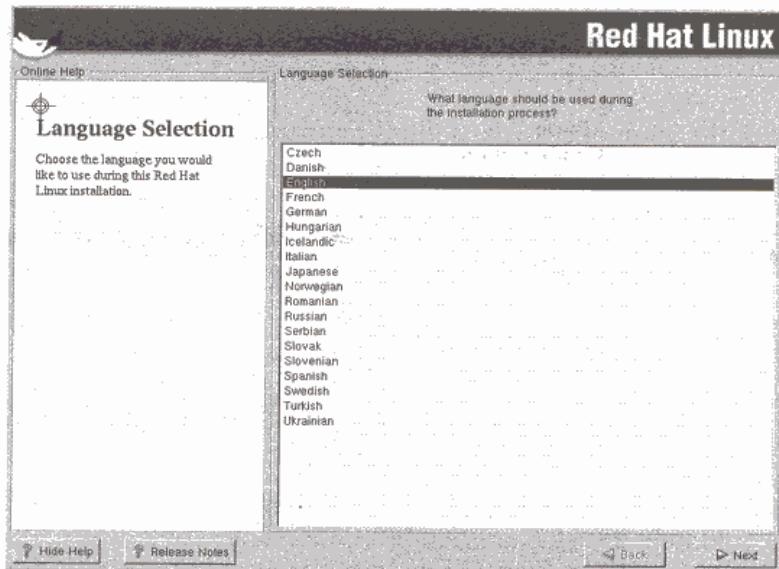


Figura 1.3. Selección del lenguaje de instalación.

4. Seleccione el tipo de ratón que posee su computadora. Para determinar el tipo de ratón que tiene (serie o PS/2) basta con que compruebe cómo es el conector del ratón. Si es redondo, su ratón es de tipo PS/2; si es rectangular, es un ratón serie (véase la Figura 1.5).
5. Seleccione el tipo de sistema que desea (véase la Figura 1.6). Existen cinco posibilidades:
 - *Workstation*. Es la más adecuada para una instalación típica, además de ser la más sencilla.
 - *Server*. Ésta es la instalación adecuada si se pretende que la computadora funcione como un servidor.

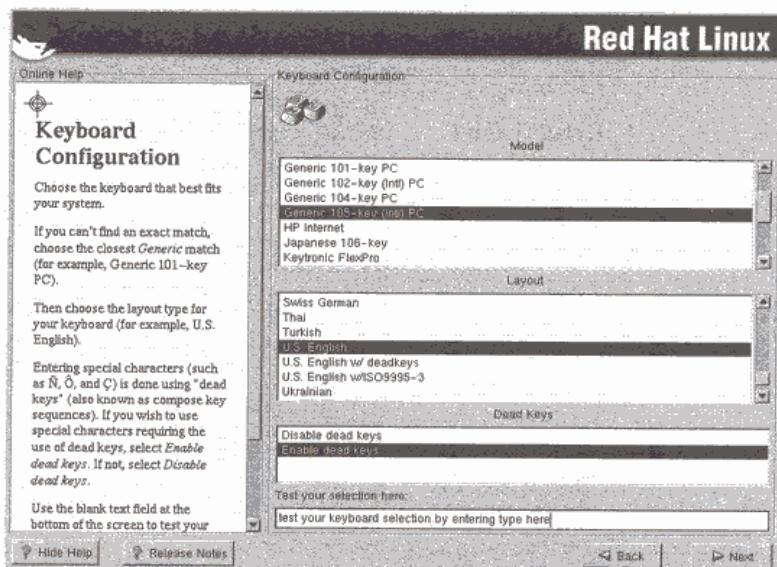


Figura 1.4. Configuración del teclado.

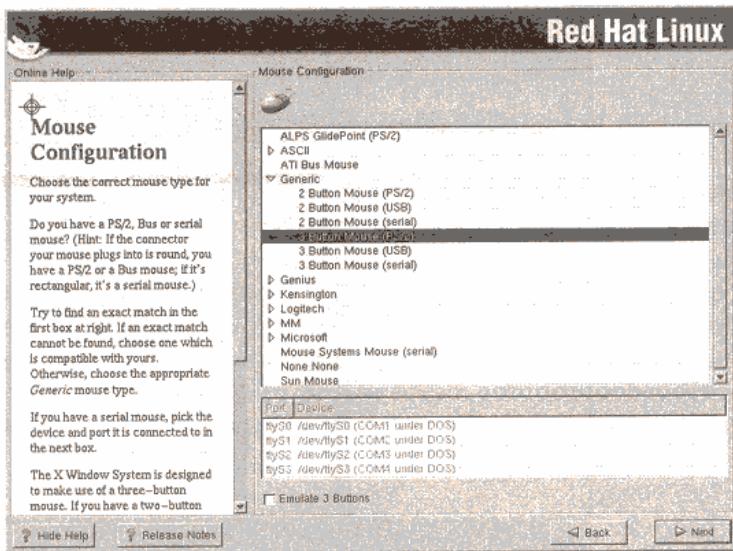


Figura 1.5. Configuración del ratón.

- *Laptop*. Instalación apropiada para instalar Linux en una computadora portátil.
- *Custom*. Esta opción es la que ofrece una mayor flexibilidad durante el proceso de instalación, ya que permite elegir qué paquetes instalar. Sin embargo, requiere un mayor conocimiento y no es recomendada para aquellos lectores que instalan Linux por primera vez.
- *Upgrade*. Con esta opción se indica que se quiere actualizar alguna versión anterior ya instalada en el sistema.

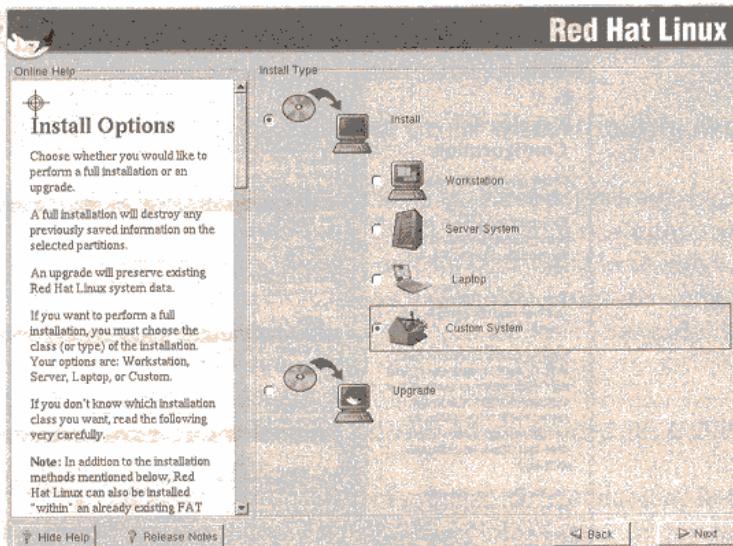


Figura 1.6. Tipo de sistema a instalar.

6. A continuación se debe particionar el disco duro donde se va a instalar Linux. Una buena opción en este punto es elegir que el particionamiento del disco se realice de forma automática (véase la Figura 1.7).
7. Por último, deberá configurar su red indicando la siguiente información: dirección IP, máscara de red, nombre del *host*, pasarela (*gateway*), servidor de DNS primario y secundario (véase la Figura 1.8).
8. Los últimos pasos a seguir consisten en la elección de la zona horaria, el idioma de soporte y la contraseña del superusuario (*root*).

1.3. EL MINIKERNEL: UN ENTORNO PARA EL DESARROLLO DE PRÁCTICAS DE DISEÑO DE SISTEMAS OPERATIVOS

El principal problema que surge cuando se pretenden realizar prácticas en asignaturas que estudian los aspectos internos de los sistemas operativos (como, por ejemplo, «Diseño de sistemas operativos») es decidir en qué entorno se llevan a cabo. La elección más obvia es utilizar un sistema operativo real del que se disponga su código fuente (tales como MINIX o Linux) y realizar modificaciones en el mismo para incluir nuevas funcionalidades. Este enfoque es el más realista, pero, sin embargo, presenta algunos problemas que dificultan considerablemente su aplicación práctica:

- Los inconvenientes que encuentra el alumno al tener que trabajar directamente sobre la «máquina desnuda». El alumno no va a poder recurrir a las herramientas habituales de depuración y se va a tener que enfrentar con un tedioso y complejo ciclo de reinicio del equipo cada vez que un error en su práctica cause que el sistema se colapse. Nótese que estos errores incluso podrían afectar al buen funcionamiento del sistema causando que pierda su configuración o incluso datos.

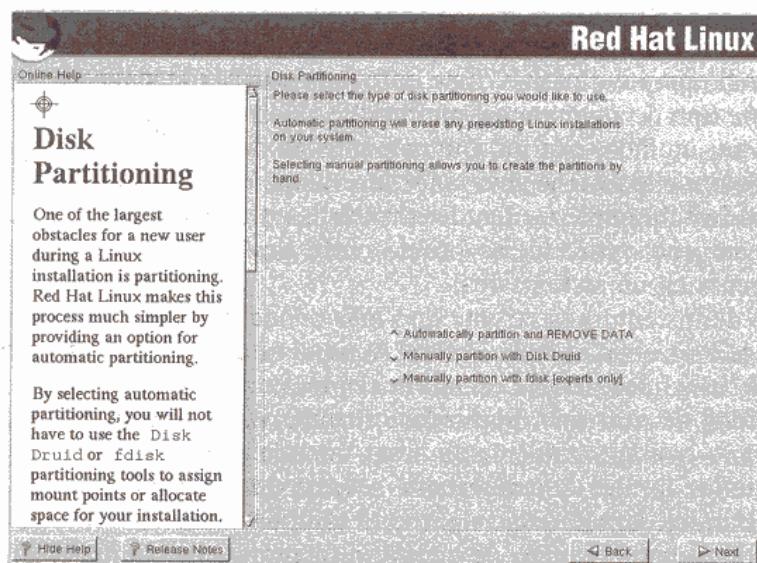


Figura 1.7. Particionamiento del disco.

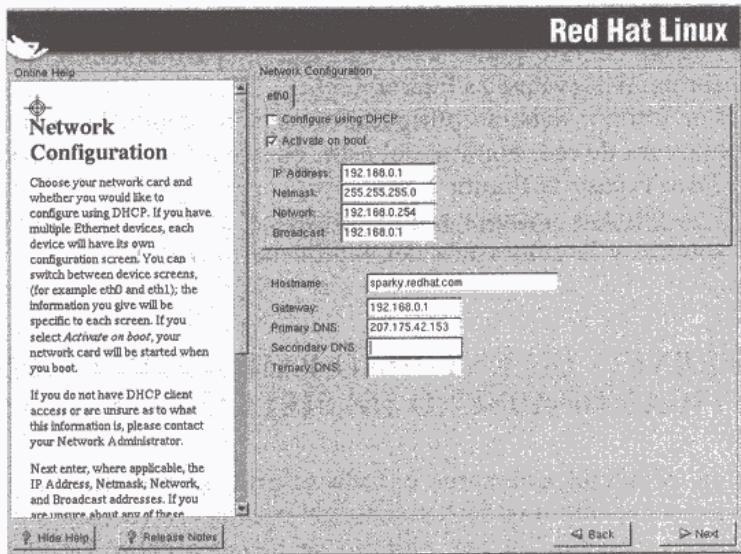


Figura 1.8. Configuración de la red.

- La ejecución directa de la práctica sobre el *hardware* también afecta al instructor, ya que le provoca considerables dificultades a la hora de evaluar la práctica.
- La necesidad de que cada alumno disponga de un equipo para el desarrollo de la práctica, lo que puede resultar inviable en cursos con un número elevado de alumnos.

Adicionalmente, en el caso de sistemas concebidos para su utilización en la vida real, como Linux, el código de los mismos es demasiado complejo debido a las «oscuras» optimizaciones que suelen estar presentes en este tipo de sistemas. En el caso de MINIX, al tratarse de un sistema diseñado con un objetivo pedagógico, su uso directo es más viable. Sin embargo, sigue presentando las deficiencias expuestas anteriormente.

Ante esta situación se han creado algunos entornos que emulan el comportamiento del sistema operativo y del *hardware* subyacente en el contexto de uno o varios procesos convencionales del sistema operativo nativo. Este enfoque es más adecuado para las características de las prácticas de este tipo de asignaturas, puesto que, dado que no se trabaja directamente sobre el *hardware*, permite, por un lado, que los alumnos puedan desarrollar sus programas de una forma convencional, y por otro, facilita al instructor la evaluación de los trabajos prácticos. Asimismo, posibilita que todos los alumnos puedan desarrollar las prácticas en un único equipo. Evidentemente, como en todo entorno emulado, se pierde parte de la experiencia que se podría obtener trabajando directamente en un sistema real.

Uno de los entornos de este tipo más conocidos y utilizados es el sistema NACHOS, desarrollado en la Universidad de Berkeley. Aunque este sistema proporciona un entorno con una serie de características interesantes, en nuestra opinión presenta algunas deficiencias (como, por ejemplo, la necesidad de usar un compilador cruzado o su carácter determinista) que hacen que hayamos considerado que era conveniente crear nuestro propio entorno, denominado *minikernel*.

De todas formas, es importante señalar que consideramos que el uso de un entorno de este tipo no elimina la necesidad de que el alumno entre en contacto con un sistema operativo real, como, por ejemplo, Linux. Al contrario, creemos que se debería fomentar este aspecto haciendo que el alumno estudie algunos fragmentos del código del sistema operativo que se consideren especialmente interesantes y relativamente abarcables (como, por ejemplo, el planificador de pro-

cesos). Asimismo, para aquellos alumnos que estén especialmente interesados o motivados, se deberían plantear prácticas optativas de programación de algún módulo interno de un sistema operativo real (como, por ejemplo, un manejador de un dispositivo de caracteres sencillo en Linux). Sin embargo, nuestra experiencia docente dicta que, dada la complejidad de este tipo de prácticas, no sería adecuado hacer que todos los alumnos las realizaran obligatoriamente.

1.3.1. Características del *minikernel*

El *minikernel* se basa en la idea del *hardware virtual*. En este apartado se explica brevemente este concepto para, de esta forma, comprender mejor las características de este entorno. Esta explicación está destinada principalmente al instructor, de manera que le permita conocer mejor cuál es el fundamento de este entorno de prácticas. Queda a criterio del instructor el pedir al alumno que lea esta sección si considera que puede facilitarle una mejor comprensión del entorno o que la obvie en caso de que considere que su lectura podría ser incluso contraproducente y crea que es preferible que el alumno vea el entorno subyacente como una «caja negra».

Una de las funciones del sistema operativo es crear una abstracción del *hardware* (o sea, «vestir» a la máquina desnuda) ofreciendo una máquina extendida sobre la que ejecutan las aplicaciones. La idea es invertir este proceso, o sea, colocar una capa encima del sistema operativo (a la que podemos denominar *hardware virtual*) que cree una máquina «real» sobre la extendida. Dicho de otra forma, un nivel de *software* que realice un proceso inverso al del sistema operativo, o sea, una «concreción» en lugar de una abstracción.

Veamos, como ejemplo, la interrupción de reloj. Un sistema UNIX la abstrae como la señal SIGALRM. Esta abstracción es más general que la interrupción real, ya que, a partir de un único temporizador real, crea múltiples temporizadores (uno por proceso).

La inversión de la abstracción (la «concreción») consiste en que la capa del *hardware virtual* cree una interrupción de reloj «virtual» a partir de la señal SIGALRM recibida por esta capa. Por tanto, realmente no se trata de una simulación propiamente dicha, ya que no se puede decir que se ha simulado una interrupción de reloj, sino que realmente la ha habido: el dispositivo genera una interrupción de reloj, que provoca una señal SIGALRM hacia el *hardware virtual*, que, a su vez, la transforma en una interrupción hacia el nivel superior.

Esto no se restringe a las interrupciones. Así, por ejemplo, siguiendo con el ejemplo del reloj, el sistema operativo, en su arranque, debe escribir en los registros de entrada/salida del controlador de reloj para establecer su frecuencia de interrupción. La capa de *hardware virtual* exporta una operación similar a la del *hardware* real permitiendo que el *software* que se ejecuta encima de ella (o sea, el sistema operativo que está encima del *hardware virtual*) pueda realizar la misma operación. Esta misma idea se aplica a otros eventos, como, por ejemplo, los siguientes:

- La interrupción del terminal y la lectura del carácter tecleado se «concretan» a partir de SIGIO y la lectura asíncrona de la entrada estándar, respectivamente. Nótese nuevamente el proceso de abstracción y concreción: el controlador del teclado genera una interrupción que el sistema operativo nativo abstrae en la señal SIGIO, que, a su vez, la capa de *hardware virtual* concreta en una interrupción virtual de teclado.
- Las excepciones se generan a partir de las señales correspondientes (por ejemplo, a partir de la señal SIGFPE se produce la excepción aritmética).

La Figura 1.9 intenta ilustrar este proceso de abstracción y concreción.

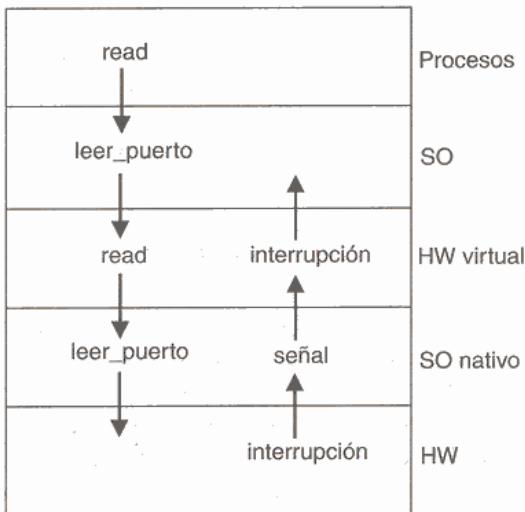


Figura 1.9. Proceso de abstracción y concreción usado en el «hardware virtual».

Es importante resaltar los distintos niveles de ejecución que conviven al trabajar en este entorno, que, como se puede apreciar en la figura anterior, se corresponden con los siguientes:

- Los programas de usuario del *minikernel*.
- El sistema operativo del *minikernel*.
- El *hardware virtual*.
- El sistema operativo nativo.
- El *hardware real*.

El *minikernel* está, por tanto, organizado como dos subsistemas independientes:

- La capa que implementa el *hardware virtual* y que, en principio, queda oculta al alumno, que sólo conoce su interfaz. Este subsistema ofrece unos servicios similares a los proporcionados por el módulo HAL (*Hardware Abstraction Layer*, nivel de abstracción del *hardware*) presente en muchos sistemas operativos reales.
- La capa del sistema operativo propiamente dicha. Éste es el nivel donde trabajará el alumno para incluir las diversas funcionalidades que se le vayan requiriendo.

En este mismo concepto de *hardware virtual* se basan otros sistemas que no tienen un perfil pedagógico, sino experimental (como vChoices) o comercial (como VMware).

Esta separación nítida entre el nivel del *hardware* y del *software* que existe en el *minikernel* proporciona al alumno una visión más realista frente a otros entornos, como MINIX/Solaris (un entorno donde se ha implementado MINIX dentro del ámbito de un proceso Solaris), en los que no existe tan clara distinción entre niveles. Además, permite al alumno trabajar directamente sobre el *hardware* para desarrollar desde cero las funcionalidades del sistema operativo que se le pidan.

A continuación se presentan algunas características adicionales de este entorno:

- La capa de *hardware virtual* se implementa usando los servicios POSIX del sistema operativo subyacente, no haciendo uso en ningún momento de código ensamblador. Actualmen-

te, está transportada a Linux y Digital UNIX, pero puede adaptarse a otros sistemas UNIX que cumplan el estándar POSIX.

- No requiere el uso de un compilador cruzado. Tanto los programas de usuario como el código del sistema operativo se compilan usando el compilador nativo de la máquina. Los programas de usuario son programas convencionales con la única peculiaridad de que usan los servicios del *minikernel* en vez de los de UNIX. Para ser preciso, hay que explicar que realmente tanto el sistema operativo como los programas de usuario se generan como bibliotecas dinámicas en vez de como ejecutables, pero se trata de un aspecto de implementación interna que no afecta a los usuarios del entorno.
- Asimismo, tanto los programas de usuario como el código del sistema operativo ejecutan sobre el procesador nativo. A diferencia de otros entornos, como NACHOS, donde los procesos de usuario ejecutan sobre la simulación de un procesador (un MIPS en el caso de NACHOS).
- Aunque no hay protección real entre el sistema operativo y los procesos de usuario ni de éstos entre sí, ya que todos ejecutan en el entorno del mismo proceso, sí existe protección en cuanto a la visibilidad de los símbolos. Así, un proceso de usuario no puede acceder directamente a símbolos definidos en el sistema operativo ni a la inversa. Se podría decir que se trata de módulos desacoplados entre sí.
- En la versión actual sólo se incluyen dos dispositivos de entrada/salida: el reloj y el terminal.

Con respecto a este último punto, hay que hacer notar que, evidentemente, aplicando de nuevo la idea del *hardware virtual*, se podría dotar al *minikernel* de más dispositivos de entrada/salida (así, por ejemplo, a partir de un archivo y de un *socket* del sistema operativo nativo, se podrían crear un disco y una conexión de red «virtuales», respectivamente). Aunque se han hecho algunos trabajos en esa línea, la experiencia docente nos ha mostrado que este tipo de dispositivos más sofisticados conlleva desarrollar un *software* mucho más complejo (piense, por ejemplo, en un disco que requiere tanto el manejador como el propio sistema de archivos) que hace que tenga más sentido usar otro tipo de entorno de prácticas diseñado específicamente para ese tipo de dispositivo específico.

Por todo ello, este entorno se ha mostrado especialmente efectivo para el desarrollo de prácticas vinculadas con la gestión de procesos y de la entrada/salida de bajo nivel vinculada con dispositivos sencillos, como son el reloj y el terminal. Concretamente, este entorno permite al alumno ver de una forma muy práctica en qué consiste realmente la multiprogramación, puesto que él mismo tiene que programar esa «magia» que hace que se ejecuten concurrentemente varios procesos en un único procesador siguiendo el ritmo que marcan las interrupciones del sistema. Por tanto, este entorno se usará en las prácticas de diseño de los capítulos de procesos, comunicación de procesos y entrada/salida. Sin embargo, se usarán entornos específicos para las prácticas de diseño de los capítulos de memoria, archivos y sistemas distribuidos.

Hay que resaltar que la multiprogramación y, en general, la concurrencia son probablemente los temas más importantes en la enseñanza de los sistemas operativos, aunque también son los más complejos de entender. Es muy difícil conseguir comprender lo que ocurre cuando se están ejecutando concurrentemente varias actividades. Esta dificultad se acentúa de manera notable cuando se está trabajando en el nivel más bajo del sistema operativo. Por un lado, en este nivel, la mayoría de los eventos son asíncronos. Por otro, en él existe una gran dificultad para la depuración dado el carácter no determinista del sistema y la falta de herramientas de depuración adecuadas.

Este entorno de prácticas pretende precisamente enfrentar al alumno con toda esta problemática. Es importante resaltar desde el principio las dificultades que se encontrará para la realización de las prácticas en este entorno. Sin embargo, aunque parezca un poco sorprendente, enfrentarse con esos problemas es, a su vez, un objetivo de las mismas.

Como resumen, hay que resaltar que en las prácticas que se desarrollarán en este entorno se van plasmar muchos de los conceptos estudiados en la teoría de sistemas operativos, tales como:

- El arranque del sistema operativo.
- El manejo de las interrupciones.
- El manejo de las excepciones.
- El manejo de las llamadas al sistema.
- La multiprogramación.
- El cambio de contexto.
- La planificación de procesos.
- La sincronización entre procesos.
- Los manejadores de entrada/salida.

1.3.2. Descripción del entorno de desarrollo

El entorno de desarrollo de la práctica intenta imitar dentro de lo que cabe el comportamiento y estructura de un sistema real. En una primera aproximación, en este entorno se pueden diferenciar tres componentes principales (que se corresponden con los tres subdirectorios que presenta la jerarquía de archivos del entorno):

- El programa cargador del sistema operativo (directorio `boot`).
- El entorno propiamente dicho (directorio `minikernel`), donde se incluye tanto el sistema operativo (módulo `kernel`) como el *hardware virtual* subyacente (módulo `HAL`).
- Los programas de usuario (directorio `usuario`).

Aunque la decisión final queda en manos del instructor para no causar cierta confusión al alumno, se recomienda no entregarle el código fuente del programa cargador (programa `boot`) ni de la capa de *hardware virtual* (módulo `HAL`). De esta forma, el alumno tiene una visión de estos módulos como «cajas negras», lo que le permite centrarse en el desarrollo del código del sistema operativo (módulo `kernel`) sin perderse en detalles innecesarios.

A continuación se describe cada una de estas partes.

1.3.2.1. Carga del sistema operativo

De forma similar a lo que ocurre en un sistema real, en este entorno existe un programa de arranque que se encarga de cargar el sistema operativo en memoria y pasar el control a su punto de entrada inicial. Este procedimiento imita el modo de arranque de los sistemas operativos reales que se realiza desde un programa cargador.

El programa cargador se encuentra en el subdirectorio `boot` y, en un alarde de originalidad, se denomina `boot`. Para arrancar el sistema operativo, y con ello el entorno de la práctica, se debe ejecutar dicho programa pasándole como argumento el nombre del archivo que contiene el sistema operativo. Así, suponiendo que el directorio actual corresponde con el subdirectorio `boot` y que el sistema operativo está situado en el directorio `minikernel` y se denomina `kernel`, se debería ejecutar:

```
boot ../minikernel/kernel
```

Nótese que no es necesario ejecutar el programa de arranque desde su directorio. Así, si se está situado en el directorio base del entorno, se podría usar el siguiente mandato:

```
boot/boot minikernel/kernel
```

Una vez arrancado, el sistema operativo continuará ejecutando mientras haya procesos de usuario que ejecutar. Nótese que este comportamiento es diferente al de un sistema real, donde el administrador tiene que realizar alguna operación explícita para parar la ejecución del sistema operativo. Sin embargo, este modelo de ejecución resulta más conveniente tanto para el alumno, a la hora de depurar su práctica, como para el instructor, en el momento de la evaluación.

1.3.2.2. El módulo HAL

El objetivo principal de este módulo es implementar la capa del *hardware* y ofrecer servicios que permitan al sistema operativo su manejo. Las principales características de este *hardware* son las siguientes:

- El «procesador» tiene los dos modos de ejecución clásicos: modo privilegiado o núcleo, en el que se ejecuta código del sistema operativo, y modo usuario, que corresponde con la ejecución del código de procesos de usuario.
- El procesador sólo pasa de modo usuario a privilegiado debido a la ocurrencia de algún tipo de interrupción.
- El registro de estado del procesador almacena el modo de ejecución del mismo. Para pasar de modo privilegiado a modo usuario, el sistema operativo debe modificar ese valor dentro del registro de estado.
- En el sistema hay dos dispositivos de entrada/salida basados en interrupciones: el terminal y el reloj.
- El terminal es de tipo proyectado en memoria. Como se verá en el capítulo de entrada/salida, esto significa que, realmente, el terminal está formado por dos dispositivos independientes: la pantalla y el teclado. La salida de datos a la pantalla del terminal se realiza escribiendo directamente en su memoria de vídeo, no implicando el uso de interrupciones. La entrada de datos mediante el teclado, sin embargo, está dirigida por las interrupciones que se producen cada vez que se pulsa una tecla.
- El reloj temporizador tiene una frecuencia de interrupción programable. Además de este temporizador que interrumpe periódicamente, hay un reloj alimentado con una batería donde se mantiene la hora mientras el equipo está apagado. Este reloj mantiene la hora como el número de milisegundos transcurridos desde el 1 de enero de 1970. Habitualmente, a este reloj se le denomina reloj CMOS.
- El tratamiento de las interrupciones se realiza mediante el uso de una tabla de vectores de interrupción. Hay seis vectores disponibles que corresponden con:
 - Vector 0: Excepción aritmética.
 - Vector 1: Excepción por acceso a memoria inválido.
 - Vector 2: Interrupción de reloj.
 - Vector 3: Interrupción del terminal.
 - Vector 4: Llamada al sistema.
 - Vector 5: Interrupción *software*. Más adelante se explicará el uso de este tipo de interrupción generada por programa.

Los dos primeros se corresponden con excepciones, los dos siguientes con interrupciones de entrada/salida y los dos últimos con interrupciones generadas por programa mediante la ejecución de la instrucción correspondiente, ya sea cuando está ejecutándose en modo usuario (interrupción de llamada al sistema) o cuando está en modo privilegiado (interrupción *software*).

- Se trata de un procesador con múltiples niveles de interrupción que, de mayor a menor prioridad, son los siguientes:
 - *Nivel 3*: Interrupción de reloj.
 - *Nivel 2*: Interrupción del terminal.
 - *Nivel 1*: Interrupción *software* o llamada al sistema.
 - *Nivel 0*: Ejecución en modo usuario.
- En cada momento el procesador ejecuta en un determinado nivel de interrupción, cuyo valor está almacenado en el registro de estado, y sólo admite interrupciones de un nivel superior al actual.
- Inicialmente, el procesador ejecuta en modo privilegiado y en el nivel de interrupción máximo, por lo que todas las interrupciones están inhibidas.
- Cuando el procesador ejecuta en modo usuario (código de los procesos de usuario), ejecuta con un nivel 0, lo que hace que estén habilitadas todas las interrupciones.
- Cuando se produce una interrupción, sea del tipo que sea, el procesador realiza el tratamiento habitual, esto es, almacenar el contador de programa y el registro de estado en la pila, poner el procesador en modo privilegiado, fijar el nivel de interrupción de acuerdo con el tipo de interrupción recibida y cargar en el contador de programa el valor almacenado en el vector correspondiente. Evidentemente, al tratarse de operaciones realizadas por el *hardware*, todas estas operaciones no son visibles al programador del sistema operativo.
- La finalización de una rutina de interrupción conlleva la ejecución de una instrucción de retorno de interrupción que restaurará el nivel de interrupción y el modo de ejecución previos.
- Se dispone de una instrucción que permite modificar explícitamente el nivel de interrupción del procesador.
- Se trata de un procesador con mapas de entrada/salida y memoria separados. Por tanto, hay que usar instrucciones específicas de entrada/salida para acceder a los puertos de los dispositivos. Hay que aclarar que, realmente, sólo hay un puerto de entrada/salida que corresponde con el registro de datos del teclado, el cual, cuando se produce una interrupción, contiene la tecla pulsada.
- El procesador dispone de seis registros de propósito general de 32 bits, que, en principio, sólo se tendrán que usar explícitamente para el paso de parámetros en las llamadas al sistema.

Además de incluir funcionalidad relacionada con el *hardware*, este módulo también proporciona funciones de más alto nivel vinculadas con la gestión de memoria. Con ello se pretende que el alumno se centre en los aspectos relacionados con la gestión de procesos y la entrada/salida y no en los aspectos relacionados con la gestión de memoria.

Las funciones ofrecidas por el módulo HAL al sistema operativo se pueden clasificar en las siguientes categorías:

- Operaciones vinculadas a la iniciación de los controladores de dispositivos. En su arranque, el sistema operativo debe asegurarse de que los controladores de los dispositivos se inician con un estado adecuado. El módulo HAL ofrece una función de este tipo por cada uno de los dos dispositivos existentes. Además, se proporciona un servicio que permite leer la hora del reloj CMOS del sistema.
 - Iniciación del controlador de teclado.

```
void iniciar_cont_teclado();
```

- Iniciación del controlador de reloj. Se especifica como parámetro cuál es la frecuencia de interrupción deseada (número de interrupciones de reloj por segundo).

```
void iniciar_cont_reloj(int ticks_por_seg);
```

- Lectura de la hora almacenada en el reloj CMOS. Normalmente, el sistema operativo lee este valor en el arranque, pero luego el mismo se encarga de mantener la hora actualizada. Nótese que el tipo `long long int` es una extensión presente en el compilador de C de GNU que permite especificar tipos cuyo tamaño es el doble que el de un entero convencional (por tanto, normalmente 64 bits).

```
unsigned long long int leer_reloj_CMOS();
```

- Operaciones relacionadas con las interrupciones. En su fase de arranque, el sistema operativo debe iniciar el controlador de interrupciones a un estado válido y deberá instalar en la tabla de manejadores sus rutinas de tratamiento para cada uno de los vectores que hay en el sistema. Asimismo, se proporciona un servicio para cambiar explícitamente el nivel de interrupción del procesador y otro para activar una interrupción *software*.

- Iniciación del controlador de interrupciones.

```
void iniciar_cont_int();
```

- Instalación de un manejador de interrupciones. Instala la función manejadora `manej` en el vector correspondiente a `nvector`. Existen varias constantes que facilitan la especificación del número de vector.

```
#define EXC_ARITM 0      /* excepción aritmética */
#define EXC_MEM 1        /* excepción en acceso a memoria */
#define INT_RELOJ 2       /* interrupción de reloj */
#define INT_TERMINAL 3   /* interrupción de terminal */
#define LLAM_SIS 4        /* vector usado para llamadas */
#define INT_SW 5          /* vector usado para int. soft. */

void instal_man_int(int nvector, void (*manej)());
```

- Esta función fija el nivel de interrupción del procesador devolviendo el previo. Permite establecer explícitamente un determinado nivel de interrupción, lo que habilita las interrupciones con un nivel superior e inhabilita las que tienen un nivel igual o inferior. Devuelve el nivel de interrupción anterior para así, si se considera oportuno, poder restaurarlo posteriormente usando esta misma función. Están definidas tres constantes que representan los tres niveles de interrupción del sistema.

```
#define NIVEL_1 1 /* Int. Software */
#define NIVEL_2 2 /* Int. Terminal */
#define NIVEL_3 3 /* Int. Reloj */
```

```
int fijar_nivel_int(int nivel);
```

- Esta función ejecuta la instrucción *hardware* que produce una interrupción *software*, que será tratada cuando el nivel de interrupción del procesador lo posibilite.

```
void activar_int_SW();
```

- Esta función consulta el registro de estado salvado por la interrupción actual y permite conocer si previamente se estaba ejecutando en modo usuario, devolviendo un valor verdadero en tal caso.

```
int viene_de_modo_usuario();
```

- Operaciones de gestión de la información de contexto del proceso. En el contexto del proceso (tipo `contexto_t`) se almacena una copia de los registros del procesador con los valores correspondientes a la última vez que ejecutó este proceso. Se ofrecen funciones para crear el contexto inicial de un nuevo proceso, así como para realizar un cambio de contexto, o sea, salvar el contexto de un proceso y restaurar el de otro.
 - Este servicio crea el contexto inicial del proceso estableciendo los valores iniciales de los registros contador de programa (parámetro `pc_inicial`) y puntero de pila a partir de la dirección inicial de la pila (parámetro `inicio_pila`) y su tamaño (parámetro `tam_pila`). Además, recibe como parámetro una referencia al mapa de memoria del proceso (parámetro `memoria`), que se debe haber creado previamente. Esta función devuelve un contexto iniciado de acuerdo con los valores especificados (parámetro de salida `contexto_ini`). Es importante resaltar que la copia del registro de estado dentro del contexto se inicia con un nivel de interrupción 0 y con un modo de ejecución usuario. De esta forma, cuando el sistema operativo active el proceso por primera vez mediante un cambio de contexto, el código del proceso se ejecutará en el modo y nivel de interrupción adecuados.

```
void fijar_contexto_ini(void *memoria, void *inicio_pila, int tam_pila, void *pc_inicial, contexto_t *contexto_ini);
```

- Esta rutina salva el contexto de un proceso y restaura el de otro. Concretamente, la salvaguardia consiste en copiar el estado actual de los registros del procesador en el parámetro de salida `contexto_a_salvar`. Por su parte, la restauración implica copiar el contexto recibido en el parámetro de entrada `contexto_a_restaurar` en los registros del procesador. Nótese que, al terminar la operación de restauración, se ha «congelado» la ejecución del proceso que invocó esta rutina y se ha «descongelado» la ejecución del proceso restaurado justo por donde se quedó la última vez que ejecutó, ya sea en otra llamada a `cambio_contexto`, si ya ha ejecutado previamente, o desde su contexto inicial, si ésta es la primera vez que ejecuta. El proceso «congelado» no volverá a ejecutar, y, por tanto, no retornará de la llamada a la función de `cambio_contexto`, hasta que otro proceso llame a esta misma rutina especificando como contexto a restaurar el de este proceso. Hay que resaltar que, dado que también se salva y restaura el registro de estado, se recuperará el nivel de interrupción que tenía previamente el proceso restaurado. Si no se especifica el proceso cuyo contexto debe salvarse (primer parámetro nulo), sólo se realiza la restauración.

```
void cambio_contexto(contexto_t *contexto_a_salvar, contexto_t *contexto_a_restaurar);
```

- Funciones relacionadas con el mapa de memoria del proceso. Como se ha explicado previamente, el módulo HAL, además de las funciones vinculadas directamente con el *hardware*, incluye operaciones de alto nivel que gestionan todos los aspectos relacionados con la gestión de memoria. Se ofrecen servicios que permiten realizar operaciones tales como crear el mapa de memoria del proceso a partir del ejecutable y liberarla cuando sea oportuno, así como para crear la pila del proceso y liberarla.

- Esta rutina crea el mapa de memoria a partir del ejecutable especificado (parámetro `prog`). Para ello debe procesar el archivo ejecutable y crear las regiones de memoria (código y datos) correspondientes. Devuelve un identificador del mapa de memoria creado, así como la dirección de inicio del programa en el parámetro de salida `dir_ini`.

```
void *crear_imagen(char *prog, void **dir_ini);
```

- Este servicio libera una imagen de memoria previamente creada (parámetro `mem`).

```
void liberar_imagen(void *mem);
```

- Esta rutina reserva una zona de memoria para la región de pila. Se especifica como parámetro el tamaño de la misma, devolviendo como resultado la dirección inicial de la zona reservada.

```
void *crear_pila(int tam);
```

- Este servicio libera una pila previamente creada (parámetro `pila`).

```
void liberar_pila(void *pila);
```

- Operaciones misceláneas. En este apartado se agrupa una serie de funciones de utilidad diversa.

- Rutinas que permiten leer y escribir, respectivamente, en los registros de propósito general del procesador.

```
int leer_registro(int nreg);
```

```
int escribir_registro(int nreg, int valor);
```

- Esta función lee y devuelve un byte del puerto de entrada/salida especificado (parámetro `dir Puerto`). El único puerto disponible en el sistema corresponde con el terminal.

```
#define DIR_TERMINAL 1
```

```
char leer_puerto(int dir Puerto);
```

- Ejecuta la instrucción `HALT` del procesador que detiene su ejecución hasta que se active una interrupción.

```
void halt();
```

- Esta función permite escribir en la pantalla los datos especificados en el parámetro `buffer` y cuyo tamaño corresponde con el parámetro `longi`. Para ello, copia en la memoria de vídeo del terminal dichos datos. La rutina de conveniencia `printk` se apoya en la anterior y permite escribir datos con formato, al estilo del clásico `printf` de C. Nótese que se ha usado el mismo nombre que la rutina de las mismas características disponible en MINIX y Linux (el nombre `printk` proviene de un apócope de `print kernel`).

```
void escribir_ker(char *buffer, unsigned int longi);
```

```
int printk(const char *, ...);
```

- Esta función escribe el mensaje especificado (parámetro `mens`) por la pantalla y termina la ejecución del sistema operativo. Una rutina con este mismo nombre (en inglés, evidentemente) y la misma funcionalidad está disponible en la mayoría de los sistemas UNIX.

```
void panico(char *mens);
```

1.3.2.3. El módulo *kernel*

Éste es el módulo que contiene la funcionalidad del sistema operativo y, por tanto, es en él donde se centrará el trabajo del alumno. Queda a criterio del instructor decidir qué se le entrega al

alumno como versión inicial de este módulo. Una opción sería no proporcionarle nada. En este caso, el alumno debería programar desde cero el sistema operativo, incluyendo la iniciación de los dispositivos, la instalación de los manejadores de interrupción, la funcionalidad para crear un proceso, etc. Sin embargo, nuestra experiencia docente recomienda una estrategia menos drástica, que consiste en proporcionar al alumno una versión inicial del sistema operativo que incluya una funcionalidad básica. Esta versión básica permite al alumno familiarizarse con el entorno, lo que no resulta fácil al principio, y dado que agiliza el arranque del trabajo práctico, permite que el instructor pueda plantear prácticas más ambiciosas. En esta sección se presenta esta versión inicial recomendada, aunque hay que reiterar que es el instructor el que tiene la última palabra sobre cuál será el punto de partida del alumno.

A continuación se describen las principales características del sistema inicial. Hay que resaltar que, dado que esta descripción se incluye en el primer capítulo del libro, ésta sólo se ocupa de aspectos generales de esta versión inicial del sistema operativo. En el capítulo dedicado a los procesos se comentarán en detalle otros aspectos de este módulo.

- Iniciación. Una vez cargado el sistema operativo, el programa cargador pasa control al punto de entrada del mismo (en este caso, a la función `main` de este módulo). En este momento, el sistema inicia sus estructuras de datos, los dispositivos *hardware* e instala sus manejadores en la tabla de vectores. En último lugar, crea el proceso inicial `init` y lo activa pasándole el control. Nótese que durante esta fase el procesador ejecuta en modo privilegiado y las interrupciones están inhibidas (nivel de interrupción 3). Sin embargo, cuando se activa el proceso `init` restaurándose su contexto inicial, el procesador pasa a ejecutar en modo usuario y se habilitan automáticamente todas las interrupciones (nivel 0), puesto que en dicho contexto inicial se ha establecido previamente que esto sea así. Obsérvese que, una vez invocada la rutina de cambio de contexto, no se puede volver nunca a esta función, ya que no se ha salvado el contexto del flujo actual de ejecución. A partir de ese momento, el sistema operativo sólo se ejecutará cuando se produzca una llamada al sistema, una excepción o una interrupción de un dispositivo.
- Tratamiento de interrupciones externas. Las únicas fuentes externas de interrupciones son el reloj y el terminal. Las rutinas de tratamiento instaladas únicamente muestran un mensaje por la pantalla indicando la ocurrencia del evento. En el caso de la interrupción del teclado, la rutina, además, usa la función `leer Puerto` para obtener el carácter tecleado. En estas rutinas habrá que incluir progresivamente la funcionalidad pedida en las distintas prácticas.
- Tratamiento de interrupción *software*. Como ocurre con las interrupciones externas, la rutina de tratamiento sólo muestra un mensaje por la pantalla.
- Tratamiento de excepciones. Las dos posibles excepciones presentes en el sistema tienen un tratamiento común que depende de en qué modo ejecutaba el procesador antes de producirse la excepción. Si estaba en modo usuario, se termina la ejecución del proceso actual. En caso contrario, se trata de un error del propio sistema operativo. Por tanto, se invoca la rutina `panic` para terminar su ejecución.
- Llamadas al sistema. Existe una única rutina de interrupción para todas las llamadas (rutina `tratar_llamsis`). Tanto el código numérico de la llamada como sus parámetros se pasan mediante registros. Por convención, el código se pasa en el registro 0 y los parámetros en los siguientes registros (parámetro 1 en registro 1, parámetro 2 en registro 2, y así sucesivamente hasta 5 parámetros). Asimismo, el resultado de la llamada se devuelve en el registro 0. La rutina `tratar_llamsis` obtiene el código numérico de la llamada e invoca indirectamente a través de `tabla_servicios` a la función correspondiente. Esta tabla guarda en cada posición la dirección de la rutina del sistema operativo que lleva a cabo la

llamada al sistema cuyo código corresponde con dicha posición. Para ilustrar el modo de operación de esta rutina, a continuación se presenta su código:

```
static void tratar_llamsis(){
    int nserv, res;

    nserv=leer_registro(0);
    if (nserv<NSERVICIOS)
        res=(tabla_servicios[nserv].fservicio)();
    else
        res=-1; /* servicio no existente */
    escribir_registro(0,res);
    return;
}
```

- En la versión inicial sólo hay tres llamadas disponibles. La función asociada con cada una de ellas está indicada en la posición correspondiente de `tabla_servicios` y, como se ha comentado previamente, será invocada desde `tratar_llamsis` cuando el valor recibido en el registro 0 así lo indique. Las llamadas disponibles en esta versión inicial son las siguientes:

```
#define CREAR_PROCESO 0
#define TERMINAR_PROCESO 1
#define ESCRIBIR 2

servicio tabla_servicios[NSERVICIOS]={
    {sis_crear_proceso},
    {sis_terminar_proceso},
    {sis_escribir}};
```

Hay que resaltar que la rutina de tratamiento de una llamada no recibe sus parámetros de forma convencional (o sea, en la pila), sino que debe tomarlos de los registros del procesador. Sin embargo, el resultado sí lo devuelve de la manera tradicional (mediante la sentencia `return`), puesto que `tratar_llamsis` se encarga posteriormente de copiarlo al registro 0. A continuación se describen someramente las tres llamadas existentes en la versión inicial de este sistema operativo.

- *Crear proceso.* Crea un proceso que ejecuta el programa almacenado en el archivo especificado como parámetro. Esta llamada devolverá un -1 si hay un error y un 0 en caso contrario. A continuación se presenta el código de esta llamada para así ilustrar cómo se accede a los parámetros de la llamada mediante la función `leer_registro`.

```
int sis_crear_proceso(){
    char *prog;
    int res;

    prog=(char *)leer_registro(1);
    res=crear_tarea(prog);
    return res;
}
```

Obsérvese que, en este caso, el código que realiza el tratamiento real de la llamada no se ha incluido en la propia función, sino que se ha delegado a una función auxiliar. Puesto que esta rutina auxiliar se invoca de manera convencional, va a recibir los parámetros de la forma habitual, como se puede observar en su prototipo:

```
static int crear_tarea(char *prog);
```

Esta delegación en una función auxiliar puede utilizarse para generar un código más estructurado, pero su uso principal aparece cuando se requiere que una determinada operación pueda ser invocada tanto desde una llamada al sistema como internamente. Esto ocurre con la operación de crear un proceso que, además de corresponder con una llamada al sistema, se necesita invocar internamente para crear al primer proceso. La llamada al sistema simplemente recoge los parámetros de los registros y realiza una llamada convencional a la función auxiliar.

- *Terminar proceso.* Termina la ejecución de un proceso liberando sus recursos.
- *Escribir.* Escribe un mensaje por la pantalla haciendo uso de la función `escribir_ker` proporcionada por el módulo HAL. Recibe como parámetros la información que se desea escribir y su longitud. Devuelve siempre un 0.

1.3.2.4. Los programas de usuario

En el subdirectorio `usuario` existe inicialmente un conjunto de programas de ejemplo que usan los servicios del `minikernel`. De especial importancia es el programa `init`, puesto que es el primer programa que arranca el sistema operativo. En un sistema real, este programa consulta archivos de configuración para arrancar otros programas que, por ejemplo, se encarguen de atender a los usuarios (procesos de `login`). De manera relativamente similar, en nuestro sistema, este proceso hará el papel de lanzador de otros procesos, aunque en nuestro caso no se trata de procesos que atiendan al usuario, puesto que el sistema no proporciona inicialmente servicios para leer del terminal. Se tratará simplemente de programas que realizan una determinada labor y terminan.

Asimismo, tanto el instructor como el propio alumno podrán desarrollar los programas que consideren oportunos para evaluar las prácticas o para probar el correcto funcionamiento de la funcionalidad que el alumno añade al sistema, respectivamente.

Como ocurre en un sistema real, los programas tienen acceso a las llamadas al sistema como rutinas de biblioteca. Para ello, existe una biblioteca estática, denominada `libserv.a`, que contiene las funciones de interfaz para las llamadas al sistema.

```
int crear_proceso(char *programa);
int terminar_proceso();
int escribir(char *texto, unsigned int longi)
```

Los programas de usuario no deben usar llamadas al sistema operativo nativo, aunque sí podrán usar funciones de la biblioteca estándar de C, como, por ejemplo, `strcpy` o `memcpy`.

La biblioteca `libserv.a` está almacenada en el subdirectorio `usuario/lib` y está compuesta de dos módulos:

- `serv`. Contiene las rutinas de interfaz para las llamadas. Se apoya en una función del módulo `misc` denominada `llamsis`, que es la que realmente ejecuta la instrucción de llamada al sistema. Para hacer accesible a los programas una nueva llamada, el alumno deberá modificar este archivo para incluir la rutina de interfaz correspondiente.
- `misc`. Como intenta indicar su nombre, este módulo contiene un conjunto diverso de funciones de utilidad. Dado el carácter «oscuro» de algunas de estas rutinas, se recomienda no proporcionar al alumno el código fuente de este módulo, aunque queda a criterio del instructor. A continuación se detallan las funciones de utilidad presentes en este módulo:

- La función trap, que contiene la instrucción que causa una llamada al sistema.
- Las funciones leer_registro y escribir_registro, que, respectivamente, permiten leer y escribir en los registros del procesador. Son idénticas a las proporcionadas en el módulo HAL, pero destinadas a ser utilizadas por los programas de usuario para pasar los parámetros de las llamadas al sistema.
- Aunque se pueden implementar las funciones de interfaz de las llamadas usando directamente la función trap y las funciones que permiten leer y escribir en los registros, se proporciona una manera más sencilla basándose en la función llamsis. Esta función de conveniencia facilita la invocación de una llamada al sistema ocupándose de la tediosa labor de llenar los registros con los valores adecuados e invocando a continuación la función trap. A continuación se muestra la estructura simplificada de esta función para que se pueda comprender mejor su funcionamiento:

```

int llamsis(int llamada, int nargs, ... /* argumentos */)
{
    int i;
    escribir_registro(0, llamada);
    for(i=1; nargs; nargs--) i++;
    escribir_registro(i, args[i]);
}

trap();
return leer_registro(0);
}

```

Usando esta función de conveniencia, las rutinas de interfaz de las llamadas se simplifican considerablemente. Así, por ejemplo, la llamada a crear_proceso queda de la siguiente forma:

```

int crear_proceso(char *prog)
{
    return llamsis(CREAR_PROCESO, 1, (int)prog);
}

```

- La definición de la función printf, que, evidentemente, se apoya en la llamada al sistema escribir de la misma manera que el printf en un sistema UNIX se apoya en la llamada write.
- Por último, está definida la función start, que va constituir el punto real de arranque del programa (o sea, la dirección a la que apuntará el contador de programa inicial en el arranque del programa). Esta rutina se encarga de invocar a la función main, que, a todos los efectos, es el punto de arranque del programa para el usuario. Este artificio, utilizado en casi todos los sistemas operativos, permite asegurarse de que el programa siempre invoca la llamada al sistema de finalización (terminar_proceso) al terminar; si no lo ha hecho la función main, lo hará start al terminar la función main.

Todos los programas de usuario utilizan el archivo de cabecera usuario/include/servicios.h, que contiene los prototipos de las funciones de interfaz a las llamadas al sistema, así como el de la función printf.

1.3.3. Consideraciones generales sobre las prácticas basadas en el *minikernel*

En los capítulos de procesos, comunicación de procesos y entrada/salida, además de en este mismo, se propondrán prácticas de diseño basadas en este entorno. En este apartado se exponen algunos comentarios generales sobre las prácticas que se pueden plantear en este entorno.

En primer lugar, hay que resaltar que las prácticas que se proponen en los distintos capítulos son sólo una muestra del tipo de prácticas que se pueden desarrollar en este entorno. Una vez que el instructor se familiarice con el mismo, puede modificar a su gusto las prácticas propuestas en este libro o plantear sus propias prácticas. La imaginación es el único límite.

Dado que todas las prácticas propuestas parten del mismo código de apoyo e implican modificar los mismos archivos, en las próximas secciones se explicarán estos aspectos para evitar tener que repetirlos en el enunciado de cada práctica específica.

Por último, hay que resaltar que el orden de presentación de estas prácticas a lo largo del libro no implica que el alumno tenga que respetarlo a la hora de desarrollarlas. El criterio del instructor y las preferencias del propio alumno pueden estimar un orden alternativo a la hora de acometer estas prácticas. Asimismo, puede plantearse una forma de trabajo incremental, de manera que se tome como punto de partida de cada práctica la solución de las prácticas previamente realizadas en este entorno o utilizar siempre como punto de partida la versión inicial proporcionada como material de apoyo. La primera opción tiene la ventaja de que el alumno alcanza un mayor grado de satisfacción al sentir cómo va creando de la nada un sistema operativo con una complejidad apreciable. La única pega es que el tamaño del código va creciendo progresivamente, dificultando la depuración del mismo.

1.3.4. Código fuente de apoyo

El código de apoyo para el conjunto de prácticas basadas en el *minikernel* es, evidentemente, el código del propio *minikernel*. En la página web del libro aparecen dos versiones del mismo: una completa destinada al instructor (*minikernel_instructor.tgz*) y otra que contiene el material de apoyo que recomendamos que se suministre al alumno (*minikernel.tgz*). En este segundo «paquete» no se incluye el código fuente de algunos módulos (por ejemplo, del módulo HAL) y se proporciona una versión inicial del sistema operativo tal como la descrita en la sección anterior. Dado que el instructor dispone de todo el material, puede crear su propio «paquete» de material de apoyo con el contenido que considere oportuno. En esta sección se describe el material de apoyo recomendado que contiene el siguiente árbol de archivos:

- **Makefile.** *Makefile* general del entorno. Invoca a los archivos *Makefile* de los subdirectorios subyacentes.
- **boot.** Este directorio está relacionado con la carga del sistema operativo. Contiene el siguiente archivo:
 - **boot.** Programa de arranque del sistema operativo.
- **minikernel.** Este directorio contiene todos los archivos correspondientes a la capa *hardware* y al sistema operativo:
 - **Makefile.** Permite generar el ejecutable del sistema operativo.
 - **kernel.** Archivo que contiene el ejecutable del sistema operativo.
 - **HAL.o.** Archivo objeto que contiene la capa del *hardware*.
 - **kernel.c.** Archivo que contiene la funcionalidad del sistema operativo. Este archivo **debe ser modificado** por el alumno para incluir la funcionalidad pedida en cada práctica.

- **include.** Subdirectorio que contiene los archivos de cabecera usados por el entorno:
 - HAL.h. Archivo que contiene los prototipos de las funciones del módulo HAL. Este archivo **no debe ser modificado** por el alumno.
 - const.h. Archivo que contiene algunas constantes útiles, como, por ejemplo, los vectores de interrupción existentes en el sistema.
 - llamsis.h. Archivo que contiene los códigos numéricos asignados a cada llamada al sistema. Este archivo **debe ser modificado** por el alumno para incluir nuevas llamadas.
 - kernel.h. Contiene definiciones usadas por el módulo kernel.c, como, por ejemplo, la tabla de servicios. Este archivo **debe ser modificado** por el alumno.
- **usuario.** Este directorio contiene diversos programas de usuario.
 - Makefile. Permite compilar los programas de usuario.
 - init.c. Primer programa que ejecuta el sistema operativo. El alumno puede modificarlo a su conveniencia para que éste invoque los programas que se consideren oportunos.
 - *.c. Programas de prueba. El alumno puede modificar a su gusto los ya existentes o incluir nuevos.
 - include. Subdirectorio que contiene los archivos de cabecera usados por los programas de usuario:
 - servicios.h. Archivo que contiene los prototipos de las funciones que sirven de interfaz a las llamadas al sistema. **Debe ser modificado** por el alumno para incluir la interfaz a las nuevas llamadas.
 - lib. Este directorio contiene los módulos que permiten generar la biblioteca que utilizan los programas de usuario. Su contenido es:
 - Makefile. Genera la biblioteca de servicios.
 - libserv.a. La biblioteca de servicios.
 - serv.c. Archivo que contiene la interfaz a los servicios del sistema operativo. Este archivo **debe ser modificado** por el alumno para incluir la interfaz a las nuevas llamadas.
 - misc.o. Contiene otras funciones de biblioteca auxiliares.

1.3.5. Recomendaciones generales

Es importante que el alumno se familiarice con el entorno antes de abordar las distintas prácticas que se proponen en el libro. Se deben llegar a entender las relaciones entre los distintos componentes del sistema.

Es recomendable dotar al alumno de libertad a la hora de diseñar el sistema siempre que proporcione la funcionalidad pédida por cada práctica.

Las características de la simulación hacen que la utilización del depurador no sea de gran ayuda. Por ello, se recomienda el desarrollo de funciones de depuración que muestren por la pantalla el contenido de diversas estructuras de datos.

Otro aspecto que conviene resaltar es que, debido al esquema de compilación usado en la práctica, puede ocurrir que un error de programación (como, por ejemplo, usar printf en vez de printf) aparezca simplemente como un *warning* en la fase de compilación y montaje. El error como tal no aparecerá hasta que se ejecute el sistema. En resumen, vigile los *warnings* que se producen durante la compilación.

1.3.6. Entrega de documentación

El alumno deberá entregar los archivos del entorno que ha modificado junto con una memoria que describa el trabajo realizado:

- memoria.txt: Memoria de la práctica. En ella se deben comentar los aspectos del desarrollo de cada práctica que el alumno considere más relevantes.
- minikernel/kernel.c: Archivo que contiene la funcionalidad del sistema operativo.
- minikernel/include/llamsis.h: Archivo que contiene el código numérico asignado a cada llamada.
- minikernel/include/kernel.h: Archivo que contiene definiciones usadas por el sistema operativo.
- usuario/include/servicios.h: Archivo que contiene los prototipos de las funciones de interfaz a las llamadas.
- usuario/lib(serv.c: Archivo que contiene las definiciones de las funciones de interfaz a las llamadas.

1.3.7. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- D. P. Bovet y M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.

1.4. PRÁCTICA: INTRODUCCIÓN DE UNA NUEVA LLAMADA AL SISTEMA EN EL MINIKERNEL

1.4.1. Objetivo de la práctica

El objetivo principal de esta práctica es lograr que el alumno se familiarice con el entorno del *minikernel*. Hay que resaltar que la práctica apenas conlleva esfuerzo de programación. Sin embargo, al tratarse del primer contacto con el entorno, el alumno puede encontrar ciertas dificultades hasta que consiga entender cómo está organizado el mismo.

- **NIVEL:** Diseño
- **HORAS ESTIMADAS:** 5

1.4.2. Descripción de la funcionalidad que debe desarrollar el alumno

1.4.2.1. Pasos para la inclusión de una nueva llamada al sistema

Dado que en la práctica se pretende incluir una nueva llamada al sistema, a continuación se presentan los pasos típicos que hay que llevar a cabo en este sistema para hacerlo. Suponiendo que el nuevo servicio se denomina *nueva*, éstos son los pasos a realizar:

- Incluir en *minikernel/kernel.c* una rutina (que podría denominarse *sis_nueva*) con el código de la nueva llamada.

- Incluir en `tabla_servicios` (archivo `minikernel/include/kernel.h`) la nueva llamada en la última posición de la tabla.
- Modificar el archivo `kernel/include/llamsis.h` para incrementar el número de llamadas disponibles y asignar el código más alto a la nueva llamada.
- Una vez realizados los pasos anteriores, el sistema operativo ya incluiría el nuevo servicio, pero sólo sería accesible desde los programas usando código ensamblador. Por tanto, es necesario modificar la biblioteca de servicios (archivo `usuario/lib/serv.c`) para que proporcione la interfaz para el nuevo servicio. Se debería también modificar el archivo de cabecera que incluyen los programas de usuario (`usuario/include/servicios.h`) para que dispongan del prototipo de la función de interfaz.
- Por último, hay que crear programas de prueba para este nuevo servicio y, evidentemente, modificar `init` para que los invoque. Asimismo, se debería modificar el archivo `Makefile` para facilitar la compilación de este nuevo programa. Evidentemente, si se usan los programas de prueba contenidos en la distribución inicial, no es necesario realizar este paso.

1.4.2.2. Llamada al sistema `obtener_id_pr`

Se debe añadir una nueva llamada, denominada `obtener_id_pr`, que devuelva el identificador del proceso que la invoca. Como puede observarse, se trata de un servicio que realiza un trabajo muy sencillo. Sin embargo, esta primera tarea servirá para familiarizarse con el mecanismo que se usa para incluir una nueva llamada al sistema. El prototipo de la función de interfaz sería el siguiente:

```
int obtener_id_pr();
```

Hay que resaltar que, aunque en este primer capítulo no se ha explicado cómo se realiza la gestión de procesos en la versión inicial del sistema operativo, esta carencia no resulta problemática para realizar esta práctica. Sólo se necesita saber que el bloque de control del proceso actual está siempre apuntado por la variable de tipo puntero `p_proc_actual` y que, dentro del mismo, hay un campo que contiene el identificador de proceso (campo `id`). En el próximo capítulo se estudiará con detalle la gestión de procesos en este entorno.

2

Procesos

En este capítulo se presentan las prácticas relacionadas con la gestión de procesos en sistemas operativos. El capítulo tiene tres objetivos básicos: mostrar al lector algunos conceptos básicos desde el punto de vista de usuario, describir los servicios que da el sistema operativo y proponer un conjunto de prácticas que permita cubrir los aspectos básicos y de diseño de procesos y procesos ligeros. De esta forma, se pueden adaptar las prácticas del tema a distintos niveles de conocimiento.

2.1. CONCEPTOS BÁSICOS DE PROCESOS

Todos los programas cuya ejecución solicitan los usuarios lo hacen en forma de procesos, de ahí la importancia para el informático de conocerlos en detalle. El proceso se puede definir como un **programa en ejecución**, y de una forma un poco más precisa, como la unidad de procesamiento gestionada por el sistema operativo.

El sistema operativo mantiene una tabla de procesos, dentro de la cual se almacena un **bloque de control del proceso (BCP)** por cada proceso. Por razones de eficiencia, la tabla de procesos se construye normalmente como una estructura estática que tiene un determinado número de BCP, todos ellos del mismo tamaño.

El BCP contiene la información básica del proceso, entre las que cabe destacar la siguientes:

- Información de identificación. Esta información identifica al usuario y al proceso. Como ejemplo, se incluyen los siguientes datos:
 - Identificador del proceso.
 - Identificador del proceso padre en caso de existir relaciones padre-hijo, como es el caso de UNIX.
 - Información sobre el usuario (identificador de usuario, identificador de grupo).
- Estado del procesador. Contiene los valores iniciales del estado del procesador o su valor en el instante en que fue interrumpido el proceso.
- Información de control del proceso. En esta sección se incluye diversa información que permite gestionar al proceso. Se destacan los siguientes datos:
 - Información de planificación y estado:
 - Estado del proceso.
 - Evento por el que espera el proceso cuando está bloqueado.
 - Prioridad del proceso.
 - Información de planificación.
 - Descripción de los segmentos de memoria asignados al proceso.
 - Recursos asignados, tales como:
 - Archivos abiertos (tabla de descriptores o manejadores de archivo).
 - Puertos de comunicación asignados.
 - Punteros para estructurar los procesos en colas o anillos. Por ejemplo, los procesos que están en estado de listo pueden estar organizados en una cola, de forma que se facilite la labor del planificador.
 - Comunicación entre procesos. El BCP puede contener espacio para almacenar las señales y para algún mensaje enviado al proceso.

Algunos sistemas operativos, como UNIX y Linux, mantienen de forma explícita una estructura jerárquica de procesos mediante una relación padre-hijo —un proceso sabe quién es su padre, el proceso que lo creó—, mientras que otros sistemas operativos como Windows NT/2000 no la mantienen.

El **entorno de un proceso** consiste en un conjunto de variables que se le pasan al proceso en el momento de su creación. El entorno está formado por una tabla NOMBRE-VALOR que se incluye en la pila del proceso. El NOMBRE especifica el nombre de la variable y el VALOR su valor. Un ejemplo de entorno en UNIX es el siguiente:

```

PATH=/usr/bin:/home/pepe/bin
TERM=vt100
HOME=/home/pepe
PWD=/home/pepe/libros/primero

```

En este ejemplo, PATH indica la lista de directorios en los que el sistema operativo busca los programas ejecutables, TERM el tipo de terminal, HOME el directorio inicial asociado al usuario y PWD el directorio actual de trabajo. Los procesos pueden utilizar las variables del entorno para definir su comportamiento. Por ejemplo, un programa de edición responderá a las teclas de acuerdo al tipo de terminal que esté utilizando el usuario y que viene definido en la variable TERM.

Estados de los procesos

De acuerdo con la Figura 2.1, un proceso puede estar en determinadas situaciones (procesamiento, listo y espera) que denominaremos estados. A lo largo de su vida, el proceso va cambiando de estado, según evolucionan sus necesidades.

Como se puede observar en la Figura 2.1, no todos los procesos activos de un sistema multitarea están en la misma situación. Se diferencian, por tanto, tres estados básicos en los que puede estar un proceso, estados que detallamos seguidamente:

- **Ejecución.** En este estado se encuentra el proceso que está siendo ejecutado por el procesador, es decir, que está en fase de procesamiento. En este estado, el estado del proceso reside en los registros del procesador.
- **Bloqueado.** Un proceso bloqueado está esperando a que ocurra un evento y no puede seguir ejecutando hasta que termine el evento. Una situación típica de proceso bloqueado se produce cuando el proceso solicita una operación de E/S. Hasta que no termina esta operación, el proceso queda bloqueado. En este estado, el estado del proceso reside en el BCP.
- **Listo.** Un proceso está listo para ejecutar cuando puede entrar en fase de procesamiento. Dado que puede haber varios procesos en este estado, una de las tareas del sistema operativo será seleccionar aquel que debe pasar a ejecución. El módulo del sistema operativo que toma esta decisión se denomina **planificador**. En este estado, el estado del proceso reside en el BCP.

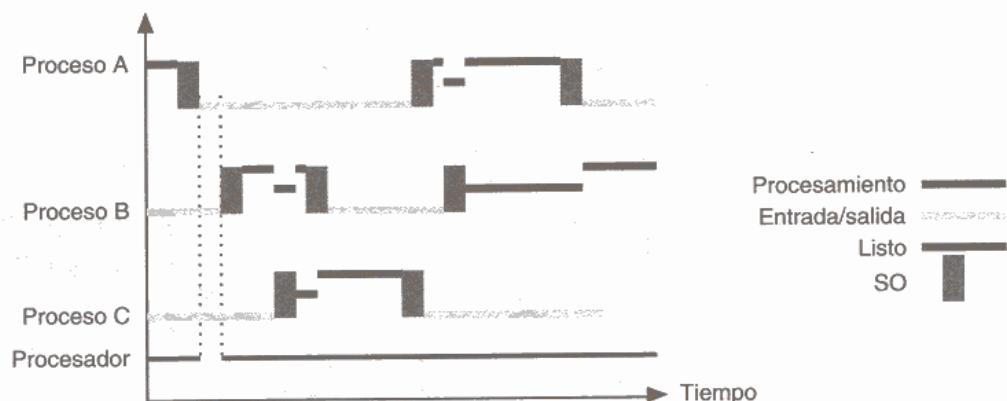


Figura 2.1. Ejemplo de ejecución en un sistema multitarea.

Planificador y activador

El planificador forma parte del núcleo del sistema operativo. Entra en ejecución cada vez que se activa el sistema operativo y su misión es seleccionar el proceso que se ha de ejecutar a continuación.

El activador también forma parte del sistema operativo y su función es poner en ejecución el proceso seleccionado por el planificador. La activación del sistema operativo se realiza mediante el mecanismo de las interrupciones. Cuando se produce una interrupción, se realizan las dos operaciones siguientes:

- Se salva el estado del procesador en el correspondiente BCP.
- Se pasa a ejecutar la rutina de tratamiento de interrupción del sistema operativo.

Llamaremos **cambio de contexto** al conjunto de estas operaciones.

2.1.1. Procesos ligeros

Un proceso ligero o *thread* es un programa en ejecución (flujo de ejecución) que comparte la imagen de memoria y otras informaciones con otros procesos ligeros. Como muestra la Figura 2.2, un proceso puede contener un solo flujo de ejecución, como ocurre en los procesos clásicos, o más de un flujo de ejecución (procesos ligeros).

Desde el punto de vista de la programación, cada proceso ligero se define como una función cuya ejecución se puede lanzar en paralelo con otras. El hilo de ejecución primario o proceso ligero primario corresponde a la función `main`.

Cada proceso ligero tiene informaciones que le son propias y que no comparte con otros procesos ligeros. Las informaciones propias se refieren fundamentalmente al contexto de ejecución, pudiéndose destacar las siguientes:

- Contador de programa.
- Pila.
- Registros.
- Estado del proceso ligero (ejecutando, listo o bloqueado).

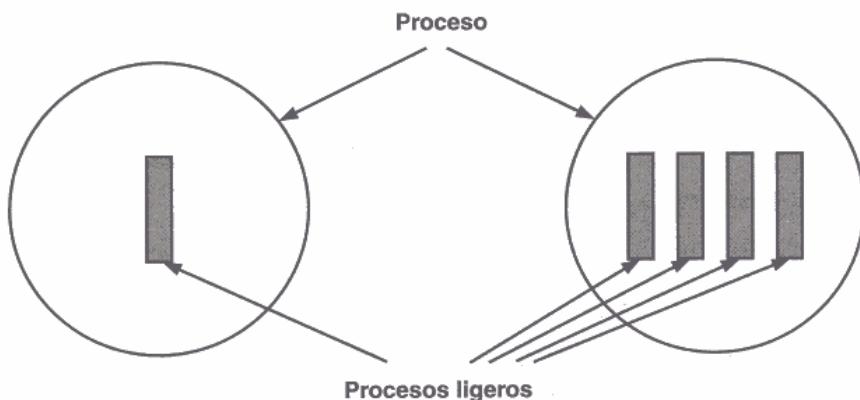


Figura 2.2. Proceso ligero.

Todos los procesos ligeros de un mismo proceso comparten la información del mismo, en concreto comparten espacio de memoria, variables globales, archivos abiertos, procesos hijos, temporizadores, señales y semáforos y contabilidad.

Es importante destacar que todos los procesos ligeros de un mismo proceso comparten el mismo espacio de direcciones de memoria, que incluye el código, los datos y las pilas de los diferentes procesos ligeros. Esto hace que no exista protección de memoria entre los procesos ligeros de un mismo proceso, algo que sí ocurre con los procesos convencionales.

El proceso ligero puede estar en uno de los tres estados de ejecutando, listo para ejecutar y bloqueado. Cada proceso ligero de un proceso tiene su propio estado, pudiendo estar unos bloqueados, otros listos y otro en ejecución.

Diseño con procesos ligeros

La utilización de procesos ligeros ofrece las ventajas de división de trabajo que dan los procesos, pero con una mayor sencillez, lo que se traduce en mejores prestaciones. En este sentido, es de destacar que los procesos ligeros comparten memoria directamente, por lo que no hay que añadir ningún mecanismo adicional para utilizarla y que la creación y destrucción de procesos ligeros requiere mucho menos trabajo que la de procesos.

Las ventajas de diseño que se pueden atribuir a los procesos ligeros son las siguientes:

- Permite la separación de tareas. Cada tarea se puede encapsular en un proceso ligero independiente.
- Facilita la modularidad al dividir trabajos complejos en tareas.
- Aumenta la velocidad de ejecución del trabajo, puesto que aprovecha los tiempos de bloqueo de unos procesos ligeros para ejecutar otros.

El paralelismo que permiten los procesos ligeros, unido a que comparten memoria (utilizan variables globales que ven todos ellos), lleva a la programación concurrente. Este tipo de programación tiene un alto nivel de dificultad, puesto que hay que garantizar que el acceso a los datos compartidos se haga de forma correcta. Los principios básicos que hay que aplicar son los siguientes:

- Hay variables globales que se comparten entre varios procesos ligeros. Dado que cada proceso ligero ejecuta de forma independiente a los demás, es fácil que ocurran accesos incorrectos a estas variables.
- Para ordenar la forma en que los procesos ligeros acceden a los datos se emplean mecanismos de sincronización como el mutex, que se describirá en el Capítulo 4. El objetivo de estos mecanismos es impedir que un proceso ligero acceda a unos datos mientras los esté utilizando otro.
- Para escribir un código correcto hay que imaginar que los códigos de los otros procesos ligeros que pueden existir están ejecutando cualquier sentencia al mismo tiempo que la sentencia que estamos escribiendo.

2.1.2. Planificación

El objetivo de la planificación de procesos y procesos ligeros es el reparto del tiempo de procesador entre los procesos que desean y pueden ejecutar. El **planificador** es el módulo del sistema operativo que realiza la función de seleccionar el proceso en estado de listo que pasa a estado de

ejecución, mientras que el **activador** es el módulo que pone en ejecución el proceso planificado. También es importante la planificación de entrada/salida. Esta planificación decide el orden en que se ejecutan las operaciones de entrada/salida que están encoladas para cada periférico.

Expulsión

La planificación puede ser con expulsión o sin ella. En un sistema sin expulsión, un proceso conserva el procesador mientras lo deseé, es decir, mientras no solicite del sistema operativo un servicio que lo bloquee. Esta solución minimiza el tiempo que gasta el sistema operativo en planificar y activar procesos, pero tiene como inconveniente que un proceso puede monopolizar el procesador (imáginate lo que ocurre si el proceso, por error, entra en un bucle infinito).

En los sistemas con expulsión, el sistema operativo puede quitar a un proceso del estado de ejecución aunque éste no lo solicite. Esta solución permite controlar el tiempo que está en ejecución un proceso, pero requiere que el sistema operativo entre de forma sistemática a ejecutar para así poder comprobar si el proceso ha superado su límite de tiempo de ejecución. Como sabemos, las interrupciones sistemáticas del reloj garantizan que el sistema operativo entre a ejecutar cada pocos milisegundos, pudiendo determinar en estos instantes si ha de producirse un cambio de proceso o no.

Colas de procesos

Para realizar las funciones de planificación, el sistema operativo organiza los procesos listos en una serie de estructuras de información que faciliten la búsqueda del proceso a planificar. Es muy frecuente organizar los procesos en colas de prioridad y de tipo.

La Figura 2.3 muestra un ejemplo con treinta colas para procesos interactivos y dos colas para procesos *batch*. Las treinta colas interactivas permiten ordenar los procesos listos interactivos según treinta niveles de prioridad, siendo, por ejemplo, el nivel 0 el más prioritario. Por su lado, las dos colas *batch* permiten organizar los procesos listos *batch* en dos niveles de prioridad.

Se puede observar en la mencionada figura que se ha incluido una palabra resumen. Esta palabra contiene un 1 si la correspondiente cola tiene procesos y un 0 si está vacía. De esta

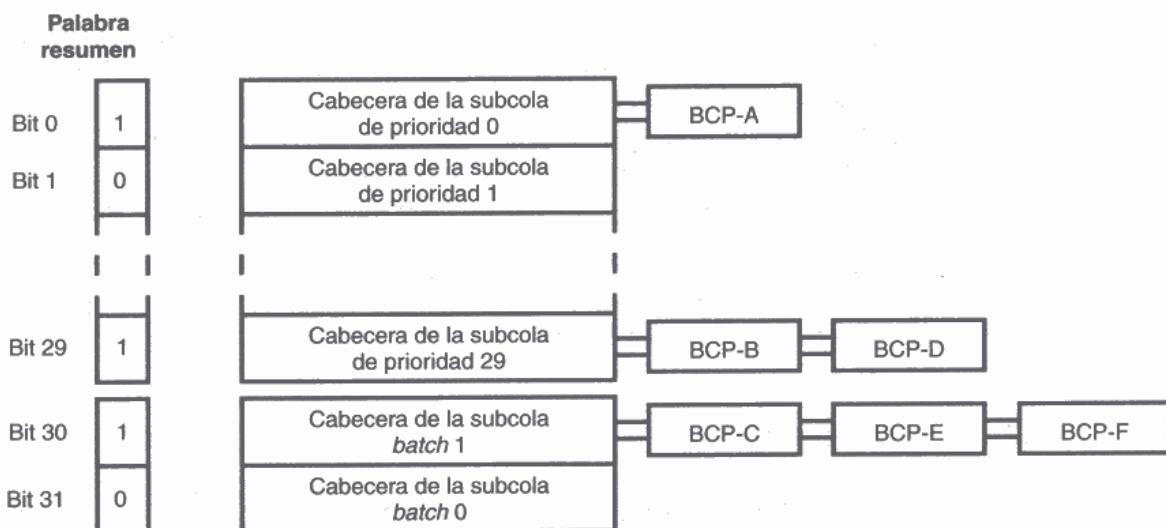


Figura 2.3. Colas de planificación.

forma se acelera el planificador, puesto que puede saber rápidamente dónde encontrará procesos listos.

Objetivos de planificación

El objetivo de la planificación es optimizar el comportamiento del sistema. Ahora bien, el comportamiento de un sistema informático es muy complejo; por tanto, el objetivo de la planificación se deberá centrar en la faceta del comportamiento en el que se esté interesado. Entre los objetivos que se suelen perseguir están los siguientes:

- Reparto equitativo del procesador.
- Eficiencia (optimizar el uso del procesador).
- Menor tiempo de respuesta en uso interactivo.
- Menor tiempo de espera en lotes (*batch*).
- Mayor número de trabajos por unidad de tiempo (*batch*).
- Cumplir los plazos de ejecución de un sistema de tiempo real.

La mayoría de estos objetivos son incompatibles entre sí, por lo que hay que centrar la atención en aquel que sea de mayor interés. Por ejemplo, una planificación que realice un reparto equitativo del procesador no conseguirá optimizar el uso del mismo. Hay que notar que algunos objetivos están dirigidos a procesos interactivos, mientras que otros lo están a procesos *batch*.

Algoritmos de planificación

Los siguientes son algunos de los algoritmos de planificación más usuales:

- **Cíclica o round-robin.** El algoritmo cíclico está diseñado para hacer un reparto equitativo del tiempo del procesador, por lo que está especialmente destinado a los sistemas de tiempo compartido. El algoritmo se basa en el concepto de **rodaja** (*slot*) de tiempo.
- **FIFO.** En este caso, la cola de procesos en estado de listo está ordenada de acuerdo al instante en que los procesos pasan a estado de listo. Los que llevan más tiempo esperando están más cerca de la cabecera.
- **Prioridades.** En el algoritmo de prioridades se selecciona para ejecutar el proceso en estado de listo que tenga la máxima prioridad.

2.1.3. Señales

Las señales tienen frente al proceso el mismo comportamiento que las interrupciones tienen frente al procesador, por lo que se puede decir que una señal es una interrupción al proceso. El proceso que recibe una señal se comporta de la siguiente forma:

- El proceso detiene su ejecución en la instrucción de máquina que está ejecutando.
- Bifurca a ejecutar una rutina de tratamiento de la señal cuyo código ha de formar parte del propio proceso.
- Una vez ejecutada la rutina de tratamiento, sigue la ejecución del proceso en la instrucción en el que fue interrumpido.

El origen de una señal puede ser un proceso o el sistema operativo.

- **Señal proceso → proceso.** Un proceso puede enviar una señal a otro proceso que tenga el mismo identificador de usuario (*uid*), pero no a los que lo tengan distinto. Un proceso también puede mandar una señal a un grupo de procesos que ha de tener su mismo *uid*.
- **Señal sistema operativo → proceso.** El sistema operativo también toma la decisión de enviar señales a los procesos cuando ocurren determinadas condiciones. Por ejemplo, las excepciones de ejecución programa (el desbordamiento en las operaciones aritméticas, la división entre cero, el intento de ejecutar una instrucción con código de operación incorrecto o de direccionar una posición de memoria prohibida) las convierte el sistema operativo en señales al proceso que ha causado la excepción.

Tipos de señales

Dado que las señales se utilizan para indicarle al proceso muchas cosas diferentes, existe una gran variedad de ellas. A título de ejemplo, se incluyen aquí unas pocas categorías de señales

- Excepciones *hardware*.
- Comunicación.
- E/S asíncrona

Efecto de la señal y armado de la misma

Como se ha indicado anteriormente, el efecto de la señal es ejecutar una rutina de tratamiento; pero para que esto sea así, el proceso debe tener armada ese tipo de señal, es decir, ha de estar preparado para recibir dicho tipo de señal.

Armar una señal significa indicarle al sistema operativo el nombre de la rutina del proceso que ha de tratar ese tipo de señal, lo que se consigue en POSIX como se verá con el servicio *sigaction*.

Algunas señales admiten que se las ignore, lo cual ha de ser indicado por el proceso al sistema operativo. En este caso, el sistema operativo simplemente desecha las señales ignoradas por ese proceso. Un proceso también puede enmascarar diversos tipos de señales. El efecto es que las señales enmascaradas quedan bloqueadas (no se desechan), a la espera de que el proceso las desenmascare.

Cuando un proceso recibe una señal sin haberla armado o enmascarado previamente, se produce la acción por defecto, que en la mayoría de los casos consiste en matar al proceso.

2.2. SERVICIOS DE PROCESOS

En esta sección se describen los servicios del sistema operativo necesarios para hacer las prácticas que se proponen en las secciones siguientes. Se describen los principales servicios que ofrece POSIX para la gestión de procesos y procesos ligeros. También se presentan los servicios que permiten trabajar con señales y temporizadores

2.2.1. Servicios POSIX para la gestión de procesos

En esta sección se describen los principales servicios que ofrece POSIX para la gestión de procesos. Estos servicios se han agrupado según las siguientes categorías: identificación de procesos, el entorno de un proceso, creación de procesos y terminación de procesos.

Identificación de procesos

POSIX identifica cada proceso por medio de un entero único denominado *identificador de proceso* de tipo `pid_t`. Los servicios relativos a la identificación de los procesos son los siguientes:

- **Obtener el identificador de proceso.** Este servicio devuelve el identificador del proceso que realiza la llamada. Su prototipo en lenguaje C es el siguiente:

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

- **Obtener el identificador del proceso padre.** Devuelve el identificador del proceso padre. Su prototipo es el que se muestra a continuación:

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

- **Obtener el identificador de usuario real.** Este servicio devuelve el identificador de usuario real del proceso que realiza la llamada. Su prototipo es:

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

- **Obtener el identificador de usuario efectivo.** Devuelve el identificador de usuario efectivo. Su prototipo es:

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

- **Obtener el identificador de grupo real.** Este servicio permite obtener el identificador de grupo real. El prototipo que se utiliza para invocar este servicio es el siguiente:

```
gid_t getgid(void);
```

- **Obtener el identificador de grupo efectivo.** Devuelve el identificador de grupo efectivo. Su prototipo es:

```
gid_t getegid(void);
```

El entorno de un proceso

El entorno de un proceso viene definido por una lista de variables que se pasan al mismo en el momento de comenzar su ejecución. Estas variables se denominan variables de entorno y son accesibles a un proceso a través de la variable externa `environ`, declarada de la siguiente forma:

```
extern char **environ;
```

- **Obtener el valor de una variable de entorno.** El servicio `getenv` permite buscar una determinada variable de entorno dentro de la lista de variables de entorno de un proceso. La sintaxis de esta función es:

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

- **Definir el entorno de un proceso.** El servicio `setenv` permite fijar el entorno de un proceso. La sintaxis de esta función es:

```
char *setenv(char **env);
```

Gestión de procesos

- **Crear un proceso.** La forma de crear un proceso en un sistema operativo que ofrezca la interfaz POSIX es invocando el servicio `fork`. El sistema operativo trata este servicio

realizando una clonación del proceso que lo solicite. El proceso que solicita el servicio se convierte en el proceso padre del nuevo proceso, que es, a su vez, el proceso hijo. El prototipo de esta función es el siguiente:

```
pid_t fork();
```

La llamada devuelve en el proceso hijo un cero, y en el proceso padre, el identificador del proceso hijo.

- **Ejecutar un programa.** El servicio exec de POSIX tiene por objetivo cambiar el programa que está ejecutando un proceso. En POSIX existe una familia de funciones exec cuyos prototipos se muestran a continuación:

```
int exec(char *path, char *arg, ...);
int execv(char *path, char *argv[]);
int execle(char *path, char *arg, ...);
int execve(char *path, char *argv[], char *envp[]);
int execlp(char *file, const char *arg, ...);
int execvp(char *file, char *argv[]);
```

- **Terminar la ejecución de un proceso.** Cuando un programa ejecuta dentro de la función main la sentencia return(valor), ésta es similar a exit(valor). El prototipo de la función exit es:

```
void exit(int status);
```

- **Esperar por la finalización de un proceso hijo.** Permite a un proceso padre esperar hasta que termine la ejecución de un proceso hijo (el proceso padre se queda bloqueado hasta que termina un proceso hijo). Existen dos formas de invocar este servicio:

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

Ambas llamadas esperan la finalización de un proceso hijo y permiten obtener información sobre el estado de terminación del mismo.

2.2.2. Servicios POSIX de gestión de procesos ligeros

En esta sección se describen los principales servicios POSIX relativos a la gestión de procesos ligeros. Estos servicios se han agrupado de acuerdo a las siguientes categorías:

- Atributos de un proceso ligero.
- Creación e identificación de procesos ligeros.
- Terminación de procesos ligeros.

Atributos de un proceso ligero

Cada proceso ligero en POSIX tiene asociado una serie de atributos que representan sus propiedades. Los valores de los diferentes atributos se almacenan en un objeto atributo de tipo `pthread_attr_t`. Existe una serie de servicios que se aplican sobre el tipo anterior y que

permiten modificar los valores asociados a un objeto de tipo atributo. A continuación se describen las principales funciones relacionadas con los atributos de los procesos ligeros.

- **Crear atributos de un proceso ligero.** Este servicio permite iniciar un objeto atributo que se puede utilizar para crear nuevos procesos ligeros. El prototipo de esta función es:

```
int pthread_attr_init(pthread_attr_t *attr);
```

- **Destruir atributos.** Destruye el objeto de tipo atributo pasado como argumento a la misma. Su prototipo es:

```
int pthread_attr_destroy(pthread_attr_t *attr);
```

- **Asignar el tamaño de la pila.** Cada proceso ligero tiene una pila cuyo tamaño se puede establecer mediante esta función cuyo prototipo es el siguiente:

```
int pthread_attr_setstacksize(pthread_attr_t *attr, int stacksize);
```

- **Obtener el tamaño de la pila.** El prototipo del servicio que permite obtener el tamaño de la pila de un proceso es:

```
int pthread_attr_getstacksize(pthread_attr_t *attr, int stacksize);
```

- **Establecer el estado de terminación.** El prototipo de este servicio es:

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
int detachstate);
```

Si el valor del argumento `detachstate` es `PTHREAD_CREATE_DETACHED`, el proceso ligero que se cree con esos atributos se considerará como independiente y liberará sus recursos cuando finalice su ejecución. Si el valor del argumento `detachstate` es `PTHREAD_CREATE_JOINABLE`, el proceso ligero se crea como no independiente y no liberará sus recursos cuando finalice su ejecución. En este caso, es necesario que otro proceso ligero espere por su finalización. Esta espera se consigue mediante el servicio `pthread_join`, que se describirá más adelante.

- **Obtener el estado de terminación.** Permite conocer el estado de terminación que se especifica en un objeto de tipo atributo. Su prototipo es:

```
int pthread_attr_getdetachstate(pthread_attr_t *attr,
int *detachstate);
```

Creación, identificación y terminación de procesos ligeros

Los servicios relacionados con la creación e identificación de procesos ligeros son los siguientes:

- **Crear un proceso ligero.** Este servicio permite crear un nuevo proceso ligero que ejecuta una determinada función. Su prototipo es:

```
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
void * (*start_routine) (void *), void *arg);
```

El primer argumento de la función apunta al identificador del proceso ligero que se crea; este identificador viene determinado por el tipo `pthread_t`. El segundo argumento especifica los atributos de ejecución asociados al nuevo proceso ligero. Si el valor de este segundo argumento es `NULL`, se utilizarán los atributos por defecto, que incluyen la creación del proceso como no independiente. El tercer argumento indica el nombre de la fun-

ción a ejecutar cuando el proceso ligero comienza su ejecución. Esta función requiere un solo parámetro que se especifica con el cuarto argumento, arg.

- **Obtener el identificador de un proceso ligero.** Un proceso ligero puede averiguar su identificador invocando este servicio, cuyo prototipo es el siguiente:

```
pthread_t pthread_self(void)
```

- **Esperar la terminación de un proceso ligero.** Este servicio es similar al wait, pero a diferencia de éste, es necesario especificar el proceso ligero por el que se quiere esperar, que no tiene por qué ser un proceso ligero hijo. El prototipo de la función es:

```
int pthread_join(pthread thid, void **value);
```

La función suspende la ejecución del proceso ligero llamante hasta que el proceso ligero con identificador thid finalice su ejecución. La función devuelve en el segundo argumento el valor que pasa el proceso ligero que finaliza su ejecución en el servicio `pthread_exit`, que se verá a continuación. Únicamente se puede solicitar el servicio `pthread_join` sobre procesos ligeros creados como no independientes.

- **Finalizar la ejecución de un proceso ligero.** Es análogo al servicio `exit` sobre procesos. Su prototipo es:

```
int pthread_exit(void *value)
```

Incluye un puntero a una estructura que es devuelta al proceso ligero que ha ejecutado la correspondiente llamada a `pthread_join`, lo que es mucho más genérico que el parámetro que permite el servicio `wait`.

2.2.3. Servicios POSIX para la gestión de señales y temporizadores

En esta sección se describen los principales servicios POSIX relativos a la gestión de señales. Los más importantes son el envío y la captura de señales.

Algunas señales como SIGSEGV o SIGBUS las genera el sistema operativo cuando ocurren ciertos errores. Otras señales se envían de unos procesos a otros utilizando el siguiente servicio:

- **Enviar una señal.** Permite enviar una señal a un proceso. El prototipo de este servicio en lenguaje C es el siguiente:

```
int kill(pid_t pid, int sig);
```

Envía la señal sig al proceso o grupo de procesos especificado por pid.

- **Armado de una señal.** El servicio que permite armar una señal es:

```
int sigaction(int sig, struct sigaction *act,
struct sigaction *oact);
```

Esta llamada tiene tres parámetros: el número de señal para la que se quiere establecer el manejador, un puntero a una estructura de tipo `struct sigaction` para establecer el nuevo manejador y un puntero a una estructura también del mismo tipo que almacena información sobre el manejador establecido anteriormente.

- **Espera por una señal.** Cuando se quiere esperar la recepción de una señal se utiliza el siguiente servicio `pause`. Este servicio bloquea al proceso que la invoca hasta que llegue una señal. Su prototipo es:

```
int pause(void);
```

Este servicio no permite especificar el tipo de señal por la que se espera; por tanto, cualquier señal no ignorada saca al proceso del estado de bloqueo.

- **Activar un temporizador.** Para activar un temporizador se debe utilizar el servicio:

```
unsigned in alarm(unsigned int seconds);
```

Esta función envía al proceso la señal `SIGALRM` después de pasado el número de segundos especificados en el parámetro `seconds`. Si `seconds` es igual a cero, se cancelará cualquier petición realizada anteriormente.

2.3. PRÁCTICA: MONITORIZACIÓN DE PROCESOS EN WINDOWS CON NTPMON

2.3.1. Objetivos de la práctica

El objetivo de esta práctica es que el alumno se familiarice con una herramienta de monitorización de procesos y de procesos ligeros en Windows. Para ello se usará la herramienta *Ntpmon*, un monitor muy sencillo, pero eficiente, que se puede conseguir de forma gratuita. Lo encontrará en la página web del libro, en el apartado dedicado al material de procesos.

NIVEL: Introducción.

HORAS ESTIMADAS: 4.

2.3.2. Descripción de la funcionalidad que debe desarrollar el alumno

Ntpmon es un monitor de los procesos y los procesos ligeros que se ejecutan en un sistema Windows. Una vez ejecutado, registra todas las operaciones relacionadas con la creación y destrucción de procesos y procesos ligeros. Por ejemplo, cada vez que se arranca el Explorer se genera un proceso y varios procesos ligeros. El resultado es una pantalla como la de la Figura 2.4, en la que, como se puede apreciar, hay varias columnas: `#` indica el número de entrada de monitorización, `CPU` indica el número de segundos de CPU usados por el proceso, `Source` es el proceso que ha ejecutado la acción, `Action` indica la acción ejecutada, `Argument` indica los argumentos que se pasan al proceso y `Elapsed` indica los segundos transcurridos con bloqueo desde que se arrancó el proceso.

La práctica tiene dos partes:

1. **Instalación del monitor Ntpmon** en la máquina de prácticas. El alumno deberá ir a la página web del libro y traer la versión del monitor que corresponda con su sistema operativo Windows NT/2000.
2. **Activación del monitor Ntpmon durante cinco minutos**, captura y elaboración de los datos de salida.

La primera parte de la práctica es muy sencilla, ya que la aplicación se descarga de la web en un archivo comprimido tipo ZIP. Basta con extraer los archivos y ya están listos para funcionar. El monitor es un programa de apenas 50 KB, pero con una gran utilidad para aquellos usuarios interesados en el funcionamiento interno del sistema operativo Windows. Haga doble clic en el archivo `Ntpmon.exe` para ejecutar el programa.

#	CPU	Source	Action	Argument	Elapsed (s)
30	0	IEXPLORE.EXE	Thread Create	TID: 1292	0.100142
31	0	IEXPLORE.EXE	Thread Create	TID: 1812	0.160227
32	0	IEXPLORE.EXE	Thread Create	TID: 1896	0.080113
33	0	IEXPLORE.EXE	Thread Create	TID: 1224	0.050071
34	0	explorer.exe	Thread Create	TID: 1664	0.030043
35	0	svchost.exe	Thread Create	TID: 1324	0.000000
36	0	IEXPLORE.EXE	Thread Create	TID: 1172	0.000000
37	0	IEXPLORE.EXE	Thread Create	TID: 1928	0.010014
38	0	IEXPLORE.EXE	Thread Create	TID: 1380	0.110156
39	0	IEXPLORE.EXE	Thread Create	TID: 892	0.150213
40	0	IEXPLORE.EXE	Thread Create	TID: 1992	0.040057
41	0	IEXPLORE.EXE	Thread Create	TID: 1984	0.140199
42	0	IEXPLORE.EXE	Thread Create	TID: 1876	0.000000
43	0	explorer.exe	Thread Create	TID: 2008	3.054331
44	0	IEXPLORE.EXE	Thread Create	TID: 1696	0.050071
45	0	IEXPLORE.EXE	Thread Create	TID: 1272	1.602272
46	0	IEXPLORE.EXE	Thread Delete	TID: 1272	1.311860
47	0	IEXPLORE.EXE	Thread Delete	TID: 1992	1.041477
48	0	IEXPLORE.EXE	Thread Delete	TID: 1984	0.000000
49	0	IEXPLORE.EXE	Thread Delete	TID: 1696	0.010014
50	0	IEXPLORE.EXE	Thread Delete	TID: 1292	0.000000
51	0	IEXPLORE.EXE	Thread Delete	TID: 1812	0.000000
52	0	IEXPLORE.EXE	Thread Delete	TID: 1896	0.000000
53	0	IEXPLORE.EXE	Thread Delete	TID: 1224	0.000000
54	0	IEXPLORE.EXE	Thread Delete	TID: 1172	0.100142
55	0	IEXPLORE.EXE	Thread Delete	TID: 1928	0.000000
56	0	IEXPLORE.EXE	Thread Delete	TID: 1380	0.000000
57	0	IEXPLORE.EXE	Thread Delete	TID: 892	0.000000
58	0	IEXPLORE.EXE	Thread Delete	TID: 1876	0.000000
59	0	IEXPLORE.EXE	Thread Delete	TID: 1736	0.000000
60	0	explorer.exe	Process Delete	IEXPLORE.EXE	0.000000

Figura 2.4. Pantalla de salida de Ntpmon.

La segunda parte tiene cuatro pasos:

1. **Control de número de operaciones** de archivos que hacen determinadas aplicaciones al abrirse. Ejecute para ello varias aplicaciones. En cada caso anote el número de operaciones antes y después y calcule las operaciones realizadas sobre procesos y procesos ligeros.
2. **Filtrado de operaciones** para estudiar sólo determinados tipos. Para ello, el alumno deberá filtrar las operaciones para que aparezcan sólo las operaciones de creación y destrucción de elementos. El alumno deberá capturar una de estas pantallas con PrintScreen e incluirla en su memoria.
3. **Salvar la monitorización** en un archivo de texto Netmon.log. Ábralo como una hoja de cálculo de Excel y aplique un autofiltro sobre la columna donde se indica el evento. Basándose en el filtrado, el alumno deberá responder a las siguientes preguntas:
 - ¿Cuántas operaciones se han hecho en total? Incluya en su memoria la lista de las operaciones ejecutadas durante la monitorización.
 - Realice un gráfico que muestre todas las operaciones realizadas y el número de las mismas. ¿Qué conclusiones se pueden sacar de este estudio?

2.3.3. Recomendaciones generales

Antes de empezar con la monitorización de aplicaciones complejas, se recomienda monitorizar acciones sencillas. Los mandatos de MS-DOS, como copy, son buenos ejemplos. De esta forma

será capaz de comprender mejor el funcionamiento de *Ntpmon* y de realizar un mejor análisis de los resultados finales.

El monitor tiene una buena herramienta de ayuda en línea, úsela.

2.3.4. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- Memoria.doc: Memoria de la práctica.
- Netmon.log: Archivo que contiene el resultado de la sesión de monitorización.

2.3.5. Bibliografía

- D. Solomon y M. Russinovich. *Inside Windows 2000*, 3.^a edición, 2000.
- R. Nagar. *Windows NT File System Internals*. O'Reilly and Associates, 1997.

2.4. PRÁCTICA: MONITORIZACIÓN DE PROCESOS EN LINUX

2.4.1. Objetivos de la práctica

El objetivo de esta práctica es que el alumno pueda monitorizar los procesos existentes en una máquina Linux y que sea capaz de observar los eventos del sistema significativos para estos objetos. Para ello se usarán dos mandatos de Linux: *ps* y *top*. Ambos mandatos son sencillos, pero muy potentes en cuanto a la monitorización de procesos.

NIVEL: Introducción.

HORAS ESTIMADAS: 6.

2.4.2. Descripción de la funcionalidad que debe desarrollar el alumno

El sistema operativo Linux tiene un sistema de archivos situado en el directorio */proc*. En este directorio se almacena información del sistema y de cada uno de los procesos activos en el mismo. Hay un subdirectorio por cada proceso y dentro del mismo se indica cuánta memoria ha consumido el proceso, los archivos abiertos, sus conexiones, etc. Casi todos los mandatos de Linux relacionados con la monitorización local o remota (*ps*, *top*, *sar*, *vmstat*, etc.) obtienen su información de este directorio.

La práctica tiene dos partes:

Monitorización de los procesos existentes en el sistema

En el caso de la práctica, el alumno debe monitorizar en primer lugar los procesos existentes en el sistema. Para ello se usará el mandato *ps* con distintas opciones:

- ax: muestra todos los procesos activos en el sistema.
- u: muestra la identidad del usuario que creó los procesos.
- f: muestra las relaciones padre-hijo en la jerarquía de procesos.

Cuando se ejecuta el mandato `ps -axuf` se obtiene una pantalla de salida como la que se muestra en la Figura 2.5, donde cada uno de los campos tiene un significado bien preciso: usuario, identificador del proceso, porcentaje de CPU consumida, porcentaje de memoria ocupada, etc. Para averiguar el significado de todas éllas, el alumno puede consultar la ayuda de Linux (`man ps`).

Para probar el efecto de la ejecución del mandato `ps`, repítalo después de ejecutar varias veces un navegador de Internet. A continuación guarde la salida de `ps -axuf` en el archivo `pslog.txt`, estudie la jerarquía de procesos y responda a las siguientes preguntas:

- ¿Cuántos procesos hay en ejecución en el sistema?
- ¿Cuántos usuarios hay conectados al sistema?
- ¿Cuántos procesos son del usuario `root`?
- ¿Cuántos navegadores hay abiertos?
- Describa una jerarquía de procesos de un usuario conectado. ¿Qué `shell` está ejecutando?
- ¿Cuál es el proceso que más tiempo de CPU ha consumido?
- ¿Cuál es el proceso que más espacio de memoria ha consumido?
- ¿Qué procesos llevan más tiempo arrancados?
- ¿En qué fecha y hora arrancó el sistema?

```

1.aguila.arcos.inf.uc3m.es - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
root 9 0.0 0.0 0 0 ? SW May07 0:45 [kupdated]
root 8 0.0 0.0 0 0 ? SW May07 0:02 [bdfflush]
root 7 0.0 0.0 0 0 ? SW May07 0:00 [kreclaimd]
root 6 0.1 0.0 0 0 ? SW May07 16:44 [kswapd]
root 5 0.0 0.0 0 0 ? SWN May07 0:00 [ksoftirqd_CPU0]
root 1 0.0 0.0 1420 80 ? S May07 12:32 init
root 3 0.0 0.0 0 0 ? SW May07 0:00 [keventd]
root 4 40.2 0.0 0 0 ? SW May07 5834:23 [kasm-idled]
root 10 0.0 0.0 0 0 ? SW< May07 0:00 [mdrecoveryd]
root 13 0.0 0.0 0 0 ? SW May07 0:00 [kreiserfsd]
root 91 0.0 0.0 1568 304 ? S May07 0:01 devfsd /dev
root 3919 0.0 0.0 0 0 ? Z May16 0:00 \_ [lp.script <d
root 613 0.0 0.0 0 0 ? SW May07 0:00 [khubd]
rpc 927 0.0 0.0 1600 4 ? S May07 0:00 portmap
root 949 0.0 0.0 1500 184 ? S May07 0:13 syslogd -m 0
root 957 0.0 0.0 1952 172 ? S May07 0:00 klogd -2
rpcuser 982 0.0 0.0 1652 4 ? S May07 0:00 rpc.statd
root 1036 0.0 0.0 2684 300 ? S May07 0:08 /usr/sbin/sshd
root 14959 0.0 0.2 3704 1044 ? S May09 5:07 \_ /usr/sbin/ssh
jcarrete 15000 0.0 0.2 2808 4 ? S May09 0:00 | \_ tcsh -c /
jcarrete 15028 0.0 0.0 3444 4 ? S May09 0:52 | \_ /usr/
root 9794 0.0 0.3 3720 1792 ? S 17:05 0:00 \_ /usr/sbin/ssh
sssoo-va 9796 0.0 0.2 2796 1256 ? S 17:05 0:00 | \_ tcsh -c /
sssoo-va 9852 0.0 0.2 2716 1132 ? S 17:05 0:00 | \_ /usr/
root 12151 0.0 0.3 3812 1988 ? S 22:09 0:00 \_ /usr/sbin/ssh
jcarrete 12152 0.0 0.3 3276 1844 pts/1 S 22:09 0:00 \_ -tcsh
jcarrete 12344 0.0 0.1 2676 732 pts/1 R 22:21 0:00 | \_ ps -a

```

Connected to aguila.arcos.inf.uc3m.es SSH2 - aes128-cbc - hmac-md5 - none 89x28

Figura 2.5. Pantalla de salida de `ps -axuf`.

Monitorización de los procesos que consumen más CPU en el sistema

La segunda parte de la práctica consiste en monitorizar la situación del sistema en un determinado momento y de los procesos que consumen más CPU en un momento dado. Para ello se usará el mandato `top`, que arroja una pantalla de salida como la de la Figura 2.6, y se volcarán los resultados al archivo `toplog.txt`.

Esta segunda parte tiene cuatro partes:

- Control de la situación del sistema.** En la cabecera de salida del mandato `top` se muestran datos de la fecha de monitorización y de la situación del sistema. Désciba en la memoria de la práctica qué indican estos datos. ¿Qué ha hecho la computadora la mayor parte del tiempo?
- Monitorización del efecto de un programa que consume CPU.** Programe una pequeña aplicación en C que incluya un bucle infinito. Ejecútela y observe su efecto en la pantalla de `top`. Incluya el programa de prueba y describa su efecto en la memoria.
- Monitorización del efecto de un programa que consume memoria.** Programe una pequeña aplicación en C que incluya un bucle que consuma mucha memoria con `malloc` y no la libere. Ejecútela y observe su efecto en la pantalla de `top`. Incluya el programa de prueba y describa su efecto en la memoria.
- Monitorización del efecto de un programa que genera entrada.** Programe una pequeña aplicación en C que incluya un bucle que pida un entero por la entrada estándar. Ejecútela y observe su efecto en la pantalla de `top`. Incluya el programa de prueba y describa su efecto en la memoria.

```

10:23pm up 10 days, 1:32, 1 user, load average: 0.22, 0.07, 0.02
96 processes: 94 sleeping, 1 running, 1 zombie, 0 stopped
CPU states: 0.0% user, 0.5% system, 0.0% nice, 99.4% idle
Mem: 513316K av, 504652K used, 3664K free, 76K shrd, 60960K buff
Swap: 996020K av, 86120K used, 909900K free 325440K cached

PID USER      PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
12365 jcarrete 14   0 1052 1052  800 R    0.5  0.2  0:00 top
  1 root      8   0 124   80  80 S    0.0  0.0  12:32 init
  3 root      9   0   0   0  0 SW    0.0  0.0  0:00 keventd
  4 root      9   0   0   0  0 SW    0.0  0.0  5834m kpm-idled
  5 root     19  19   0   0  0 SW    0.0  0.0  0:00 kssoftrqd_CPU0
  6 root      9   0   0   0  0 SW    0.0  0.0  16:44 kswapd
  7 root      9   0   0   0  0 SW    0.0  0.0  0:00 kreclaimd
  8 root      9   0   0   0  0 SW    0.0  0.0  0:02 bdfflush
  9 root      9   0   0   0  0 SW    0.0  0.0  0:45 kupdated
 10 root     -1 -20   0   0  0 SW<  0.0  0.0  0:00 mdrecoveryd
 13 root      9   0   0   0  0 SW    0.0  0.0  0:00 kreiserfsd
 91 root     8   0 336  304 304 S    0.0  0.0  0:01 devfsd
 613 root     9   0   0   0  0 SW    0.0  0.0  0:00 khubd
 927 rpc      9   0 120   4   4 S    0.0  0.0  0:00 portmap
 949 root     9   0 240  184 184 S    0.0  0.0  0:13 syslogd
 957 root     9   0 740  172 172 S    0.0  0.0  0:00 klogd
 982 rpcuser  9   0 108   4   4 S    0.0  0.0  0:00 rpc.statd
1036 root     9   0 468  300 264 S    0.0  0.0  0:08 sshd
1066 root     8   0 468  348 292 S    0.0  0.0  0:03 xinetd
1150 root     9   0  88   4   4 S    0.0  0.0  0:00 rpc.rquotad
1192 root     9   0 284   4   4 S    0.0  0.0  0:00 rpc.mountd

```

Connected to aguila.arcos.inf.uc3m.es SSH2 - aes128-cbc - hmac-md5 - none | 89x28

Figura 2.6. Salida del mandato `top`.

2.4.3. Recomendaciones generales

Antes de empezar con la monitorización de todo el sistema de archivos, se recomienda monitorizar acciones sencillas. La ejecución de mandatos de Linux es un buen ejemplo.

Para conocer el comportamiento de los mandatos propuestos se puede consultar el manual de Linux para los mismos (`man ps` y `man top`, respectivamente).

2.4.4. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- `Memoria.doc`: Memoria de la práctica.
- `pslog.txt`: Archivo que contiene el resultado de la sesión de monitorización con `ps`.
- `toplog.txt`: Archivo que contiene el resultado de la sesión de monitorización con `top`.
- Programas de prueba para `top`: `bucle.c`, `memoria.c` y `entrada.c`, respectivamente.

2.4.5. Bibliografía

- A. Afzal. *Introducción a UNIX*. Prentice-Hall, 1997.
- C. Newham y B. Rosenblatt. *Learning the bash shell*. Sebastopol: O'Reilly, 1995.
- S. R. Bourne. *The UNIX System*. Addison-Wesley, 1983.
- J. Carretero, F. García, J. Fernández y A. Calderón. *Diseño e implementación de programas en lenguaje C*. Madrid, España: Prentice-Hall, 2002.

2.5. PRÁCTICA: DESCUBRIR LA JERARQUÍA DE PROCESOS

2.5.1. Objetivos de la práctica

El objetivo es que el alumno pueda apreciar de forma práctica el carácter jerárquico de los procesos en UNIX. Asimismo, esta práctica permitirá que el alumno se familiarice con la programación de *scripts* en UNIX, puesto que la solución al problema planteado se acometerá usando un *script*.

NIVEL: Introducción.

HORAS ESTIMADAS: 10.

2.5.2. Descripción de la funcionalidad que debe desarrollar el alumno

Se plantea desarrollar un *script*, denominado `arbol_proc`, que, dada una lista de identificadores de proceso recibida como argumento, deberá mostrar por la salida estándar la jerarquía de procesos descendientes de cada uno de los procesos de la lista. A continuación se describen las características que debe incluir el programa:

- El *script* debe utilizar el mandato `ps` para obtener la lista de procesos existentes en ese instante, de manera que pueda determinar los procesos descendientes. Se deberán usar las

opciones del mandato `ps` que permitan obtener toda la jerarquía implicada. Un aspecto importante es que el *script* sólo debe invocar una vez el mandato `ps` durante su ejecución, ya que si se invocase varias veces el estado obtenido podría ser incoherente.

- Con respecto al control de errores, si alguno de los identificadores pasados como argumento no se corresponde con ningún proceso del sistema, el programa continuará su ejecución devolviendo al final un 1. En caso contrario, se devolverá al final un 0.
- El *script* debe ignorar la señal SIGQUIT y terminar ordenadamente cuando recibe la señal SIGINT. La terminación ordenada implica borrar todos los archivos intermedios que haya podido usar este *script* antes de terminar. Dependiendo de cómo se implemente el programa, puede que éste tenga que crear archivos intermedios para guardar información entre diferentes pasos del algoritmo. Si es así, estos archivos se crearán en el directorio desde donde se ejecuta el mandato y nunca deben sobrevivir a la terminación del mismo, sea cual sea la causa de dicha terminación.
- Por lo que se refiere al formato de salida, el programa debe generar una línea en la salida estándar por cada proceso tratado. La primera línea corresponderá con el primer proceso de la lista de argumentos. A continuación se escribirá una línea por cada proceso descendiente de dicho proceso, siguiendo un recorrido del árbol en profundidad (si un proceso tiene un hijo, la línea correspondiente al hijo se generará antes que la del proceso hermano). Una vez terminada toda la jerarquía del primer proceso pasado como argumento, se repetirá el tratamiento con los restantes.

El formato de la línea que se escribe por cada proceso es el siguiente:

```
+...+ pid pid-padre nombre
```

El número de símbolos + se corresponderá con el nivel del proceso dentro de la jerarquía de procesos descendientes, siendo el nivel de los procesos pasados como argumento cero (o sea, no se escribe ningún + en la línea correspondiente a estos procesos). En cuanto al nombre, se podría escribir sólo el nombre del programa que está ejecutando el proceso o incluir también los argumentos del mismo.

Una vez implementado este *script*, se puede plantear que el alumno complete la práctica con la siguiente funcionalidad:

- Añadir un nuevo modo de operación de forma que, en vez de mostrar los procesos descendientes, encuentre los procesos antecedentes de los recibidos como argumentos. Para especificar el modo de operación, en vez de utilizar una opción en la línea de mandatos se usará el propio nombre del *script*. En el caso de que el programa se ejecute con el nombre `arbol_proc`, se comportará como se ha descrito inicialmente, mostrando la jerarquía descendente. Sin embargo, si el nombre es `arbol_proc_asc`, mostrará la ascendente. Hay que resaltar que se trata de un único programa con dos nombres diferentes. En cuanto a las líneas que escribirá el mandato en su modo de operación ascendente serán idénticas al caso anterior, pero sustituyendo el signo + por un -.

2.5.3. Recomendaciones generales

Para realizar el programa se recomienda consultar la página de manual del *shell*, donde se encuentra casi todo lo necesario para realizar la práctica. Asimismo, se debe estudiar en detalle el funcionamiento del mandato `ps`, ya que en él se centra esta práctica.

Se recomienda usar algunas utilidades de depuración que proporciona el *shell*. Por ejemplo, el mandato `set -x` hace que el *shell* imprima los mandatos y sus argumentos según se van ejecutando.

2.5.4. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- `memoria.txt`: memoria de la práctica.
- `arbol_proc`: archivo que contiene el *script*.

2.5.5. Bibliografía

- A. Afzal. *Introducción a UNIX*. Prentice-Hall, 1997.
- C. Newham y B. Rosenblatt. *Learning the bash shell*. Sebastopol: O'Reilly, 1995.
- S. R. Bourne. *The UNIX System*. Addison-Wesley, 1983.

2.6. PRÁCTICA: GESTIÓN DE SEÑALES EN POSIX

2.6.1. Objetivos de la práctica

Esta práctica permite al alumno familiarizarse con los servicios para la gestión de señales en POSIX.

NIVEL: Introducción.

HORAS ESTIMADAS: 4.

2.6.2. Descripción de la funcionalidad que debe desarrollar el alumno

En el material de apoyo de la práctica se proporciona al alumno un archivo objeto que incluye la definición de tres funciones, cuyo prototipo se muestra a continuación:

```
void modulo1(void);
void modulo2(void);
void modulo3(void);
```

La ejecución de cada una de estas funciones puede provocar de forma aleatoria la generación de las siguientes señales: SIGFPE, SIGPIPE, SIGSEGV, SIGBUS y SIGILL. El alumno deberá realizar un programa que calcule la tasa de fallos, es decir, la tasa de señales que se generan, en cada una de las funciones anteriores. Para ello se puede utilizar como base un fragmento de código similar al mostrado a continuación para cada una de las funciones:

```
#define INTENTOS 1000
for(i = 0; i < INTENTOS; i++) {
    // llamada a la función
```

```

modulo_I();

< Contabilizar si se ha generado una señal y de qué tipo >
}
< Imprimir la tasa de señales de cada tipo (porcentaje) que se ha
generado en el bucle anterior >

```

Para alcanzar la funcionalidad descrita en la práctica, el alumno deberá gestionar correctamente las señales que se generan en el programa.

2.6.3. Código fuente de apoyo

Para facilitar la realización de la práctica se dispone del fichero `practica_2.6.tgz`, disponible en la página web del libro, que contiene código fuente de apoyo. Al extraer su contenido se crea el directorio `practica_2.6`, donde se debe desarrollar la práctica. Dentro de este directorio se encuentran los siguientes ficheros:

`Makefile`: archivo fuente para la herramienta `make`. No debe ser modificado. Con él se consigue la recopilación automática de los archivos fuente cuando se modifiquen. Basta con ejecutar el mandato `make` para que el programa se compile de forma automática.

`modulos.o`: archivo objeto que contiene la definición de las funciones: `modulo1`, `modulo2` y `modulo3`.

`modulos.h`: archivo de cabecera que contiene la declaración de las funciones: `modulo1`, `modulo2` y `modulo3`.

2.6.4. Recomendaciones generales

Es importante estudiar el funcionamiento de los servicios que ofrece el estándar POSIX para la gestión de señales. En el apartado 2.2.3 se presenta una breve descripción de los mismos.

2.6.5. Entrega de documentación

El alumno deberá encargarse de entregar los siguientes archivos:

`memoria.txt`: memoria de la práctica

`gestion.c`: código fuente en C con el programa que gestiona las señales que se producen en las funciones `modulo1`, `modulo2` y `modulo3` y que calcula la tasa de errores de las tres funciones individuales.

2.6.6. Bibliografía

- F. García, J. Carretero, J. Fernández y A. Calderón. *Lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.
- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- K. A. Robbins y S. Robbins. *UNIX Programación Práctica*. Prentice-Hall, 1997.

2.7. PRÁCTICA: INTÉPRETE DE MANDATOS SENCILLO

Esta práctica permite al alumno familiarizarse con los servicios para la gestión de procesos que proporciona POSIX. Asimismo, se pretende que conozca cómo es el funcionamiento interno de un intérprete de mandatos en UNIX/Linux.

2.7.1. Objetivos de la práctica

El alumno deberá diseñar y codificar, en lenguaje C y sobre sistema operativo UNIX/Linux, un programa que actúe como intérprete de mandatos sencillo.

Con la realización de este programa, el alumno adquirirá valiosos conocimientos de programación en el entorno UNIX tanto en el uso de las llamadas al sistema operativo (`fork`, `exec`, `signal`, etc.) como en el manejo de herramientas, como el visualizador de páginas de manual `man`, el compilador de C `cc`, el regenerador de programas `make`, etc. El desarrollo de esta práctica se hará de forma incremental. En este capítulo se realizará la funcionalidad necesaria para ejecutar mandatos sencillos, es decir, no se incluye la ejecución de mandatos conectados mediante tuberías. Esta parte se dejará para el Capítulo 4.

Nota. Durante la lectura de este documento encontrará la notación «`man -s# xxxxx`», que sugiere usar el mandato `man` de UNIX/Linux para obtener información sobre el comando `xxxxx` de la sección #.

NIVEL: Avanzado.

HORAS ESTIMADAS: 18.

2.7.2. Descripción de la funcionalidad que debe desarrollar el alumno

El intérprete de mandatos a desarrollar o *minishell* utiliza la entrada estándar (descriptor de archivo 0) para leer las líneas de mandatos que interpreta y ejecuta. Utiliza la salida estándar (descriptor de archivo 1) para presentar el resultado de los mandatos ejecutados. Y utiliza la salida de error estándar (descriptor de archivo 2) para notificar los errores que se puedan producir. Si ocurre un error en alguna llamada al sistema, se utilizará para notificarlo la función de biblioteca `perror`.

Un aspecto importante de todo intérprete de mandatos, incluido el que se debe desarrollar en esta práctica, se centra en cómo obtener la línea de mandatos que introduce el usuario. Para obtener dicha línea se proponen dos posibles alternativas:

1. Que el alumno programe la funcionalidad necesaria para obtener las líneas de mandatos que teclea el usuario.
2. Que el alumno utilice un analizador que se proporciona al alumno en la página web del libro (véase la sección Código fuente de apoyo). En la siguiente sección se describe el empleo de este analizador.

Uso del analizador proporcionado al alumno para obtener la línea de mandatos

Para el desarrollo de esta práctica se proporciona al alumno un *analizador* que permite leer los mandatos introducidos por el usuario. El alumno sólo deberá preocuparse de implementar el intérprete de mandatos. Este analizador es capaz de procesar cualquier línea de mandatos introducida

por el usuario, es decir, mandatos simples, mandatos conectados por tuberías, redirecciones y mandatos en *background*. Las redirecciones se consiguen con las siguientes sintaxis:

```
< archivo Utiliza archivo como entrada estándar del mandato.
> archivo Utiliza archivo como salida estándar del mandato.
>& archivo Usa archivo como salida de error estándar.
```

Para obtener la línea de mandatos tecleada por el usuario debe utilizarse la función `obtener_mandato`, cuyo prototipo se muestra a continuación:

```
int obtener_mandato(char ****args,
                     char **file_in,
                     char **file_out,
                     char **file_err,
                     int *bg);
```

La llamada devuelve 0 en caso de teclear Control-D (EOF), -1 si se encontró un error. Si se ejecuta con éxito, la llamada devuelve el número de mandatos introducido. Es decir:

- para `ls -l` devuelve 1
- para `ls -l | sort | lpr` devuelve 3.

El argumento `args` permite tener acceso a todos los mandatos introducidos por el usuario. Los archivos utilizados para la redirección se pueden obtener a partir de los siguiente argumentos:

- `file_in` apuntará al nombre del archivo a utilizar en la redirección de entrada en caso de que exista o NULL si no hay ninguno.
- `file_out` apuntará al nombre del archivo a utilizar en la redirección de salida en caso de que exista o NULL si no hay ninguno.
- `file_err` apuntará al nombre del archivo a utilizar en la redirección de salida en caso de error que exista o NULL si no hay ninguno.

El argumento `bg` es 1 si el mandato o secuencia de mandatos debe ejecutarse en *background* y 0 en caso contrario.

Si el usuario teclea `ls -l | sort > fichero`. Los argumentos anteriores tendrán la disposición que se muestra en la Figura 2.7.

En el archivo `main.c` (archivo que debe rellenar el alumno con el código del *minishell*) tiene la siguiente estructura:

```
char    ***args;
char    **argv;
char    *file_in;
char    *file_out;
char    *file_err;
char    bg;
int     mandatos;
int     i, j;

while((mandatos = obtener_mandato(&args, &file_in, &file_out,
&file_err, &bg)) != EOF)
{
```

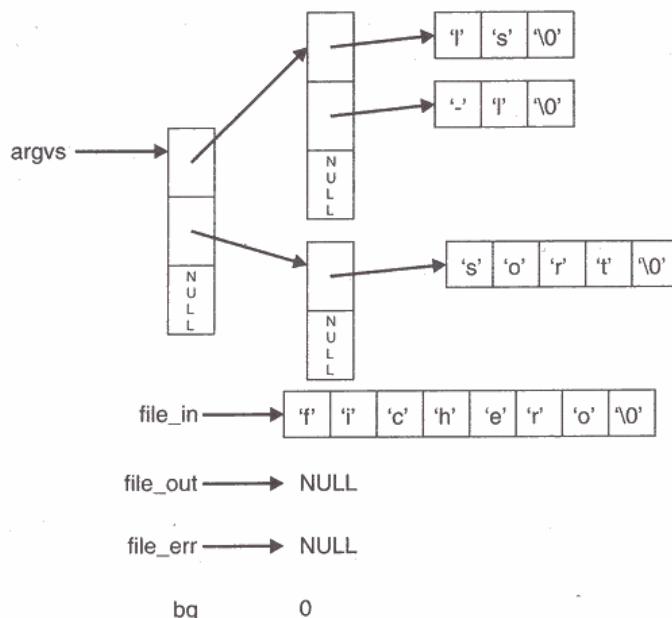


Figura 2.7. Obtención de la línea de mandatos.

```

if (mandatos < 1)
    continue;

for (i = 0; i < mandatos; i++)
{
    argv = args[i];           /* mandatos simple */

    for (j = 0; argv[j] != NULL; j++)
        printf("%s ", argv[j]); /* argumento del
                                   mandato simple */
    printf("\n");
}

if (file_in != NULL)
    printf("< %s\n", file_in); /* redirección de entrada */

if (file_out != NULL)
    printf("> %s\n", file_out); /* redirección de salida */

if (file_err)
    printf(">& %s\n", file_err); /* redirección de salida
                                 de error */

if (bg)
    printf("&\n");             /* mandato en background */
}

```

Se recomienda al alumno que, sin modificar este archivo, compile y ejecute el *minishell* introduciendo diferentes mandatos y secuencias de mandatos para comprender claramente cómo acceder a cada uno de los mandatos de una secuencia.

Desarrollo del *minishell*

Como se ha comentado al principio de la sección, el desarrollo del *minishell* se hará de forma incremental y se completará su funcionalidad en los Capítulos 4 y 5. En este capítulo se desarrollará la siguiente funcionalidad:

1. *Ejecución de mandatos simples* del tipo `ls -l`, `who`, etc. En este caso se asumirá que la función `obtener_mandato`, que se proporciona al alumno, devuelve como valor 1. Asimismo, el argumento a ejecutar se obtendrá en `argvs[0]`. No se contemplará la ejecución de secuencias de mandatos conectados a través de tuberías.
2. *Ejecución de mandatos simples en «background»*. En esta fase se debe incluir la posibilidad de que el usuario pueda ejecutar mandatos en *background* (usando el carácter & al final del mismo), de manera que no tenga que esperar por la finalización del mismo para introducir un nuevo mandato. Por tanto, cuando el *minishell* detecta un mandato en *background* (variable `bg` con valor 1) no deberá esperar la terminación del proceso hijo creado para su ejecución, sino que escribirá inmediatamente el *prompt*, invocando de nuevo a la función `obtener_mandato`. Uno de los aspectos que se tendrá en cuenta en esta versión es el relacionado con el manejo de señales. El proceso correspondiente al propio intérprete, así como los correspondientes a mandatos en *background*, deben ejecutarse con las señales generadas desde el teclado (`SIGINT`, `SIGQUIT`) ignoradas, de forma que no mueran por la generación de las mismas. Por el contrario, los mandatos lanzados en *foreground* deben morir si le llegan estas señales; por tanto, en este caso, la acción para estas señales debe ser la acción tomada por defecto. La ejecución de mandatos en *background* implica que mientras el *minishell* está esperando por la terminación de un mandato que no se ha lanzado en *background*, pueden llegarle notificaciones de la terminación de un mandato previo ejecutado en *background*. En ese caso, el programa imprimirá un mensaje indicando que un mandato en *background* ha terminado (especificando el identificador de proceso) y continuará esperando a la terminación del mandato que no está en *background*.

El *minishell* desarrollado en este capítulo tampoco se preocupará de la redirección de los mandatos a archivos.

2.7.3. Código fuente de apoyo

Para facilitar la realización de esta práctica se dispone del archivo `practica_2.7.tgz`, disponible en la página web del libro, que contiene código fuente de apoyo. Al extraer su contenido, se crea el directorio `practica_2.7`, donde se debe desarrollar la práctica. Dentro de este directorio se encuentran los siguientes archivos:

Makefile: archivo fuente para la herramienta `make`. No debe ser modificado. Con él se consigue la recopilación automática sólo de los archivos fuente que se modifiquen.

Archivos del analizador: archivos que incluyen el código del analizador que se encarga de procesar los mandatos introducidos por el usuario.

main.c: archivo fuente de C que muestra cómo usar el analizador. Este archivo es el que se *debe modificar*. Se recomienda estudiar detalladamente para la correcta comprensión del uso de la función de interfaz, `obtener_mandato`.

2.7.4. ENTREGA DE DOCUMENTACIÓN

El alumno deberá entregar los siguientes archivos:

`memoria.txt`: memoria de la práctica.

`main.c`: código fuente del *minishell*, implementando todas la funcionalidades que se requieren.

2.7.5. Bibliografía

- F. García, J. Carretero, J. Fernández y A. Calderón. *Lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.
- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- K. A. Robbins y S. Robbins. *UNIX Programación Práctica*. Prentice-Hall, 1997.
- M. J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, 1985.

2.8. GESTIÓN DE PROCESOS EN EL MINIKERNEL

En el capítulo anterior se presentaron las características generales del sistema operativo inicial que se recomienda proporcionar al alumno como material de apoyo. En esta sección se mostrará con más detalle cómo se realiza la gestión de procesos en este sistema operativo básico. Asimismo, se presentará una serie de aspectos teóricos requeridos para poder abordar las prácticas de gestión de procesos que se proponen en este capítulo.

2.8.1. Los procesos en el *minikernel*

La versión inicial de este módulo incluye una gestión de procesos básica que corresponde con un sistema monoprogramado. Para ser más precisos, hay que aclarar que en este sistema inicial, aunque se puedan crear y cargar en memoria múltiples programas, el proceso en ejecución continúa hasta que termina, ya sea voluntaria o involuntariamente, debido a una excepción. Obsérvese que ninguna de las tres llamadas al sistema disponibles inicialmente puede causar que el proceso pase a un estado de bloqueado. A continuación se presentan las principales características de la gestión de procesos en este sistema operativo inicial:

- La tabla de procesos (`tabla_procs`) es un vector de tamaño fijo de BCP.
- El BCP contiene los siguientes campos:

```
typedef struct BCP_t {
    int id;           /* ident. del proceso */
    int estado; /* TERMINADO|LISTO|EJECUCION|BLOQUEADO*/
    contexto_t contexto_regs; /* copia de regs. de UCP */
    void * pila; /* dir. inicial de la pila */
    BCPptr siguiente; /* puntero a otro BCP */
    void *info_mem; /* descriptor del mapa de memoria */
} BCP;
```

Evidentemente, el alumno tendrá que incluir nuevos campos en el BCP cuando así lo requiera.

- El BCP dispone de un puntero (*siguiente*) que permite que el sistema operativo construya listas de BCP que tengan alguna relación entre sí. En esta versión inicial sólo aparece una lista de este tipo, la cola de procesos listos (*lista_listos*), que agrupa a todos los procesos listos para ejecutar, incluido el que está ejecutándose actualmente.
- El tipo usado para la cola de listos (tipo *lista_BCP*) permite construir listas con enlace simple, almacenando referencias al primero y al último elemento de la lista. Para facilitar su gestión, se ofrecen funciones que permiten eliminar e insertar BCP en una lista de este tipo. Este tipo puede usarse para otras listas del sistema operativo (por ejemplo, para un semáforo). Hay que resaltar que estas funciones están programadas de manera que cuando se quiere cambiar un BCP de una lista a otra, hay que usar primero la función que elimina el BCP de la lista original y a continuación llamar a la rutina que lo inserta en la lista destino. Asimismo, conviene hacer notar que, por simplicidad, el uso de listas basadas en el tipo *lista_BCP* exige que un BCP no pueda estar en dos listas simultáneamente. Si se quiere plantear un esquema en el que se requiera que un BCP esté en más de una lista, se deberá implementar un esquema de listas alternativo.
- La variable *p_proc_actual* apunta al BCP del proceso en ejecución. Como se comentó previamente, este BCP está incluido en la cola de listos.
- Con respecto a la creación de procesos, la rutina realiza los pasos típicos implicados en la creación de un proceso: buscar una entrada libre, crear el mapa de memoria a partir del ejecutable, reservar la pila del proceso, crear el contexto inicial, llenar el BCP adecuadamente, poner el proceso como listo para ejecutar e insertarlo al final de la cola de listos.
- La liberación de un proceso cuando ha terminado voluntaria o involuntariamente implica liberar sus recursos (imagen de memoria, pila y BCP), invocar al planificador para que elija otro proceso y hacer un cambio de contexto a ese nuevo proceso. Nótese que, dado que no se va a volver a ejecutar este proceso, se especifica un valor nulo en el primer argumento de *cambio_contexto*.
- Por lo que se refiere a la planificación, dado que la versión inicial de este módulo se corresponde con un sistema monoprogramado, el planificador (función *planificador*) no se invoca hasta que termina el proceso actual. El algoritmo que sigue el planificador es de tipo FIFO: simplemente selecciona el proceso que esté primero en la cola de listos. Nótese que si todos los procesos existentes estuviesen bloqueados (situación imposible en la versión inicial), se invocaría la rutina *espera_int*, de la que no se volvería hasta que se produjese una interrupción.
- Hay que resaltar que en este sistema operativo no existe un proceso nulo. Si todos los procesos existentes están bloqueados en un momento dado, lo que no es posible en la versión inicial, es el último en bloquearse el que se queda ejecutando la rutina *espera_int*. Esto puede resultar sorprendente al principio, ya que se da una situación en la que la cola de listos está vacía, pero sigue ejecutando el proceso apuntado por *p_proc_actual*, aunque esté bloqueado. La situación es todavía más chocante cuando termina el proceso actual estando los restantes procesos bloqueados, puesto que en este caso es el proceso que ha terminado el que se queda ejecutando el bucle de la función *planificador* hasta que se desbloquee algún proceso. Obsérvese que alguien tiene que mantener «vivo» al sistema operativo mientras no haya trabajo que hacer.

2.8.2. Consideraciones sobre la implementación de procesos

La implementación de la gestión de procesos presenta numerosos aspectos de bajo nivel relativamente complejos que, además, no suelen tratarse en los libros de propósito general sobre sistemas

operativos. Sin embargo, para acometer prácticas de diseño realistas sobre esta temática hay que tratar con estos aspectos. Esta sección presenta los conceptos que consideramos que se requieren para afrontar las prácticas planteadas.

Sincronización dentro del sistema operativo

El sistema operativo es un programa con un alto grado de concurrencia. Esto puede crear problemas de sincronización muy complejos. Este hecho ha causado que los sistemas operativos sean tradicionalmente un módulo *software* con una tasa de errores apreciable. A continuación se estudian los dos problemas de concurrencia que se presentan típicamente dentro del sistema operativo analizando sus posibles soluciones.

El primer escenario sucede cuando ocurre una interrupción con un nivel de prioridad superior mientras se está ejecutando código del sistema operativo vinculado con el tratamiento de una llamada, una excepción o una interrupción. Se activará la rutina de tratamiento correspondiente, que puede entrar en conflicto con la labor que ha dejado a medias el flujo de ejecución interrumpido. Piense, por ejemplo, en una llamada o rutina de interrupción que manipula la lista de listos que es interrumpida por una interrupción que también modifica esta estructura de datos. Puede ocurrir una condición de carrera que deje corrupta la lista.

La solución habitual es elevar el nivel de interrupción del procesador durante el fragmento correspondiente para evitar la activación de la rutina de interrupción conflictiva. Es importante, y como tal recomendamos que lo tenga en cuenta el instructor a la hora de evaluar las prácticas, intentar elevar el nivel interrupción del procesador justo lo requerido y minimizar el fragmento durante el cual se ha elevado explícitamente dicho nivel. Así, por ejemplo, en un fragmento de código del sistema operativo donde se manipula el *buffer* del terminal, bastaría con inhibir la interrupción del terminal, pudiendo seguir habilitada la interrupción del reloj.

En el segundo escenario sucede que, mientras se está realizando una llamada al sistema, se produce un cambio de contexto a otro proceso (por ejemplo, debido a que se ha terminado la rodaja del proceso actual). Este proceso, a su vez, puede ejecutar una llamada al sistema que entre en conflicto con la llamada previamente interrumpida. Se produce, por tanto, la ejecución concurrente de dos llamadas al sistema. Esta situación puede causar una condición de carrera. Así, por ejemplo, dos llamadas concurrentes que intenten crear un proceso podrían acabar obteniendo el mismo BCP libre. Un poco más adelante, en la sección dedicada a los cambios de contexto involuntarios, se analizará cuál es la solución típica ante este tipo de problemas.

Hay que resaltar que el estudio de los problemas de concurrencia en el segundo caso es más complejo que en el primero. En el primero, el análisis sería el siguiente:

- Hay que estudiar cada parte del código de una llamada al sistema para ver si puede verse afectada por la ejecución de una rutina de interrupción. Si una determinada parte del código de la llamada se ve afectado por la rutina de interrupción de nivel N, durante ese fragmento se elevará el nivel de interrupción del procesador al valor N.
- Se debe hacer un proceso similar con cada rutina de interrupción: se estudia su código y se comprueba si en alguna parte puede verse afectado por una interrupción de nivel superior. En caso afirmativo, se eleva el nivel de interrupción para evitar el problema de condición de carrera.

Así, en el caso del «procesador» del *minikernel*, habría que analizar los siguientes conflictos:

- El código de cada llamada al sistema (y de la rutina de tratamiento de la interrupción *software*) con la rutina del terminal y del reloj.

- El código de la rutina de tratamiento de la interrupción del terminal con la rutina del reloj.
- No habría que analizar posibles conflictos durante la ejecución de la rutina de tratamiento de la interrupción del reloj, ya que tiene máxima prioridad.

En el caso de los problemas de sincronización debido a la ejecución concurrente de llamadas, el análisis de conflictos es mucho más amplio, ya que, en principio, habría que analizar las posibles interferencias entre todas las llamadas al sistema. Dada la dificultad y envergadura del análisis, la solución habitual, como se verá un poco más adelante, es adoptar una solución que resuelve drásticamente este conflicto: impedir que se ejecuten llamadas al sistema concurrentemente.

Los cambios de contexto

Un concepto básico dentro de la gestión de procesos es el de cambio de contexto, o sea, la operación que cambia el proceso asignado al procesador. Es importante distinguir entre dos tipos de cambio de contexto:

- Cambio de contexto voluntario: se produce cuando el proceso en ejecución pasa al estado de bloqueado debido a que tiene que esperar por algún tipo de evento. Sólo pueden ocurrir dentro de una llamada al sistema. No pueden darse nunca en una rutina de interrupción, ya que ésta generalmente no está relacionada con el proceso que está actualmente ejecutando. Obsérvese que el proceso deja el procesador, puesto que no puede continuar.
- Cambio de contexto involuntario: se produce cuando el proceso en ejecución tiene que pasar al estado de listo, ya que debe dejar el procesador por algún motivo (por ejemplo, debido a que se le ha acabado su rodaja de ejecución o porque hay un proceso más urgente listo para ejecutar). Nótese que en este caso el proceso podría seguir ejecutando.

El objetivo de esta sección es mostrar cómo se concretan estos cambios de contexto cuando se está programando la gestión de procesos del sistema operativo.

Cambios de contexto voluntarios

Con respecto a los cambios de contexto voluntarios, el programador del sistema operativo va a realizar una programación que podríamos considerar normal, pero va a incluir llamadas a la rutina `cambio_contexto` cuando así se requiera. Así, por ejemplo, el seudocódigo de una llamada que lee del terminal podría ser como el siguiente:

```

leer() {
    .....
    Si no hay datos disponibles
        proc_anterior = proc_actual;
        proc_anterior-> estado= BLOQUEADO;
        Mover proc_anterior de listos a lista del terminal
        proc_actual = planificador();
        cambio_contexto(proc_anterior, proc_actual);
    .....
}

```

Dado que este código se repite mucho dentro del sistema operativo con la única diferencia de a qué lista se mueve el proceso, normalmente se suele codificar una función para encapsular esta funcionalidad (podría llamarse, por ejemplo, `bloquear`). Nótese que, como se comentó previa-

mente, este tipo de cambios de contexto sólo puede darse dentro de una llamada, por lo que sólo se podrá llamar a `bloquear` desde el código de una llamada. El ejemplo previo quedaría de la siguiente forma:

```
Si no hay datos disponibles
    bloquear();
```

Además, se pueden resaltar los siguientes aspectos sobre los cambios de contexto voluntarios:

- Cuando el proceso por fin vuelve a ejecutar lo hará justo después de la llamada al cambio de contexto.
- Una determinada llamada puede incluir varias llamadas a `bloquear`, puesto que podría tener varias condiciones de bloqueo.
- El código de la función `bloquear` deberá elevar al máximo el nivel de interrupción del procesador, puesto que está modificando la lista de procesos listos y es prácticamente seguro que todas las rutinas de interrupción manipulan esta lista.
- El proceso no volverá a ejecutar hasta que se produzcan dos eventos:
 - En primer lugar, deberá producirse el evento que desbloquee el proceso, incluyéndolo en la lista de listos.
 - En un segundo término, un proceso en ejecución va a hacer un cambio de contexto, voluntario o involuntario, y llama al planificador que, por fin, elige al proceso anterior, que continuará ejecutando justo después de donde se quedó.

Obsérvese que en una operación de cambio de contexto siempre se repite la misma situación: hay un proceso que se queda «congelado» en la llamada y le «pasa el testigo» a otro que se quedó «congelado» previamente en una llamada al cambio de contexto o bien es la primera vez que ejecuta.

De la misma manera que ocurre con la función `bloquear`, habitualmente, las operaciones que hay que llevar a cabo para desbloquear un proceso se suelen encapsular en una función `desbloquear` que generalmente recibe como parámetro la lista donde está el proceso que se quiere desbloquear. Así, en el ejemplo anterior del terminal, la rutina de interrupción del terminal incluiría algo como lo siguiente:

```
int_terminal() {
    .....
    Si hay procesos esperando
        desbloquear(lista del terminal);
    .....
}
```

Nótese que la rutina `desbloquear` podría ser invocada tanto desde una llamada al sistema como desde una interrupción y que, dado que manipula la lista de procesos listos, debería ejecutarse gran parte de la misma con el nivel de interrupción al máximo.

Cambios de contexto involuntarios

Una llamada al sistema o una rutina de interrupción pueden desbloquear a un proceso más importante o pueden indicar que el proceso actual ha terminado su turno de ejecución. Esta situación

puede ocurrir dentro de una ejecución anidada de rutinas de interrupción. Así, por ejemplo, una interrupción del reloj que indica el final del turno de ejecución puede llegar mientras se está tratando una interrupción del terminal que, a su vez, interrumpió la ejecución de una llamada al sistema. Para evitar los problemas de sincronización entre llamadas explicados previamente, la mayoría de los sistemas operativos difieren este cambio de contexto involuntario hasta que termine todo el trabajo del sistema operativo. Por tanto, en el ejemplo planteado, terminaría, en primer lugar, la rutina de interrupción del reloj. A continuación, proseguiría la rutina de interrupción del terminal en el punto donde se quedó. Por último, cuando ésta terminase, se reanudaría la llamada al sistema y, al final de la misma, justo antes de continuar la ejecución del proceso en modo usuario, se produciría el cambio de contexto involuntario. De esta forma se evitan las llamadas al sistema concurrentes.

La cuestión es cómo implementar este esquema de cambio de contexto retardado. La solución más elegante es utilizar la interrupción *software*. Algunos procesadores proporcionan este mecanismo, que consiste en una instrucción especial, que sólo puede ejecutarse en modo privilegiado, que causa una interrupción de mínima prioridad (o sea, una instrucción que tiene un comportamiento similar a la instrucción de TRAP, pero que se usa sólo en modo privilegiado). Como veremos a continuación, las características de este mecanismo le hacen idóneo para realizar el cambio de contexto retardado. Por ello, se ha incluido dentro del *hardware* proporcionado por el *minikernel* (y, por eso, en los procesadores que no disponen de este mecanismo se simula por *software*).

Con la interrupción *software* el cambio de contexto involuntario es casi trivial: cuando dentro del código de una llamada o una interrupción se detecta que hay que realizar, por el motivo que sea, un cambio de contexto involuntario, se activa la interrupción *software*. Dado que se trata de una interrupción del nivel mínimo, su rutina de tratamiento no se activará hasta que el nivel de interrupción del procesador sea 0 (o sea, modo usuario). Si había un anidamiento de rutinas de interrupción, todas ellas habrá terminado antes de activarse la rutina de la interrupción *software*. Esta rutina de tratamiento se encargará de realizar el cambio de contexto involuntario, que se producirá justo cuando se pretendía: en el momento que el sistema operativo ha terminado su trabajo.

Para terminar, hay que comentar un aspecto sutil de los cambios de contexto involuntarios. Supóngase una situación como la siguiente:

- Un proceso invoca una llamada al sistema (como, por ejemplo, una llamada que lee del terminal).
- Mientras se está ejecutando la llamada al sistema, se produce una interrupción que genera un cambio de contexto involuntario activando para ello la interrupción *software*.
- Finaliza la rutina de interrupción reanudándose la llamada interrumpida.
- El mecanismo de interrupción *software* nos asegura que cuando termine esta llamada se realizará el cambio retardado. Sin embargo, puede que esta llamada no termine por el momento debido a que se produce un cambio de contexto voluntario por bloqueo (siguiendo con el ejemplo, se ha detectado que el *buffer* del terminal está vacío). Supongamos que se da esa situación y continúa ejecutando otro proceso justo después de la llamada al cambio de contexto donde se quedó la última vez.
- Este otro proceso termina la llamada al sistema donde se quedó «congelado» en su última ejecución y, justo al retornar de la misma, cuando el nivel de interrupción vuelve a 0, salta la interrupción *software*.

Si se analiza la situación, se puede detectar que no tiene sentido hacer un cambio de contexto involuntario en la rutina de la interrupción *software*, ya que el proceso que debía dejar el procesa-

dor ya lo hizo previamente de forma voluntaria. Dado que en la mayoría de los procesadores (incluido el del *minikernel*), una vez activada la interrupción *software* no se puede desactivar, para evitar esta situación, además de la interrupción *software*, se usa una variable (denominada *need_resched* en Linux) para indicar si hay un cambio de contexto involuntario pendiente:

- Cuando se detecta que hay que realizar el cambio involuntario, además de activar la interrupción *software*, se anota el hecho en la variable.
- Si antes de procesarse la interrupción *software* se realiza un cambio de contexto voluntario, se desactiva la variable.
- La rutina de tratamiento de la interrupción *software* consulta la variable y sólo realiza el cambio de contexto si ésta está activa.

2.9. PRÁCTICA: INCLUSIÓN DE UNA LLAMADA BLOQUEANTE DE TEMPORIZACIÓN EN EL MINIKERNEL

2.9.1. Objetivos de la práctica

Esta práctica pretende incidir en varios aspectos fundamentales dentro del concepto de multiprogramación. Por un lado, el alumno podrá aprender cómo se implementa un cambio de contexto de tipo voluntario, con su correspondiente bloqueo y desbloqueo. Por otro, permite conocer cómo implementar una temporización, lo que podría considerarse también como parte del tema de entrada/salida. Asimismo, la práctica hace que el alumno tome contacto con los problemas de sincronización que existen dentro del sistema operativo debido a la ejecución concurrente de múltiples actividades asíncronas.

NIVEL: Diseño.

HORAS ESTIMADAS: 6.

2.9.2. Descripción de la funcionalidad que debe desarrollar el alumno

Se debe incluir una nueva llamada, denominada *dormir*, que permita que un proceso pueda quedarse bloqueado un plazo de tiempo. Su prototipo es el siguiente:

```
int dormir(unsigned int segundos);
```

El plazo se especifica en segundos como parámetro de la llamada. La inclusión de esta llamada significará que el sistema pasa a ser multiprogramado, ya que cuando un proceso la invoca pasa al estado bloqueado durante el plazo especificado y se deberá asignar el procesador al proceso elegido por el planificador. Nótese que en el sistema sólo existirán cambios de contexto voluntarios y, por tanto, sigue sin ser posible la existencia de llamadas al sistema concurrentes. Sin embargo, dado que la rutina de interrupción del reloj va a manipular listas de BCP, es necesario revisar el código del sistema para detectar posibles problemas de sincronización en el manejo de estas listas y, como se vio en la sección previa, solventarlos elevando el nivel de interrupción en los fragmentos de código correspondientes. Aunque el alumno pueda implementar esta llamada como considere oportuno, a continuación se sugieren algunas pautas:

- Modificar el BCP para incluir algún campo relacionado con esta llamada.
- Definir una lista de procesos esperando plazos.
- Incluir la llamada que, entre otras labores, debe poner al proceso en estado bloqueado, reajustar las listas de BCP correspondientes y realizar el cambio de contexto.
- Añadir a la rutina de interrupción la detección de si se cumple el plazo de algún proceso dormido. Si es así, debe cambiarle de estado y reajustar las listas correspondientes.
- Revisar el código del sistema para detectar posibles problemas de sincronización y solucionarlos adecuadamente.

Dado que se trata de un error muy habitual en esta práctica, se debería comprobar que si dos procesos se duermen «simultáneamente» (las dos llamadas a `dormir` se ejecutan sin que se produzca una interrupción del reloj entre ellas) el mismo plazo de tiempo, se deberán despertar en la misma interrupción del reloj.

2.10. PRÁCTICA: PLANIFICACIÓN *ROUND-ROBIN* EN EL MINIKERNEL

2.10.1. Objetivos de la práctica

Con esta práctica el alumno se verá enfrentado con la problemática asociada a los cambios de contexto involuntarios y aprenderá a cómo usar el mecanismo de la interrupción *software* para resolverlos. Asimismo, seguirá completando la funcionalidad típica asociada al manejador del reloj, añadiéndole funcionalidad relacionada con la planificación.

NIVEL: Diseño.

HORAS ESTIMADAS: 5.

2.10.2. Descripción de la funcionalidad que debe desarrollar el alumno

Se va a sustituir el algoritmo de planificación FIFO presente en la versión inicial del sistema operativo por un algoritmo de tipo *round-robin*, donde el tamaño de la rodaja será igual a la constante `TICKS_POR_RODAJA`. Con la inclusión de este algoritmo, aparecen cambios de contexto involuntarios, lo que causa un gran impacto sobre los problemas de sincronización dentro del sistema al poderse ejecutar varias llamadas de forma concurrente. Para solventar estos problemas, como se vio previamente, no se van a permitir los cambios de contexto involuntarios mientras el proceso está ejecutando en modo sistema. Para lograr este objetivo, la solución planteada se va a basar en el mecanismo de interrupción *software*. Así, en la rutina de tratamiento de la interrupción *software* se realizará el cambio de contexto del proceso actual pasándolo al final de la cola de listos. Además, la implementación del *round-robin* debe cubrir los siguientes aspectos:

- Al asignar el procesador a un proceso, se le debe conceder siempre una rodaja completa, con independencia de si la rodaja previa la consumió completa o no. Por tanto, cuando un proceso se desbloquea y pasa a ejecutar, se le asignará una rodaja completa, no lo que le restaba de la anterior.
- Si un proceso que tiene pendiente un cambio de contexto involuntario se bloquea como parte de la ejecución de una llamada, no se debe aplicar dicho cambio de contexto. Si se cumple la rodaja mientras un proceso está haciendo una llamada al sistema, se activa que hay un cambio de contexto involuntario pendiente; pero si cuando el proceso continúa con

la llamada se queda bloqueado, se deberá desactivar este cambio pendiente para no aplicarlo a otro proceso.

2.11. PRÁCTICA: PLANIFICACIÓN POR PRIORIDADES EN EL MINIKERNEL

2.11.1. Objetivos de la práctica

Con esta práctica el alumno continúa aumentando sus conocimientos sobre cómo programar algoritmos de planificación de procesos, debiéndose enfrentar con nuevos tipos de cambios de contexto involuntarios.

NIVEL: Diseño.

HORAS ESTIMADAS: 6.

2.11.2. Descripción de la funcionalidad que debe desarrollar el alumno

El objetivo de esta práctica es incluir en el *minikernel* un algoritmo de planificación basado en prioridades con las siguientes características:

- Es un esquema de prioridades estáticas de carácter expulsivo. No existe el concepto de rodaja de tiempo y, por tanto, un proceso en ejecución seguirá ejecutando hasta que termine, se bloquee o se desbloquee un proceso con mayor prioridad.
- La prioridad es un valor entero entre 10 (mínima) y 50 (máxima).
- Cada proceso inicialmente tiene asociada una prioridad heredada del padre.
- El proceso `init` tiene inicialmente la prioridad mínima.
- Un proceso puede cambiar su prioridad mediante la llamada:

```
int fijar_prio(unsigned int prio);
```

Devuelve 0 si no hay error y -1 en caso contrario (si el valor `prio` no está en el intervalo de prioridades válidas).

- La prioridad no sólo se usará como criterio de reparto del procesador, sino que también se deberá aplicar cuando varios procesos compiten por un recurso de uso exclusivo. Si hay varios procesos bloqueados esperando para usar un determinado recurso, cuando éste quede libre, deberá obtenerlo el proceso más prioritario. En las prácticas planteadas hasta ahora no se ha producido este tipo de situación de competencia por un recurso. Sin embargo, sí aparecerá en las prácticas propuestas en capítulos posteriores. Así, por ejemplo, puede darse entre procesos bloqueados en un semáforo o en procesos esperando que se tecleen datos en un terminal.

La inclusión de este esquema de prioridades tiene varias repercusiones sobre el código del *minikernel*. Se pueden destacar las dos siguientes:

- El planificador, que hasta ahora era trivial, se complica, ya que debe buscar el proceso con mayor prioridad.
- Aparecen nuevos tipos de cambios de contexto involuntarios. Siempre que se despierte un proceso, ya sea como consecuencia de una interrupción o de una llamada, podría ocurrir que tenga más prioridad que el actual. Puesto que en el *minikernel* no se permiten los cambios de

contexto involuntarios mientras el proceso está ejecutando en modo sistema, cuando se despierte un proceso que tenga mayor prioridad que el actual se anotará que hay cambio de contexto involuntario pendiente y se aplicará el mecanismo de las interrupciones *software*.

2.12. PRÁCTICA: PLANIFICACIÓN TIPO LINUX EN EL MINIKERNEL

2.12.1. Objetivos de la práctica

El objetivo de esta práctica es enfrentar al alumno con un algoritmo de planificación inspirado en uno real, concretamente el de Linux, para que, así, el alumno puede apreciar la complejidad de los algoritmos reales.

NIVEL: Diseño.

HORAS ESTIMADAS: 8.

2.12.2. Descripción de la funcionalidad que debe desarrollar el alumno

A continuación se va a añadir al esquema implementado en la práctica anterior un mecanismo de prioridades dinámicas similar al usado en Linux para los procesos de propósito general. El esquema tendrá las siguientes características:

- La prioridad efectiva de un proceso se calcula a partir de su prioridad base (con las mismas características que en la práctica anterior) y su perfil de ejecución.
- La prioridad efectiva inicial de un proceso es igual a su prioridad base.
- En este esquema existen las rodajas de tiempo, pero su tamaño es proporcional a la prioridad efectiva del proceso (por ejemplo, si un proceso tiene una prioridad efectiva de 40, su rodaja corresponde con 40 interrupciones del reloj).
- En cada interrupción del reloj se decrementa la prioridad efectiva del proceso actual. Si llega a cero, el proceso en ejecución deja el procesador y se selecciona el proceso con mayor prioridad efectiva.
- Cuando la prioridad efectiva de todos los procesos listos para ejecutar sea igual a cero, se realiza un reajuste de las prioridades de **todos los procesos** mediante la siguiente fórmula:

$$\text{prioridad efectiva} = \text{prioridad efectiva} / 2 + \text{prioridad base}$$
- Como ocurría en la práctica anterior, siempre que se despierta un proceso hay que comparar su prioridad efectiva con la del actual y activar un cambio de contexto involuntario en caso de que sea mayor.

2.13. PRÁCTICA: IMPLEMENTACIÓN DE PROCESOS LIGEROS EN EL MINIKERNEL

2.13.1. Objetivos de la práctica

Una de las principales dificultades que encuentra el alumno al enfrentarse con el concepto de proceso ligero (o *thread*) es conseguir distinguir esta nueva entidad del concepto de proceso convencional. Generalmente, al alumno le resulta difícil saber qué información comparten los

procesos ligeros del mismo proceso y cuál es específica de cada proceso ligero. El objetivo es que el alumno pueda resolver de forma práctica estas dudas y, para ello, nada mejor que tener que implementar desde cero un esquema de procesos ligeros sobre un sistema inicial que no los proporciona, como ocurre en la versión inicial del *minikernel*.

NIVEL: Diseño.

HORAS ESTIMADAS: 10.

2.13.2. Descripción de la funcionalidad que debe desarrollar el alumno

La inclusión de procesos ligeros en el *minikernel* sólo requiere incluir una nueva llamada al sistema para crear un proceso ligero, a la que se denominará `crear_thread`, cuyo prototipo será el siguiente:

```
int crear_thread(void *dir_funcion);
```

Esta rutina crea un flujo de ejecución que ejecuta la rutina que empieza en la dirección `dir_funcion`. Devuelve 0 si no hay error y -1 en caso contrario (por ejemplo, si se ha alcanzado el número máximo de procesos ligeros en el sistema).

No va a existir una primitiva específica para terminar un proceso ligero, sino que, por simplicidad, va a usarse la primitiva `terminar_proceso`. El nuevo significado de esta primitiva es el siguiente:

- El sistema operativo llevará la cuenta del número de procesos ligeros que tiene cada proceso (como mínimo 1, el flujo implícito que se crea al llamar a `crear_proceso`).
- Cuando el sistema operativo recibe una llamada `terminar_proceso`, termina la ejecución del proceso ligero actual y decrementa el número de procesos ligeros del proceso. Si este valor llega a cero, el proceso completo termina.
- No es necesario que una función que va a ser ejecutada desde un proceso ligero incluya al final un `terminar_proceso`, ya que el entorno de apoyo se encarga de incluirlo automáticamente.

Un ejemplo del uso de estas primitivas sería el siguiente:

```
#include "servicios.h"

int f(){
    .....
}

inf g(){
    .....
    terminar_proceso(); /* no es necesario incluirlo */
}

int main(){
    .....
    crear_thread(f);
    .....
    crear_thread(g);
    .....
}
```

Algunos aspectos que hay que tener en cuenta a la hora de incluir procesos ligeros son los siguientes:

- Hay que diseñar las estructuras de datos adecuadas para crear esta nueva abstracción. Un posible diseño sería el siguiente:
 - Definir, además del vector de BCP, un vector de procesos ligeros (un vector de descriptores de *thread*, BCT) con un tamaño máximo MAX_THREADS (por ejemplo, 32) que almacenará la información de todos los procesos ligeros existentes en el sistema.
 - Si se considera necesario, los procesos ligeros de un mismo proceso podrían estar enlazados en una lista apuntada desde el BCP.
 - El BCT debe incluir una referencia al BCP correspondiente para poder acceder a la información global del proceso.
 - Será necesario determinar qué aspectos están vinculados al proceso (almacenados en el BCP) y cuáles están relacionados con un proceso ligero (almacenados en el BCT).
- El planificador pasa de planificar procesos convencionales a hacerlo sobre proceso ligeros. Todas las listas de planificación (lista de listos, listas de bloqueados, etc.) son ahora listas de BCT en vez de BCP.
- Es importante que se preste atención a la diferencia que hay entre crear el primer proceso ligero de un proceso, que se crea implícitamente dentro de `crear_proceso`, y crear los sucesivos procesos ligeros del mismo, que se hará en la llamada `crear_thread`.

3

Gestión de memoria

En este capítulo se presentan, en primer lugar, algunos conceptos generales sobre la gestión de memoria. A continuación se plantea una serie de prácticas que intentan que el lector entienda de forma aplicada conceptos estudiados en la teoría de sistemas operativos, tales como el mapa de memoria del proceso, la proyección de archivos en memoria, las bibliotecas dinámicas o la memoria virtual.

3.1. CONCEPTOS BÁSICOS SOBRE GESTIÓN DE MEMORIA

En este apartado se mostrarán, en primer lugar, las distintas fases que conlleva la generación de un ejecutable y se estudiará cómo es el mapa de memoria de un proceso. A continuación, se introducirá brevemente el concepto de memoria virtual. Por último, se presentará el concepto de proyección de archivos y se estudiarán algunos de los servicios POSIX de gestión de memoria.

3.1.1. Modelo de memoria de un proceso

El sistema operativo gestiona el mapa de memoria de un proceso durante la vida del mismo. Dado que el mapa inicial de un proceso está muy vinculado con el archivo que contiene el programa ejecutable asociado al mismo, este apartado comenzará estudiando cómo se genera un archivo ejecutable y cuál es la estructura típica del mismo. A continuación se analizará cómo evoluciona el mapa a partir de ese estado inicial y qué tipos de regiones existen típicamente en el mismo identificando cuáles son sus características básicas.

Fases en la generación de un ejecutable

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

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

Una biblioteca es una colección de objetos normalmente relacionados entre sí. La manera de generar el ejecutable comentado hasta ahora consiste en compilar los módulos fuente de la aplicación y enlazar los módulos objeto resultantes junto con los extraídos de las bibliotecas correspondientes. De esta forma, se podría decir que el ejecutable es autocontenido: incluye todo el código

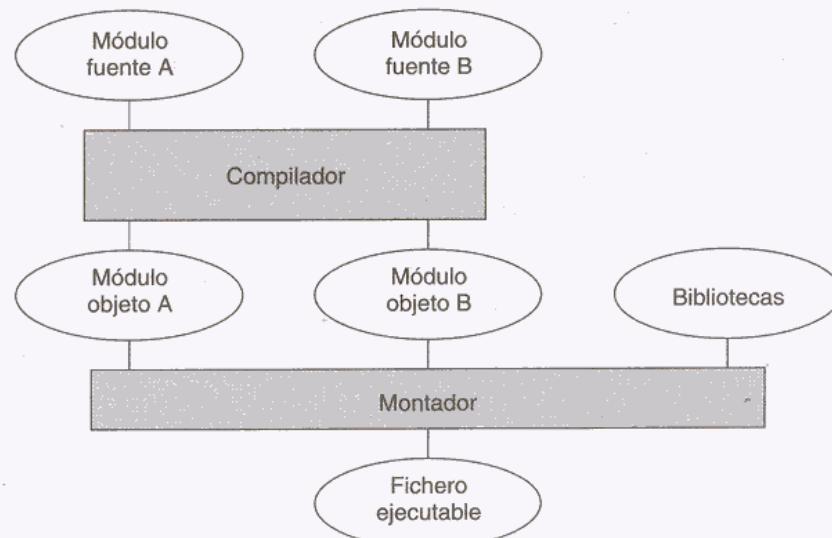


Figura 3.1. Fases en la generación de un ejecutable.

que necesita el programa para poder ejecutarse. Existe, sin embargo, una alternativa que presenta numerosas ventajas: las bibliotecas dinámicas.

Con este nuevo mecanismo, el proceso de montaje de una biblioteca de este tipo se difiere y, en vez de realizarlo en la fase de montaje, se realiza en tiempo de ejecución del programa. Cuando en la fase de montaje el montador procesa una biblioteca dinámica, no incluye en el ejecutable código extraído de la misma, sino que simplemente anota en el ejecutable el nombre de la biblioteca para que ésta sea cargada y enlazada en tiempo de ejecución. El uso de bibliotecas dinámicas presenta múltiples ventajas: se reduce el tamaño de los ejecutables, se favorece el compartimiento de información y se facilita la actualización de la biblioteca.

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

Formato del ejecutable

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

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

- Código (texto): contiene el código del programa.
- Datos con valor inicial: almacena el valor inicial de todas las variables globales a las que se les ha asignado un valor inicial en el programa.

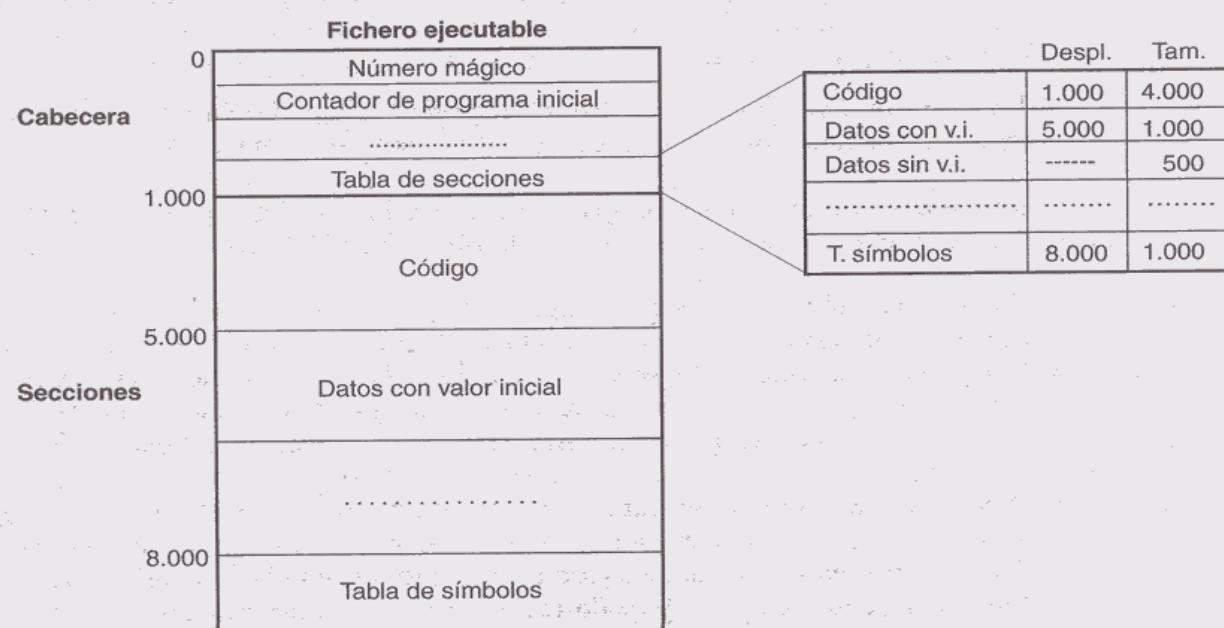


Figura 3.2. Formato simplificado de un ejecutable.

- Datos sin valor inicial. Se corresponde con todas las variables globales a las que no se les ha dado un valor inicial. Como muestra la figura, este apartado aparece descrito en la tabla de secciones de la cabecera, pero, sin embargo, no se almacena normalmente en el ejecutable, ya que el contenido de la misma es irrelevante.

Mapa de memoria de un proceso

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

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

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

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

- *Código* (o texto). Se trata de una región compartida de lectura/ejecución. Es de tamaño fijo. El soporte de esta región está en el apartado correspondiente del ejecutable.
- *Datos con valor inicial*. Se trata de una región privada, ya que cada proceso que ejecuta un determinado programa necesita una copia propia de las variables del mismo. Es de lectura/escritura y de tamaño fijo. El soporte de esta región está en el apartado correspondiente del ejecutable.
- *Datos sin valor inicial*. Se trata de una región privada, de lectura/escritura y de tamaño fijo (el indicado en la cabecera del ejecutable). Como se comentó previamente, esta región no tiene soporte en el ejecutable, ya que su contenido inicial es irrelevante.
- *Pila*. Esta región es privada y de lectura/escritura. Servirá de soporte para almacenar los registros de activación de las llamadas a funciones (las variables locales, parámetros, dirección de retorno, etc.). Se trata, por tanto, de una región de tamaño variable que crecerá

cuando se produzcan llamadas a funciones y decrecerá cuando se retorne de las mismas. Típicamente, esta región crece hacia las direcciones más bajas del mapa de memoria. En el mapa inicial existe ya esta región que contiene típicamente los argumentos especificados en la invocación del programa.

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

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

En la Figura 3.3 se muestra un hipotético mapa de memoria que contiene algunos de los tipos de regiones comentados en este apartado.

Como puede apreciarse en la figura anterior, la carga de una biblioteca dinámica implicará la creación de un conjunto de regiones asociadas a la misma que contendrán las distintas secciones de la biblioteca (código y datos globales).

3.1.2. Memoria virtual

En prácticamente todos los sistemas operativos modernos se usa la técnica de memoria virtual. En este apartado se analizarán muy brevemente los conceptos básicos de esta técnica.

La memoria en un sistema está organizada como una jerarquía de niveles de almacenamiento entre los que se mueve la información dependiendo de la necesidad de la misma en un determinado instante. La técnica de memoria virtual se ocupa de la transferencia de información entre la memoria principal y la secundaria. La memoria secundaria está normalmente soportada en un disco (o partición) que, dado que se implementa sobre un esquema de paginación, se le denominado dispositivo de paginación o de *swap*.

Un esquema de paginación implica que el mapa de memoria de cada proceso se considera dividido en **páginas**. A su vez, la memoria principal del sistema se considera dividida en zonas del mismo tamaño, que se denominan **marcos de página**. Un marco de página contendrá en un determinado instante una página de memoria de un proceso. La estructura de datos que relaciona cada

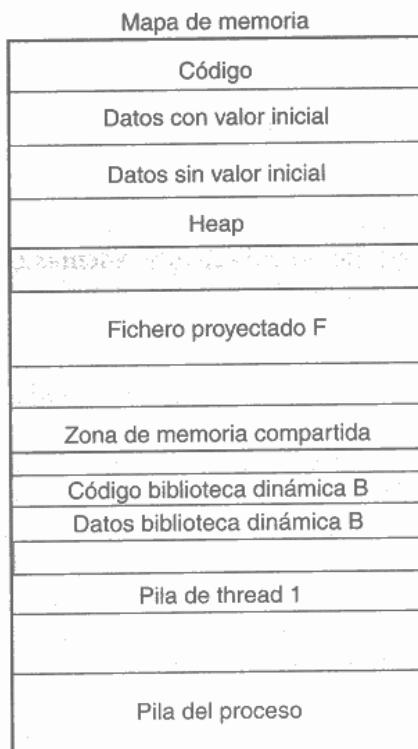


Figura 3.3. Mapa de memoria de un proceso hipotético.

página con el marco donde está almacenada es la **tabla de páginas**. El *hardware* de gestión de memoria (MMU, *Memory Management Unit*) usa esta tabla para traducir todas las direcciones que genera un programa. La Figura 3.4 muestra cómo es este esquema de traducción.

Cada entrada de la tabla de páginas, además del número de marco que corresponde con esa página, contiene información adicional, tal como la siguiente:

- **Información de protección.** Un conjunto de bits que especifican qué tipo de accesos están permitidos. Típicamente, se controla el acceso de lectura, de ejecución y de escritura.
- **Indicación de página válida.** Un bit que especifica si esa página es válida, o sea, tiene una traducción asociada.
- **Indicación de página accedida.** La MMU activa este bit indicador cuando se accede a una dirección lógica que pertenece a esa página.
- **Indicación de página modificada.** La MMU activa este bit indicador cuando se escribe en una dirección lógica que pertenece a esa página.

Además de las tablas de páginas, el sistema operativo debe usar una estructura para almacenar el estado de ocupación de la memoria principal. Se trata de la **tabla de marcos de página** que permite conocer qué marcos están libres y cuáles están ocupados.

Por último, será necesario que el sistema operativo almacene por cada proceso su **tabla de regiones** que contenga las características de cada región especificando qué rango de páginas pertenecen a la misma.

La memoria virtual se construye generalmente sobre un esquema de paginación por demanda. Cuando un proceso necesita acceder a una página que no está en memoria principal (a lo que se

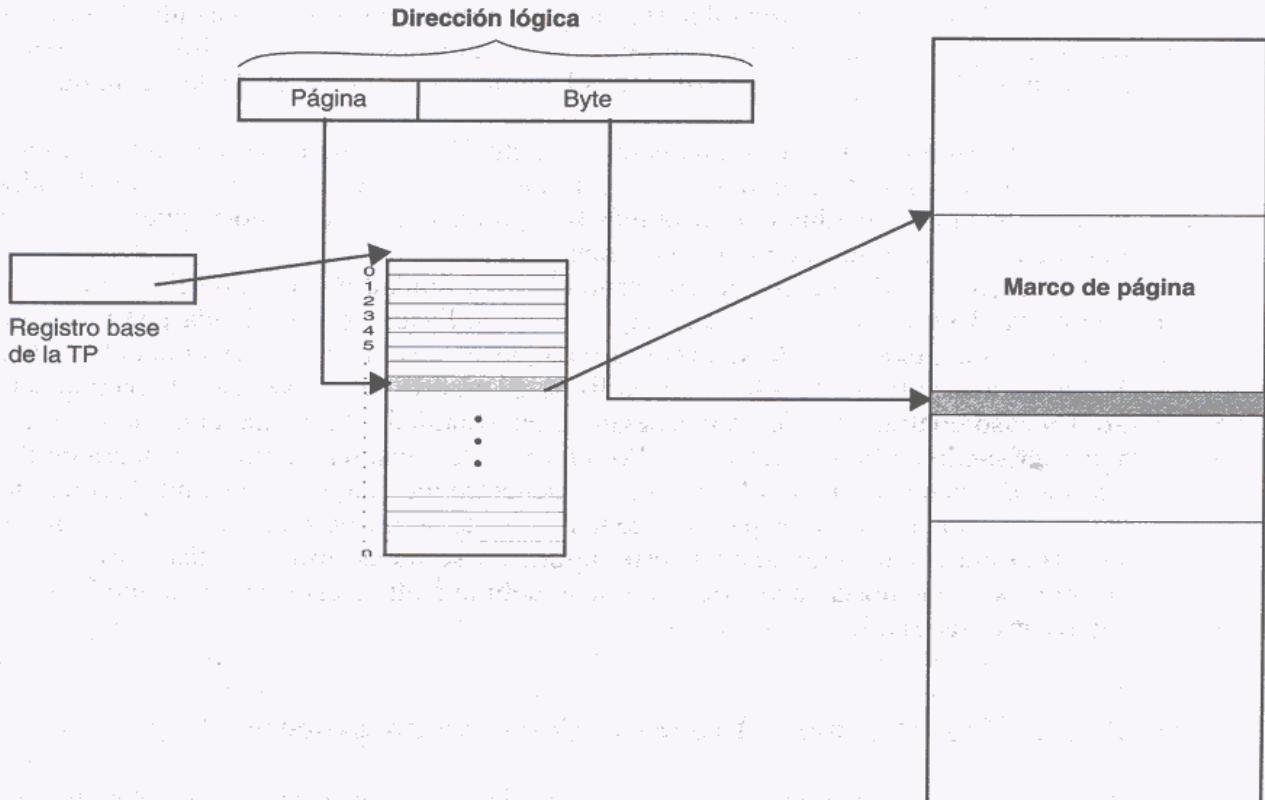


Figura 3.4. Esquema de traducción de la paginación.

denomina **fallo de página**), el sistema operativo se encarga de transferirla desde la memoria secundaria. Si al intentar traer la página desde la memoria secundaria se detecta que no hay espacio en la memoria principal (no hay marcos libres), será necesario expulsar una página de la memoria principal y transferirla a la secundaria. Por tanto, las transferencias desde la memoria principal hacia la secundaria se realizan normalmente por expulsión. El algoritmo para elegir qué página debe ser expulsada se denomina algoritmo de reemplazo y se analizará más adelante. A continuación se especifican los pasos típicos en el tratamiento de un fallo de página:

- La MMU produce una excepción y típicamente deja en un registro especial la dirección que causó el fallo.
- Se activa el sistema operativo que comprueba si se trata de una dirección correspondiente a una página realmente inválida o se corresponde con una página ausente de memoria. Si la página es inválida, se aborta el proceso o se le manda una señal. En caso contrario, se realizan los pasos que se describen a continuación.
- Se consulta la tabla de marcos para buscar uno libre.
- Si no hay un marco libre, se aplica el algoritmo de reemplazo para seleccionar una página para expulsar. El marco seleccionado se desconectará de la página a la que esté asociado poniendo como inválida la entrada correspondiente. Si la página está modificada, previamente hay que escribir su contenido a la memoria secundaria.
- Una vez que se obtiene el marco libre, ya sea directamente o después de una expulsión, se inicia la lectura de la nueva página sobre el marco y al terminar la operación se rellena la entrada correspondiente a la página para que esté marcada como válida y apunte al marco utilizado.

La política de reemplazo determina qué página debe ser desplazada de la memoria principal para dejar sitio a la página entrante. A continuación se describirán brevemente dos de los algoritmos de reemplazo más típicos: el algoritmo FIFO y el de la segunda oportunidad (o reloj).

El algoritmo FIFO selecciona para la sustitución la página que lleva más tiempo en memoria. La implementación de este algoritmo es simple. Sin embargo, el rendimiento del algoritmo no es siempre bueno. La página que lleva más tiempo residente en memoria puede contener instrucciones o datos que se acceden con frecuencia.

El algoritmo de reemplazo con segunda oportunidad es una modificación sencilla del FIFO que evita el problema de que una página muy utilizada sea eliminada por llevar mucho tiempo residente. Para ello, cuando se necesita reemplazar una página, se examina el bit de referencia de la página más antigua (la primera de la lista). Si no está activo, se usa esta página para el reemplazo. En caso contrario, se le da una segunda oportunidad a la página poniéndola al final de la lista y desactivando su bit de referencia. Por tanto, se la considera como si acabara de llegar a memoria. La búsqueda continuará hasta que se encuentre una página con su bit de referencia desactivado. Nótese que si todas las páginas tienen activado su bit de referencia, el algoritmo degenera en un FIFO puro. Para implementar este algoritmo se puede usar una lista circular de las páginas residentes en memoria en vez de una lineal, debido a ello a esta estrategia también se le denomina **algoritmo del reloj**.

3.1.3. Operaciones sobre las regiones de un proceso

A continuación se analiza cómo se realizan las diversas operaciones sobre las regiones en un sistema con memoria virtual.

Creación de una región

Cuando se crea una región, no se le asigna memoria principal, puesto que se cargará por demanda. Todas las páginas de la región se marcan como no residentes. De esta forma, el primer acceso causará un fallo de página.

El sistema operativo actualiza la tabla de regiones para reflejar la existencia de la nueva región y guarda la información de las características de las páginas de la región llenando las entradas de la tabla de páginas de acuerdo a las mismas. Algunas de estas características son relevantes a la MMU, como, por ejemplo, la protección. Sin embargo, otras sólo le conciernen al sistema operativo, como, por ejemplo, si las páginas de la región son privadas o compartidas.

Las páginas de una región con soporte en un archivo (por ejemplo, las de código o las correspondientes a la región de datos con valor inicial) se marcan para indicar esta situación (**bit de cargar de archivo**), almacenándose también la dirección del bloque correspondiente del archivo. Las páginas sin soporte (por ejemplo, las páginas correspondientes a la región de datos sin valor inicial) se marcan con un valor especial que indica que hay que llenarlas a cero (**bit de llenar con ceros**). Nótese que un fallo sobre una página de este tipo no provocaría una operación de entrada/salida al disco.

El caso de la creación de la región de pila del proceso es un poco diferente, ya que esta región tiene un contenido previo (los argumentos del programa) que no está almacenado en un archivo. Típicamente, se reserva uno o más bloques en el *swap* y se copia en ellos el contenido inicial de la pila. También se puede crear la pila inicial en memoria principal.

Una vez creada una región, el tratamiento que se le da a la página cuando se expulsa y está modificada va a depender de si es privada o compartida. Si la región es privada, se escribe en el

swap. Sin embargo, si es compartida, se escribe directamente en el soporte para que todos los procesos puedan ver las modificaciones.

En la creación de la imagen inicial del proceso se crean todas las regiones iniciales siguiendo el procedimiento que se acaba de describir y se marcan los huecos como páginas inválidas. En la Figura 3.5 se muestra cómo es la imagen de memoria inicial del proceso en un sistema con memoria virtual.

Liberación de una región

Cuando se libera una región, se debe actualizar la tabla de regiones para reflejar este cambio.. Las páginas asociadas a la región hay que marcarlas como inválidas tanto para la MMU como para el sistema operativo. Además, si se trata de una región privada, se liberan los marcos de memoria que puedan contener información de la misma. La liberación puede deberse a una solicitud explícita (como ocurre cuando se desprojeta un archivo) o a la finalización del proceso que conlleva la liberación de todas sus regiones

Cambio del tamaño de una región

Con respecto a una disminución de tamaño en una región, esta operación implica una serie de acciones similares a la liberación, pero que sólo afectan a la zona liberada. Por lo que se refiere a un aumento de tamaño, hay que comprobar que la región no se solapa con otra región y establecer las nuevas páginas como no residentes y con las mismas características que el resto de las páginas de la región.

Duplicado de una región

Esta operación está asociada al servicio `fork` de POSIX y no se requiere en sistemas operativos que tengan un modelo de creación de procesos más convencional. Cuando se produce una llamada a este servicio, se debe crear un nuevo proceso que sea un duplicado del proceso que la invoca. Ambos procesos compartirán las regiones de carácter compartido que hay en el mapa del proceso original. Sin embargo, las regiones de carácter privado deben ser un duplicado de las regiones originales. Esta operación de duplicado es costosa, ya que implica crear una nueva región y copiar su contenido desde la región original. Para agilizar esta operación, la mayoría de las versiones de UNIX ha optimizado esta operación de duplicado usando la técnica del *copy-on-write* (**COW**), que intenta realizar un duplicado por demanda. Sólo se copia una página de la región cuando uno de los procesos la intenta modificar mientras se comparte.

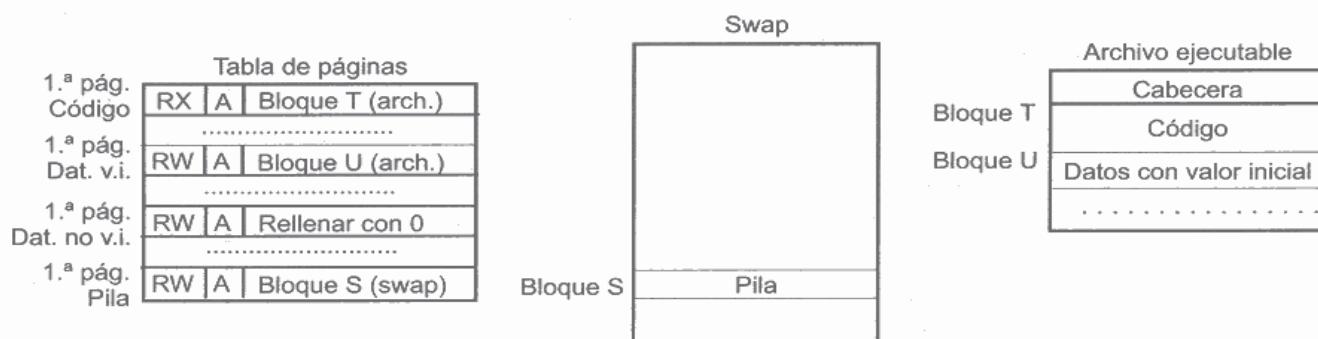


Figura 3.5. Estado inicial de ejecución de un programa en un sistema con memoria virtual.

3.1.4. Proyección de archivos en memoria

La generalización de la técnica de memoria virtual permite ofrecer a los usuarios una forma alternativa de acceder a los archivos. El sistema operativo va a permitir que un programa solicite que se haga corresponder una zona de su mapa de memoria con los bloques de un archivo cualquiera, ya sea completo o una parte del mismo. En la solicitud, el programa especifica el tipo de protección asociada a la región. Como se puede observar en la Figura 3.6, el sistema operativo deberá manipular la tabla de páginas del proceso para que se corresponda con los bloques del archivo proyectado.

Una vez que el archivo está proyectado, si el programa accede a una dirección de memoria perteneciente a la región asociada al archivo, estará accediendo al archivo. El programa ya no tiene que usar los servicios del sistema operativo para leer y escribir en el archivo. El propio mecanismo de memoria virtual será el que se encargue de ir trayendo a memoria principal los bloques del archivo cuando se produzca un fallo de página al intentar acceder a la región asociada al mismo y de escribirlos cuando la página sea expulsada estando modificada.

El acceso a un archivo mediante su proyección en memoria presenta numerosas ventajas sobre el acceso convencional basado en los servicios de lectura y escritura, puesto que se disminuye considerablemente el número de llamadas al sistema necesarias para acceder a un archivo, se evitan copias intermedias de la información y se facilita la forma de programar los accesos a los archivos.

3.1.5. Servicios de gestión de memoria

El estándar POSIX define servicios de gestión de memoria para realizar la proyección y desproyección de archivos (`mmap`, `munmap`). El servicio `mmap` tiene el siguiente prototipo:

```
caddr_t mmap (caddr_t direc, size_t longitud, int protec,
              int indicador, int descriptor, off_t despl);
```

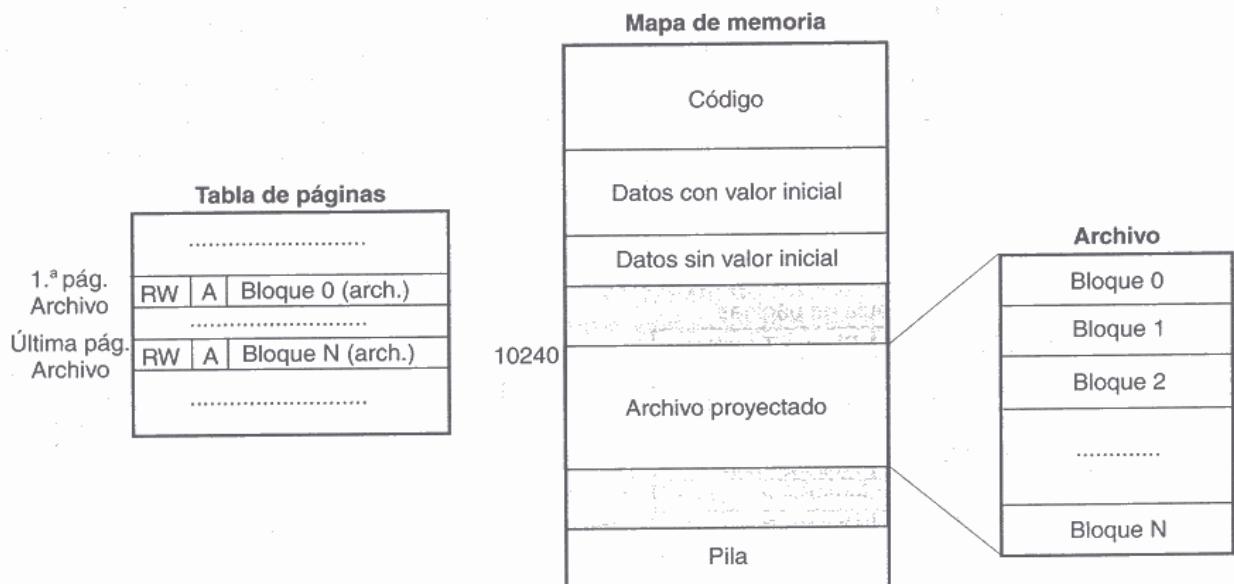


Figura 3.6. Proyección de un archivo.

El primer parámetro indica la dirección del mapa donde se quiere que se proyecte el archivo. Generalmente, se especifica un valor nulo para indicar que se prefiere que sea el sistema el que decida dónde proyectar el archivo. En cualquier caso, la función devolverá la dirección de proyección utilizada.

El parámetro descriptor se corresponde con el descriptor del archivo que se pretende proyectar (que debe estar previamente abierto) y los parámetros despl y longitud establecen qué zona del archivo se proyecta: desde la posición despl hasta despl + longitud.

El argumento protec establece la protección sobre la región, que puede ser de lectura (PROT_READ), de escritura (PROT_WRITE), de ejecución (PROT_EXEC) o cualquier combinación de ellas. Esta protección debe ser compatible con el modo de apertura del archivo. Por último, el parámetro indicador permite establecer ciertas propiedades en la región:

- MAP_SHARED. La región es compartida. Las modificaciones sobre la región afectarán al archivo. Un proceso hijo compartirá esta región con el padre.
- MAP_PRIVATE. La región es privada. Las modificaciones sobre la región no afectarán al archivo. Un proceso hijo no compartirá esta región con el padre, sino que obtendrá un duplicado de la misma.
- MAP_FIXED. El archivo debe proyectarse justo en la dirección especificada en el primer parámetro siempre que éste sea distinto de cero.

En el caso de que se quiera proyectar una región sin soporte (región anónima), en algunos sistemas se puede especificar el valor MAP_ANON en el parámetro indicador. Otros sistemas UNIX no ofrecen esta opción, pero permiten proyectar el dispositivo /dev/zero para lograr el mismo objetivo.

Cuando se quiere eliminar una proyección previa o parte de la misma se usa el servicio munmap, cuyo prototipo es:

```
int munmap (caddr_t direc, size_t longitud);
```

Los parámetros direc y longitud definen una región (o parte de una región) que se quiere desproyectar.

Por lo que se refiere a la carga explícita de bibliotecas dinámicas, la mayoría de los sistemas UNIX ofrece las primitivas dlopen, dlsym y dlclose, cuyos prototipos son los siguientes:

```
void *dlopen (const char *biblio, int modo);
void *dlsym(void *descriptor, char *simbolo);
int dlclose (void *descriptor);
```

La rutina dlopen recibe como argumentos el nombre de la biblioteca y un valor que especifica cómo se desea que se realice la carga de la biblioteca y devuelve un descriptor que identifica a dicha biblioteca cargada. Por lo que se refiere al modo, aunque permite indicar distintas posibilidades a la hora de cargarse la biblioteca de forma obligatoria, sólo es necesario especificar uno de los dos siguientes valores: RTLD_LAZY, que indica que las referencias a símbolos que estén pendientes de resolver dentro de la biblioteca no se llevarán a cabo hasta que sea estrictamente necesario, o RTLD_NOW, que especifica que durante la propia llamada dlopen se resuelvan todas las referencias pendientes que haya dentro de la biblioteca que se desea cargar.

La función dlsym recibe como parámetros el descriptor de una biblioteca dinámica previamente cargada y el nombre de un símbolo (una variable o una función). Esta función busca ese símbolo dentro de la biblioteca especificada y devuelve la dirección de memoria donde se encuentra dicho símbolo. Como primer parámetro, en vez de un descriptor de biblioteca, se puede espe-

cificar la constante RTLD_NEXT. Si desde una biblioteca dinámica se invoca a la función `dlsym` especificando esa constante, el símbolo será buscado en las siguientes bibliotecas dinámicas del proceso a partir de la propia biblioteca que ha invocado la función `dlsym`. Como se verá en una de las prácticas propuestas en este capítulo, este mecanismo suele utilizarse para interponer una biblioteca dinámica.

La rutina `dlclose` descarga la biblioteca especificada por el descriptor.

3.2. PRÁCTICA: ANÁLISIS DEL MAPA DE MEMORIA DE LOS PROCESOS

3.2.1. Objetivos de la práctica

El objetivo principal es entender de forma aplicada cómo está organizado el mapa de memoria de un proceso y cómo evoluciona durante la ejecución del mismo. Para ello se accederá al archivo `/proc/pid/maps` de Linux.

NIVEL: Introducción.

HORAS ESTIMADAS: 8.

3.2.2. El archivo `/proc/pid/maps`

El sistema operativo Linux ofrece un tipo de sistema de archivos muy especial: el sistema de archivos `proc`. Este sistema de archivos no tiene soporte en ningún dispositivo. Su objetivo es poner a disposición del usuario datos del estado del sistema en la forma de archivos. Esta idea no es original de Linux, ya que casi todos los sistemas UNIX la incluyen. Sin embargo, Linux se caracteriza por ofrecer más información del sistema que el resto de variedades de UNIX. En este sistema de archivos se puede acceder a información general sobre características y estadísticas del sistema, así como a información sobre los distintos procesos existentes. La información relacionada con un determinado proceso se encuentra en un directorio que tiene como nombre el propio identificador del proceso (*pid*). Así, si se pretende obtener información de un proceso que tiene un identificador igual a 1234, habrá que acceder a los archivos almacenados en el directorio `/proc/1234/`. Para facilitar el acceso de un proceso a su propia información, existe, además, un directorio especial, denominado `self`. Realmente, se trata de un enlace simbólico al directorio correspondiente a dicho proceso. Así, por ejemplo, si el proceso con identificador igual a 2345 accede al directorio `/proc/self/`, está accediendo realmente al directorio `/proc/2345/`.

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

08048000-0804a000	r-xp	00000000	08:01	65455	/bin/cat
0804a000-0804c000	rw-p	00001000	08:01	65455	/bin/cat
0804c000-0804e000	rwxp	00000000	00:00	0	
40000000-40013000	r-xp	00000000	08:01	163581	/lib/ld-2.2.5.so
40013000-40014000	rw-p	00013000	08:01	163581	/lib/ld-2.2.5.so

```

40022000-40135000 r-xp 00000000 08:01 165143 ... /lib/libc-2.2.5.so
40135000-4013b000 rw-p 00113000 08:01 165143 ... /lib/libc-2.2.5.so
4013b000-4013f000 rw-p 00000000 00:00 0 ...
bffffe000-c0000000 rwxp ffffff000 00:00 0 ...

```

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

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

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

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

3.2.3. Descripción de la funcionalidad que debe desarrollar el alumno

La práctica consiste en desarrollar una serie de programas simples que realice labores que afecten al mapa de memoria para que el alumno analice cómo evoluciona el mapa debido a estas acciones. Para ello, se incluirá en los puntos de interés de cada programa la impresión del mapa de memoria del proceso en ese instante. Por simplicidad, se recomienda usar directamente la función de biblioteca system, de manera que se ejecute el mandato cat para imprimir el estado del mapa del proceso en ese instante. Nótese que no sería correcto incluir una llamada a system como la siguiente:

```
system("cat /proc/self/maps");
```

Con esta llamada se estaría imprimiendo el mapa del proceso que ejecuta el mandato `cat`. Se debería construir en tiempo de ejecución la cadena de caracteres que constituye el mandato que se especificará en la función `system`, de manera que en ella se incluya el *pid* del proceso que nos interesa.

A continuación se plantea una serie de ejemplos que el alumno debería programar para luego ejecutar y analizar la información sobre el mapa de memoria impresa por los mismos:

- Desarrolle un programa que incluya variables globales con y sin valor inicial, así como variables locales (por ejemplo, variables definidas dentro de la función `main`), tanto de tipo escalar como vectores. Además, el programa usará una variable externa al módulo. Concretamente, la variable global `errno` (`#include <errno.h>`). El programa debe imprimir su mapa de memoria junto con las direcciones de estas variables, incluida `errno`, y de la propia función `main`. Una vez ejecutado, se analizará el mapa impreso identificando a qué región pertenecen las distintas variables y la propia función `main`. Para facilitar el desarrollo de este programa, como ejemplo, se muestra a continuación cómo se podrían imprimir las direcciones de `errno` y de `main`.

```
printf("main %p errno %p\n", main, &errno);
```

- Realice un programa que incluya una función que tenga definida una variable local con un tamaño de 14.000 bytes (por ejemplo, un vector de caracteres de 14.000 elementos). El programa debería imprimir el mapa antes de llamar a la función, dentro de la propia función y después de la ejecución de la misma.
- Desarrolle un programa que incluya una llamada `malloc` que reserve un espacio de 14.000 bytes y luego lo libere usando `free`. El programa debería imprimir el mapa antes de la reserva, mientras está reservado el espacio y después de su liberación.
- Para comparar cómo afecta el uso de las bibliotecas en el mapa del proceso, se plantea realizar tres versiones del mismo programa que usa la función coseno (`cos`) definida en la biblioteca `libm`:

- Versión compilada con bibliotecas estáticas. El programa imprimirá el mapa del proceso. Es interesante aplicar el mandato `size` al ejecutable resultante y comparar el resultado con las otras versiones del programa. El mandato de compilación debe especificar que se desea usar la versión estática de la biblioteca.

```
cc      programa.c -static -lm -o programa
```

- Versión compilada con bibliotecas dinámicas. El programa imprimirá el mapa del proceso. El mandato de compilación no necesita especificar que se desean usar bibliotecas dinámicas, ya que éste es el comportamiento por defecto.

```
cc      programa.c -lm -o programa
```

- Versión que carga explícitamente en tiempo de ejecución la biblioteca matemática (usando `dlopen`) obtiene la dirección de la función coseno (usando `dlsym`), la invoca y elimina la biblioteca dinámica (usando `dlclose`). Se debería imprimir el mapa antes del `dlopen`, mientras está cargada la biblioteca y después del `dlclose`. Nótese que en el mandato de compilación no se especifica la biblioteca matemática, sino la biblioteca que gestiona la carga explícita de bibliotecas dinámicas.

```
cc      programa.c -ldl -o programa
```

- Realice un programa que cree un *thread*. El programa debería imprimir el mapa antes de la creación, durante la existencia del *thread* y después de su finalización.

6. Desarrolle un programa que proyecte un archivo. Se debería imprimir el mapa antes del `mmap`, mientras está proyectado el archivo y después del `munmap`. Pruebe a especificar distintos parámetros en la llamada `mmap` (compartido o privado, distintos permisos, diversos tamaños y desplazamientos, etc.) y analice cómo afectan al mapa.
7. Programe un ejemplo que cree un proceso hijo mediante `fork`. El programa debería imprimir el mapa antes de la creación. Una vez creado el hijo, ambos procesos deben imprimir su mapa.
8. Desarrolle un programa que cree un proceso hijo usando `fork`, tal que el hijo ejecute otro programa mediante `exec`. El hijo debe escribir su mapa antes del `exec` y después del mismo. Para realizar fácilmente esta última impresión, se recomienda que el programa que se va a ejecutar mediante `exec` sea directamente el mandato `cat` sobre el archivo `maps` correspondiente.

3.2.4. Entrega de documentación

Se recomienda que el alumno entregue todos los programas desarrollados, así como una memoria donde analice razonadamente cada caso.

3.2.2. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

3.3. PRÁCTICA: OBTENCIÓN DE ESTADÍSTICAS SOBRE EL MAPA DE MEMORIA DE LOS PROCESOS

3.3.1. Objetivos de la práctica

Esta práctica tiene un objetivo similar a la anterior: llegar a conocer de forma aplicada cómo es el mapa de memoria de un proceso. Asimismo, esta práctica permitirá que el alumno profundice en sus conocimientos sobre la programación de *scripts* en UNIX, puesto que la solución al problema planteado se acometerá usando un *script*.

NIVEL: Introducción.

HORAS ESTIMADAS: 8.

3.3.2. Descripción de la funcionalidad que debe desarrollar el alumno

Se plantea desarrollar un *script*, denominado `estadisticas_mapa`, que recibirá como argumento una lista de identificadores de proceso y deberá mostrar por la salida estándar una serie de estadísticas sobre el mapa de memoria actual de cada uno de ellos. A continuación se describen las características que debe incluir el programa:

- El *script* debe acceder al archivo `maps` de cada proceso para obtener la información sobre su mapa de memoria.

- Con respecto al control de errores, si no se puede acceder al archivo maps de alguno de los procesos solicitados, ya sea porque no existe o por falta de permiso, el programa continuará su ejecución devolviendo al final un 1. En caso contrario, se devolverá al final un 0.
- Por cada proceso, el *script* debe realizar los siguientes recuentos «jerárquicos»:
 - Cuál es el número total de bytes que ocupa el mapa del proceso.
 - Cuántos bytes corresponden a regiones privadas y cuántos a compartidas.
 - Dentro de cada uno de los dos grupos anteriores (regiones privadas o compartidas), se debe calcular cuántos bytes corresponden a regiones con soporte en archivo y cuántos a regiones sin soporte (anónimas).
 - A su vez, en cada uno de los cuatro grupos planteados se distinguirá tres subconjuntos disjuntos:
 - Cuántos bytes corresponden a regiones modificables (con, al menos, permiso de escritura).
 - Cuántos bytes corresponden a regiones ejecutables, pero no modificables (sin permiso de escritura y, al menos, con permiso de ejecución).
 - Cuántos bytes corresponden a regiones de sólo lectura (sólo con permiso de lectura).
- Dado el carácter jerárquico de la salida, se puede sangrar usando tabuladores de manera que se refleje dicha jerarquía:

```

mapa proceso PID1
  total NNNN
    privados NNNN
      con soporte NNNN
        modificables NNNN
        ejecutables (no modificables) NNNN
        sólo lectura NNNN
      anónimos NNNN
        modificables NNNN
        ejecutables (no modificables) NNNN
        sólo lectura NNNN
    compartidos NNNN
      con soporte NNNN
        modificables NNNN
        ejecutables (no modificables) NNNN
        sólo lectura NNNN
      anónimos NNNN
        modificables NNNN
        ejecutables (no modificables) NNNN
        sólo lectura NNNN
mapa proceso PID2
  .
  .
  .

```

El alumno debe aplicar el *script* a varios procesos (algunos de ellos pueden ser los desarrollados en la práctica anterior) y analizar los resultados obtenidos.

3.3.3. Recomendaciones generales

Una de las dificultades de este *script* es realizar los cálculos de los tamaños de las regiones, dado que éstos aparecen en hexadecimal. Para ello, se recomienda usar directamente en el *script* el mandato bc (calculadora interactiva). Por tanto, consulte la página de manual de este mandato.

Asimismo, se recomienda usar algunas utilidades de depuración que proporciona el *shell*. Por ejemplo, el mandato `set -x` hace que el *shell* imprima los mandatos y sus argumentos según se van ejecutando.

3.3.4. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- `memoria.txt`: Memoria de la práctica, que debe incluir el análisis de los resultados de aplicar este mandato a distintos procesos.
- `estadisticas_mapa`: Archivo que contiene el *script*.

3.3.5. Bibliografía

- A. Afzal. *Introducción a UNIX*. Prentice-Hall, 1997.
- C. Newham y B. Rosenblatt. *Learning the bash shell*. Sebastopol: O'Reilly, 1995.
- S. R. Bourne. *The UNIX System*. Addison-Wesley, 1983.

3.4. PRÁCTICA: ESTUDIO DE LOS FALLOS DE PÁGINA DE UN PROCESO EN LINUX

3.4.1. Objetivos de la práctica

El objetivo de esta práctica es entender qué tipos de fallos de página se pueden producir en Linux cuando se ejecuta un proceso.

NIVEL: Introducción.

HORAS ESTIMADAS: 4.

3.4.2. Descripción de la funcionalidad que debe desarrollar el alumno

La práctica consiste en analizar el comportamiento de dos programas que el alumno debe ejecutar en Linux. Los nombres de los dos archivos ejecutables que se proporcionan (véase el apartado Material de apoyo) son `prog1` y `prog2`. Cuando se ejecuta cada uno de estos programas, muestra por pantalla su identificador de proceso y a continuación solicita al usuario que pulse una tecla para continuar su ejecución. El identificador de proceso que se muestra por pantalla se utilizará para acceder al directorio `/proc` (véanse las Prácticas 3.2 y 3.3).

El objetivo de la práctica es determinar el comportamiento que tiene cada uno de estos procesos cuando se ejecuta desde el punto de vista del número de fallos de página que se producen. Linux distingue dos tipos de fallos de página:

- Fallos de página secundarios (*minor page fault*). Se corresponden con fallos de página que no requieren acceso a disco. Este tipo de fallos de página se producen, por ejemplo, en el

primer acceso a una página de datos sin valor inicial y en el acceso, en general, a una página que no tiene soporte inicial en disco.

- Fallos de página primarios (*major page fault*). Se corresponden con fallos de página que requieren acceso a disco; por ejemplo, acceso a páginas de código, de datos con valor inicial, datos que residen en *swap* y datos de archivos proyectados en memoria.

El archivo `/proc/self/maps` permite conocer el número de fallos de página primarios y secundarios de un proceso. Para monitorizar a los programas `prog1` y `prog2` anteriores debe accederse al directorio correspondiente a cada uno de estos procesos en ejecución y determinar qué ocurre con los fallos de página. El archivo anterior consta de una serie de entradas. A continuación se muestran las primeras entradas, entre las que se encuentran el número de fallos de página del proceso:

- Identificador de proceso.
- Nombre del archivo ejecutable.
- Estado del proceso.
- Identificador del proceso padre.
- El grupo del proceso.
- El identificador de sesión del proceso.
- El terminal utilizado por el proceso.
- Grupo de procesos propietario del terminal del proceso.
- *Flags* del proceso.
- Número de fallos de página secundarios.
- Número de fallos de página secundarios del proceso y sus hijos.
- Número de fallos de página primarios del proceso.
- Número de fallos de página primarios del proceso y sus hijos.

Los valores de interés para el desarrollo de esta práctica son los cuatro últimos valores. El alumno debe ejecutar el programa `prog1` y `prog2` y, consultando el archivo anterior, responder a las siguientes preguntas:

1. Indicar el comportamiento de cada uno de los programas.
2. Número de fallos de página que generan.
3. Indicar a qué pueden ser debidos los fallos de página que se producen.

3.4.3. Material de apoyo

En la página web del libro se puede encontrar el archivo `practica-3.4.tgz`. Al descomprimir dicho archivo se obtendrán los programas `prog1` y `prog2`.

3.4.4. Entrega de documentación

Se recomienda que el alumno entregue el siguiente archivo:

- `memoria.txt`: Memoria de la práctica, que debe incluir la respuesta a las preguntas realizadas anteriormente.

3.4.5. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- S. R. Bourne. *The UNIX System*. Addison-Wesley, 1983.

3.5. PRÁCTICA: ESTUDIO DE LOS FALLOS DE PÁGINA DE UN PROCESO EN WINDOWS

3.5.1. Objetivos de la práctica

Esta práctica es similar a la anterior y su objetivo es entender qué tipos de fallos de página se pueden producir en Windows cuando se ejecuta un proceso.

NIVEL: Introducción.

HORAS ESTIMADAS: 4.

3.5.2. Descripción de la funcionalidad que debe desarrollar el alumno

Esta práctica es similar a la anterior y consiste en analizar el comportamiento de dos programas que el alumno debe ejecutar en Windows. Los nombres de los dos archivos ejecutables que se proporcionan (véase el apartado Material de apoyo) son *prog1* y *prog2*. Cuando se ejecuta cada uno de estos programas, solicita al usuario que pulse una tecla para continuar su ejecución.

El objetivo de la práctica es determinar el comportamiento que tiene cada uno de estos procesos cuando se ejecuta desde el punto de vista del número de fallos de página que se producen. Windows distingue dos tipos de fallos de página (similares a los de Linux):

- Fallos de página *soft*. Se corresponden con fallos de página que no requieren acceso a disco. Este tipo de fallos de página se producen, por ejemplo, en el primer acceso a una página de datos sin valor inicial y en el acceso, en general, a una página que no tiene soporte inicial en disco.
- Fallos de página *hard*. Se corresponden con fallos de página que requieren acceso a disco; por ejemplo, acceso a páginas de código, de datos con valor inicial, datos que residen en *swap* y datos de archivos proyectados en memoria.

Para el desarrollo de esta práctica se utilizará el monitor de rendimiento de sistema que proporciona Windows 2000 y que se describe brevemente en el siguiente apartado.

Uso del monitor de rendimiento

Para ejecutar el monitor de rendimiento siga los siguientes pasos:

1. Seleccione el botón **Inicio | Programas | Herramientas administrativas**.
2. Pulse dentro del menú anterior la opción **Monitor de sistema (Performance)**.

A continuación aparecerá una ventana como la que se muestra en la Figura 3.7.

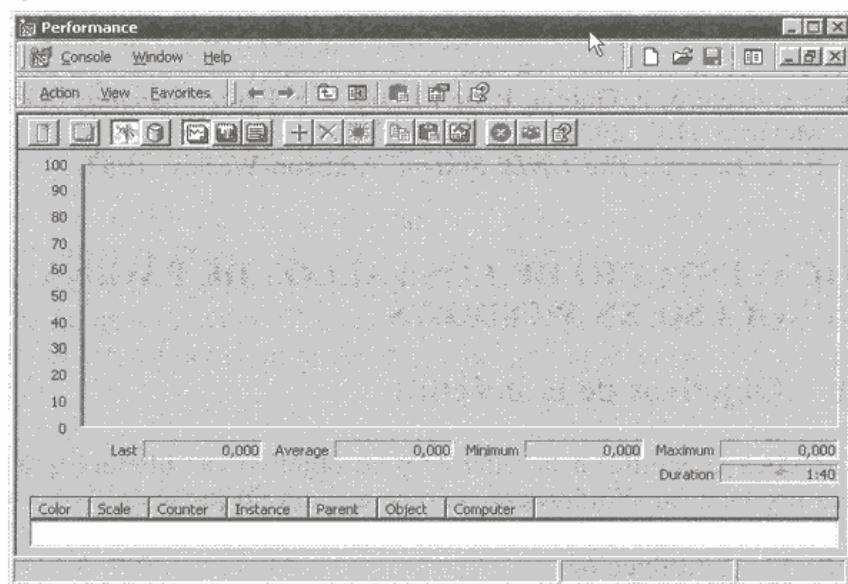


Figura 3.7. Ventana inicial del monitor de rendimiento de Windows.

El monitor de rendimiento permite monitorizar ciertos parámetros relacionados con la memoria del sistema y de los diferentes procesos que ejecutan en el mismo. Para monitorizar uno o varios parámetros basta con pulsar el ícono **Agregar contador** (*add counter*) de la pantalla inicial que se muestra en la Figura 3.8 y elegir los parámetros que se quieren visualizar. Para comprobar los fallos de página que genera cada proceso, se seleccionará como objeto de rendimiento **Proceso** (**Process**) y el proceso que se desea monitorizar. El contador *fallos de página por segundo* contabiliza los fallos de página que genera el proceso, tanto fallos de página *soft* como fallos de página *hard*. Para analizar de qué tipo son los fallos de página que se generan en el sistema hay

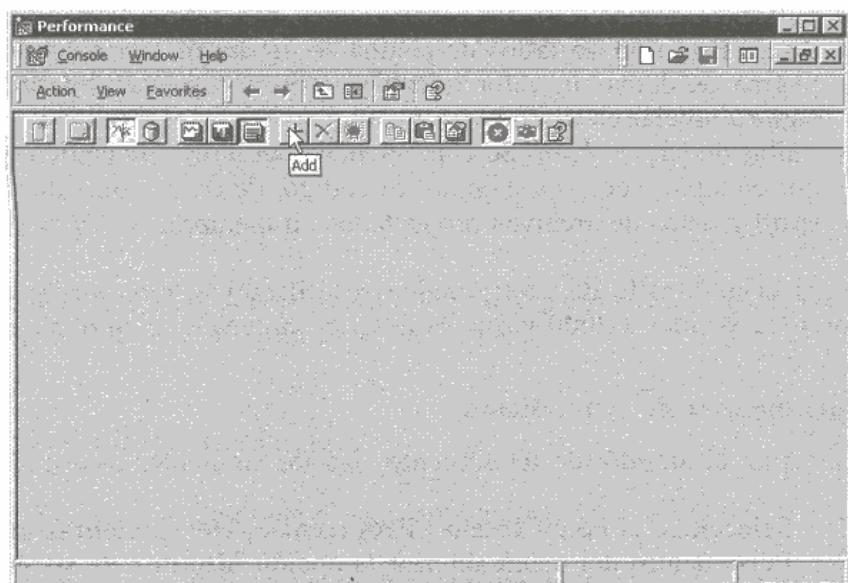


Figura 3.8. Agregar un nuevo contador.

que añadir como objeto de monitorización la *memoria (Memory)* y seleccionar los dos siguientes contadores:

- El parámetro *páginas/sec* contabiliza el número de fallos de página *hard* por segundo, es decir, fallos de página que requieren acceso a disco.
- El parámetro *fallos de página/sec* contabiliza el número de fallos de página total.

Utilizando los parámetros anteriores, el alumno debe ejecutar el programa *prog1* y *prog2* y responder a las siguientes cuestiones:

1. Indicar el comportamiento de cada uno de los programas.
2. Número de fallos de página que generan.
3. Indicar a qué pueden ser debidos los fallos de página que se producen.

3.5.3. Recomendaciones generales

Antes de empezar con la monitorización, se recomienda que el alumno se familiarice con el monitor de rendimiento del sistema. Para ello, puede consultarse la ayuda en línea que ofrece dicho monitor. Cuando se monitoricen los programas anteriores, conviene que no haya ningún otro programa ejecutando en la máquina.

3.5.4. Material de apoyo

En la página web de libro se puede encontrar el archivo *practica-3.5.zip*. Al descomprimir dicho archivo se obtendrán los programas *prog1* y *prog2*.

3.5.5. Entrega de documentación

Se recomienda que el alumno entregue el siguiente archivo:

- *memoria.doc*: Memoria de la práctica, que debe incluir la respuesta a las preguntas realizadas anteriormente.

3.5.6. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.

3.6. PRÁCTICA: GESTIÓN DE MEMORIA DINÁMICA

3.6.1. Objetivos de la práctica

El objetivo principal de la práctica es que el alumno llegue a conocer de forma aplicada el concepto de memoria dinámica de un proceso y la manera de gestionar este tipo de memoria. Esta gestión

se inscribe dentro del problema general de la asignación dinámica de espacio que aparece en otras situaciones dentro del campo de los sistemas operativos (por ejemplo, en la gestión del espacio de un sistema de archivos).

NIVEL: Avanzado.

HORAS ESTIMADAS: 14.

3.6.2. Conceptos generales sobre la memoria dinámica

La mayoría de los lenguajes de alto nivel ofrecen la posibilidad de reservar espacio en tiempo de ejecución. En el caso del lenguaje C, se proporciona la función `malloc` para realizar esta labor.

Tradicionalmente, se usa la región de *heap* del proceso como soporte para la memoria dinámica que reserva un programa en tiempo de ejecución. El sistema operativo sólo realiza una gestión básica de esta región. Generalmente, las aplicaciones no usan directamente los servicios proporcionados por el sistema operativo para la gestión del *heap*, sino que la biblioteca de cada lenguaje de programación utiliza estos servicios básicos para construir a partir de ellos unos más avanzados orientados a las aplicaciones.

En el caso de UNIX, se proporciona el servicio `sbrk` para incrementar o decrementar la región de datos del proceso (el *heap* se puede considerar como la extensión de la región de datos sin valor inicial):

```
void *sbrk(ptrdiff_t incremento);
```

Esta función modifica el tamaño de la región de datos aumentándola (o disminuyéndola si el valor es negativo) de acuerdo con el argumento recibido. Devuelve la dirección hasta donde se extendía la región de datos antes de la llamada. Concretamente, la dirección de la primera posición que quedaba fuera de la zona de datos antes de llevar a cabo esta última petición. Hay que resaltar que esta función, disponible en la mayoría de los sistemas UNIX, no está incluida en el estándar POSIX.

Como se analizará en la práctica, la implementación de la memoria dinámica sobre el *heap*, que es la solución más típica, presenta como inconveniente que se requiere que toda la memoria dinámica ocupe una zona contigua en el mapa de memoria del proceso cuando esto no es realmente necesario. Esto puede llevar a situaciones donde el *heap* no se puede expandir debido a que ha «chocado» con la siguiente región del mapa.

La biblioteca que implementa la gestión de memoria dinámica para un determinado lenguaje debe encargarse de gestionar el espacio «en bruto» que le ofrece el *heap* para ir satisfaciendo las peticiones del programa. Según el programa vaya solicitando y liberando memoria dinámica, se van creando espacios reservados y huecos en la región de *heap*, que tendrá que gestionar la biblioteca siguiendo una determinada política de asignación de espacio. Ante una solicitud de reserva de espacio, existen, típicamente, tres estrategias básicas:

- El mejor ajuste (*best-fit*). Se elige la zona libre más pequeña que satisfaga la petición. *A priori*, puede parecer la mejor solución. Sin embargo, esto no es así. Por un lado, se generan nuevos espacios libres muy pequeños. Por otro lado, la selección del mejor hueco exige comprobar cada uno de ellos o mantenerlos ordenados por tamaño. Ambas soluciones conducen a un algoritmo ineficiente.
- El peor ajuste (*worst-fit*). Se elige el hueco más grande. Con ello se pretende que no se generen nuevos huecos pequeños. Sin embargo, sigue siendo necesario recorrer toda la lista de huecos o mantenerla ordenada por tamaño.

- El primero que ajuste (*first-fit*). Aunque pueda parecer sorprendente *a priori*, ésta suele ser la mejor política. Es muy eficiente, ya que basta con encontrar una zona libre de tamaño suficiente y proporciona un aprovechamiento de la memoria aceptable.

3.6.3. Descripción de la funcionalidad que debe desarrollar el alumno

La práctica consistirá en diseñar y codificar, en lenguaje C y sobre el sistema operativo Linux/UNIX, un módulo con rutinas que proporcionen mecanismos de gestión de memoria dinámica. Concretamente, se implementarán las propias funciones de gestión de memoria disponibles en C: `malloc`, `free` y `realloc`. Nótese, por tanto, que se están reescribiendo estas rutinas estándar de C.

Se tomará como punto de partida el código desarrollado y explicado en el apartado 8.7 del libro clásico de C de Kernighan y Ritchie incluido en la bibliografía de la práctica. Dicho código se entregará como material de apoyo de la práctica. Es muy importante comprenderlo correctamente para poder desarrollar esta práctica. El alumno podrá observar al analizar este código de apoyo que no se trata de un paradigma de la programación modular y estructurada, sino más bien al contrario, se trata de un código bastante enrevesado y poco legible, a pesar de tratarse de un fragmento extraído de un libro de programación. Dado que el tamaño de este código es relativamente pequeño, su análisis puede ser un ejercicio formativo, ya que permite ver cómo a veces los programadores *profesionales* tienen tendencia a generar código difícil de entender, más aún cuando usan como herramienta un lenguaje tan permisivo como C.

Centrándonos en el código de apoyo, en él están implementadas las funciones para reserva de espacio `malloc` y para liberación de espacio `free`. Nótese que se trata de una implementación simplificada que, aunque cumple toda la funcionalidad real asociada a estas funciones, no se preocupa de aspectos, tales como la eficiencia al recorrer las estructuras de listas usadas en la misma.

Antes de plantear la labor que debe realizarse en esta práctica, se van a resaltar algunos de los aspectos más relevantes del código que se proporciona como material de apoyo.

- Se mantiene una lista circular de huecos ordenada por direcciones. La función `malloc` consulta esta lista para buscar un hueco que satisfaga una determinada solicitud de reserva. Para simplificar el código, esta lista tiene siempre al menos un componente: un falso hueco (`base`) de tamaño cero que se crea en la primera llamada `malloc`.
- Cada hueco tiene asociada una cabecera (`Header`) que contiene un campo con la longitud del hueco medida en *tamaños de cabecera* (incluyendo la propia cabecera) y un puntero al siguiente hueco. Así, el espacio se asigna tomando como unidad el tamaño de la cabecera, redondeando por exceso el número de bytes pedidos en la llamada `malloc` a un número exacto de cabeceras. Se desperdiciará, por tanto, parte del espacio asignado.
- Los bloques asignados no están incluidos en ninguna lista. Tienen asociada una cabecera igual que la de los huecos, pero donde el único campo significativo es el tamaño.
- La estrategia de asignación de espacio es *next-fit*: se utiliza, al igual que en *first-fit*, el primer hueco que se encuentre cuyo tamaño sea suficiente para satisfacer la petición, pero, a diferencia de esta estrategia, se comienza cada vez la búsqueda desde donde terminó la última. La variable global `freep` cumple esta función apuntando siempre al hueco que está situado en la lista justo antes del hueco por el que comenzará la búsqueda. Esto es, la búsqueda de un hueco empieza siempre por el hueco siguiente al apuntado por `freep` (`freep->s.ptr`).
- Tanto el puntero que devuelve `malloc` como el que recibe `free` apuntan justo después de la cabecera del bloque, esto es, a la zona donde el usuario puede almacenar sus datos.

Nótese que, evidentemente, la aplicación que usa esta biblioteca no debe ser consciente de la existencia de las cabeceras.

- Cuando `malloc` detecta que no se puede satisfacer una petición, se invoca a `sbrk` (a través de la función interna `morecore`) para aumentar el tamaño de la zona de datos asignada al programa y se añade esta nueva zona a la lista de huecos usando `free`.
- La rutina `free` se encarga de comprobar si el bloque que se libera genera un hueco que se compacta con otros huecos adyacentes.
- Cuando sólo se necesita usar parte de un hueco para satisfacer una petición, se utiliza la parte de direcciones más altas, creándose un nuevo hueco con la parte de direcciones más bajas.
- Se resuelve el problema del alineamiento de los datos. En numerosas arquitecturas existen restricciones en la manera en que un determinado tipo de datos puede almacenarse en la memoria. Así, por ejemplo, en muchas arquitecturas, los enteros deben comenzar en una dirección que sea múltiplo de 4. Como el módulo de la práctica no conoce el tipo de los datos que se guardarán en la zona pedida, deberá asegurar que las direcciones que devuelve permitan almacenar a partir de ellas un valor del tipo de datos más restrictivo en esa arquitectura. Para ello se define un tipo `Align` y se fuerza que el tamaño de la cabecera sea múltiplo de este tipo definiendo ésta como una `union` con un campo `x` de tipo `Align`.

Partiendo del código comentado y manteniendo sus mismas características, se pide incluir los siguientes aspectos:

- Como se comentó en el apartado anterior, el uso del *heap* como soporte de la memoria dinámica implica que toda la memoria dinámica tiene que estar en una zona contigua del mapa. Para eliminar esta restricción, como primera modificación, se plantea usar regiones de memoria anónimas para implementar la memoria dinámica. Se modificará el código de `morecore` para que deje de usar la llamada `sbrk`, utilizando en su lugar una llamada `mmap` para establecer una proyección anónima. Será, por tanto, necesario asegurar que el tamaño de la región proyectada sea múltiplo del tamaño de página (se puede usar la llamada `sysconf` para averiguar el tamaño de la página). Además, para evitar un número excesivo de llamadas a `mmap`, se seguirá el criterio de proyectar una región que tenga un tamaño como mínimo de dos páginas.
- Para hacer más robusto y tolerante a fallos el código de la biblioteca, se va a incluir un sencillo mecanismo de control de errores que detecte cuando, por error, se está intentando liberar un espacio que no ha sido reservado previamente. Si se detecta esta situación, la rutina `free` debe retornar inmediatamente sin llevar a cabo la solicitud de liberación. Nótese que si se analiza el código de la función `free`, puede comprobarse que este tipo de error puede causar graves problemas al correcto funcionamiento de la misma, ya que causaría la inclusión en la lista de huecos de un hueco que realmente no existe. Como mecanismo de comprobación de validez, se propone añadir a la cabecera de un bloque un «número mágico» (un valor previamente conocido), de manera que la rutina `free` pueda comprobar su presencia antes de proceder a la liberación.
- Modificar el tamaño de la unidad de asignación de espacio. En lugar de asignar el espacio usando como unidad el tamaño de la cabecera, se usará el tamaño del tipo de datos más restrictivo en cuanto al alineamiento (para la práctica se usará el tipo `Align`, que está definido como `long`). El uso de esta unidad de asignación más pequeña mejorará el aprovechamiento del espacio de almacenamiento. Algunos aspectos que hay que tener en cuenta al desarrollar esta modificación son:
 - El cambio en el tamaño de la unidad de asignación afecta al cálculo del número de unidades que se necesitan para satisfacer una petición y al valor que se almacena en el

campo `size` de la cabecera. En el código de apoyo se guarda en dicho campo el número de cabeceras que ocupa un bloque incluyendo la propia cabecera. Con el cambio de unidad de asignación, en su lugar se deberá almacenar el número de `Aligns` que ocupa en total el bloque (datos + cabecera).

- Otro aspecto que hay que revisar en el código original es el relacionado con la aritmética de los punteros. En el lenguaje C, la expresión `p + n`, siendo `p` un puntero y `n` un entero, dará un resultado diferente dependiendo del tipo de puntero. Así, si `p` es de tipo `(Header *)`, se le sumará al valor de `p` el tamaño de `n` cabeceras, mientras que si es de tipo `(Align *)` se le sumará `n` veces el tamaño del tipo `Align`. Para resolver este tipo de problemas puede ser necesario usar la operación de *cast* para transformar tipos de punteros. Un ejemplo de este tipo de operación extraído del código original es el siguiente: `bp = (Header *)ap - 1.`
- En la implementación inicial pueden aparecer huecos de tamaño 1 cuando se asigna espacio en `malloc` si en el espacio que sobra del hueco elegido sólo cabe una cabecera. Estos huecos pueden ralentizar la búsqueda del hueco apropiado en posteriores llamadas `malloc`, ya que se tienen que consultar aunque no sirvan para satisfacer ninguna petición. Se modificará el código para que la función `malloc` sólo genere un hueco sobrante si su tamaño es mayor o igual que el tamaño de la cabecera más el de una unidad de asignación, esto es, si el espacio sobrante no sirve para, al menos, satisfacer una llamada `malloc` de un byte, no se genera el hueco y se deja que el bloque ocupe todo, apuntando en su cabecera el tamaño de todo el espacio.
- Codificar la función `realloc`, que cambia el tamaño de un bloque previamente reservado manteniendo su contenido inalterado hasta el mínimo de los tamaños nuevo y viejo. La funcionalidad de esta rutina será idéntica a la de su homónima estándar de C. Asimismo, al igual que la rutina `free`, esta función deberá comprobar que realmente se está tratando de redimensionar un bloque previamente reservado. En caso de error, devolverá un valor nulo. Si todo va bien, devuelve la dirección donde reside el bloque una vez redimensionado. Para aclarar el comportamiento que debe tener esta función, analizaremos a continuación los distintos casos que pueden presentarse.
 - Si la petición consiste en disminuir el tamaño, deberá generarse un hueco con la parte sobrante siempre que su tamaño cumpla los requisitos del punto anterior. Un aspecto importante es que, aunque en principio la zona sobrante sea de un tamaño inservible, puede tener adyacente un hueco con el que se puede compactar, debiéndose producir en este caso la generación y compactación del hueco. En resumen, no sólo se generará un hueco con la parte sobrante si su tamaño es insuficiente y además no es adyacente a un hueco. En el caso de que se genere un hueco, la variable `freep` deberá quedar apuntando al hueco anterior al generado, o sea, siguiendo el mismo criterio que la función `free`.
 - Si la petición consiste en aumentar el tamaño y existe un hueco adyacente a la parte final del bloque con un tamaño suficientemente grande, no será necesaria la reubicación (o sea, mover el contenido del bloque a otra zona, lo cual es una operación costosa). Sólo será necesario, si se cumplen los requisitos de tamaño mínimo, generar un nuevo hueco con la parte sobrante del hueco adyacente usado y ajustar el nuevo tamaño del bloque. En este caso, la variable `freep` deberá quedar apuntando al hueco anterior al usado en la expansión.
 - En caso de que se trate de una petición de aumentar el tamaño y bien no haya un hueco adyacente a la parte final o bien el tamaño del mismo no es suficiente, hará falta una reubicación. Se buscará espacio para el nuevo tamaño del bloque usando el

orden *next-fit*, se copiará el contenido al nuevo destino y, por último, se liberará la zona original ocupada por el bloque. La variable `freep`, por tanto, quedará con el valor correspondiente a la liberación del espacio original.

3.6.4. Código fuente de apoyo

Para facilitar la realización de la práctica, se dispone en la página web del libro del archivo `practica-3.6.tgz`, que contiene el código fuente de apoyo que se corresponde con el código incluido en el libro de programación anteriormente comentado. Al extraer su contenido, se crea el directorio `practica 3.6` donde se debe desarrollar la práctica. Dentro de este directorio se encuentran los siguientes archivos:

- `Makefile`: archivo fuente para la herramienta `make`. Con él se consigue la recopilación automática de los archivos fuente cuando se modifiquen. Basta con ejecutar el mandato `make` para que el programa se compile de forma automática.
- `mialloc.h`: archivo de cabecera con definiciones de constantes y prototipos de funciones. Este archivo no debería ser modificado a la hora de realizar la práctica.
- `mialloc.c`: archivo fuente de C donde se incluirá la funcionalidad pedida por el enunciado.

3.6.5. Recomendaciones generales

Es importante analizar el código de apoyo proporcionado con la práctica, ya que será el punto de partida para la realización de la misma. Se recomienda ir incluyendo de forma incremental las modificaciones planteadas siguiendo el orden con el que aparecen en el enunciado.

3.6.6. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- `memoria.txt`: Memoria de la práctica.
- `mialloc.c`: Archivo fuente de C donde estará incluida la funcionalidad pedida por el enunciado.

3.6.7. Bibliografía

- B. Kernigham y D. Ritchie. *The C programming language*. 2.^a edición, Prentice-Hall, 1988.
- W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

3.7. PRÁCTICA: PROYECCIÓN DE ARCHIVOS EN MEMORIA

3.7.1. Objetivos de la práctica

El objetivo principal es que el alumno se familiarice con el concepto de proyección de archivos en memoria y con los servicios disponibles en POSIX para llevar a cabo esta operación. La

práctica intenta mostrar cómo esta técnica facilita la programación de distintas operaciones sobre los archivos y, además, generalmente agiliza su ejecución comparando con las soluciones basadas en el acceso convencional a los archivos. Asimismo, se pretende enseñar cómo influye la existencia de múltiples procesos, ya sean convencionales o ligeros, en la proyección de los archivos.

NIVEL: Avanzado.

HORAS ESTIMADAS: 20.

3.7.2. Descripción de la funcionalidad que debe desarrollar el alumno

La práctica se va a dividir en dos partes. En la primera, se desarrollarán programas que manejen archivos de tipo texto usando la técnica de proyección, mientras que en la segunda se usarán archivos de tipo binario.

Proyección de archivos en memoria con archivos de tipo texto

En esta primera parte de la práctica se plantea la realización de varios programas que usan la técnica de proyección de archivos en memoria para manejar archivos de texto. Los tres programas que se proponen están inspirados en mandatos estándar de UNIX, aunque considerablemente simplificados. Antes de pasar a describirlos, se considera conveniente resaltar que sería recomendable alentar a los alumnos para que lleven a cabo implementaciones más completas de estos programas o que aborden la realización de versiones basadas en proyección de archivos de otros mandatos estándar de UNIX que manejen archivos de texto. A continuación se describen los tres ejercicios planteados:

- Programa `cola`. Se debe desarrollar un programa similar al mandato `tail` que imprima las últimas líneas de un archivo usando, evidentemente, la técnica de proyección de archivos en memoria. El programa recibirá como argumentos el número de líneas que se quieren imprimir y el nombre del archivo deseado.
- Programa `buscar`. Se pretende implementar una versión simplificada del mandato `grep` que imprima las líneas de un archivo que contengan una determinada secuencia de caracteres. El programa recibirá como argumentos la secuencia de caracteres y el archivo donde se quiere buscar.
- Programa `amayuscula`. Este programa, que se podría considerar basado en el mandato `tr`, debe modificar cualquier carácter de un archivo que corresponda con una letra minúscula (de la *a* a la *z*) por la correspondiente letra mayúscula (de la *A* a la *Z*). Si el programa recibe como argumento sólo el nombre de un archivo, se cambiarán las minúsculas por mayúsculas en el propio archivo. En cambio, si se especifican dos nombres de archivo, el primero permanecerá inalterado, copiándose en el segundo el contenido del primero, pero con las letras pasadas a mayúsculas.

Sería interesante que el alumno desarrollara estos mismos programas accediendo a los archivos de forma convencional. Esto le permitiría observar la facilidad que proporciona la técnica de proyección de archivos en memoria a la hora de programar los algoritmos, ya que el programador no tiene que estar manejando *buffers* ni estudiando «trozos» independientes del archivo. Con esta técnica, el archivo se puede manejar como si fuera una gran cadena de caracteres continua.

Proyección de archivos en memoria con archivos de tipo binario

En esta segunda parte de la práctica se plantea el desarrollo de varios programas que usan la técnica de proyección de archivos en memoria para manejar archivos de tipo binario. Concretamente, se van a usar archivos cuyo contenido va a corresponder con vectores de números en coma flotante. Básicamente, la práctica va a consistir en la realización de programas que ordenen vectores de este tipo. Sin embargo, dadas las dificultades para poder generar y leer archivos de tipo binario, se plantea como primera etapa el desarrollo de dos programas que realicen esta labor:

- Programa `genera_vector`. Crea un archivo con el nombre especificado como argumento del programa y en él incluye, usando la técnica de proyección en memoria, los números reales que lee de la entrada estándar.
- Programa `imprime_vector`. Usando la técnica de proyección en memoria, imprime en la salida estándar con formato de texto el contenido del archivo binario especificado como argumento del programa y que corresponde con un vector de números reales.

Una vez programadas estas dos utilidades, se plantea el desarrollo de los programas de ordenación de vectores que constituyen el objetivo principal de la práctica:

- Programa `ordena_vector`. El programa debe ordenar de forma creciente el contenido de un vector de números reales en coma flotante. Para ello, se recomienda utilizar la función de biblioteca `qsort`. Si el programa recibe como argumento sólo el nombre de un archivo, se ordenará el archivo «in-situ». En cambio, si se especifican dos nombres de archivo, el primero permanecerá inalterado, debiendo quedar en el segundo el resultado del ordenamiento.
- Programa `ordena_y_suma_vectores`. Se trata de un programa que ordena de forma concurrente un conjunto de vectores y a continuación los suma. El programa recibirá como argumento el nombre de N archivos (al menos dos): los $N-1$ primeros, que deberán tener el mismo tamaño, representan vectores de números reales que, en primer lugar, serán ordenados concurrentemente y que, a continuación, serán tratados como operandos de la suma, mientras que el enésimo será considerado como el contenedor del vector resultante. Después de la ejecución del programa, además de haberse generado el vector resultado, habrán quedado ordenados los vectores operando. Con respecto al esquema de concurrencia utilizado, se deberán desarrollar dos soluciones: una basada en procesos convencionales (`ordena_y_suma_vectores_pr`) y otra en procesos ligeros (`ordena_y_suma_vectores_th`).
- Se plantea añadir una opción a las dos versiones del programa anterior (opción `-n`, que deberá especificarse como primer argumento del programa), tal que si se indica esta opción los archivos especificados como operandos seguirán siendo ordenados antes de proceder a realizar la suma, pero esta ordenación no quedará posteriormente reflejado en los mismos. Así, después de la ejecución del programa, sólo se habrá generado el vector resultado, quedando inalterados los vectores especificados como operandos. En principio, parece que esta opción se resolvería directamente usando proyecciones de tipo privado, pero esto no ocurre así en las dos versiones del programa. El alumno debe analizar esta situación y determinar el motivo de esta diferencia de comportamiento planteando cómo solucionar el problema surgido.

Para que el alumno pueda ver de forma práctica las mejoras en eficiencia que proporciona la técnica de proyección de archivos en memoria frente a la forma convencional de acceder a los

archivos (lecturas y escrituras), se recomienda que desarrolle una versión del programa `ordena_vector` basada en accesos convencionales a archivos y que compare la eficiencia de ambas soluciones ejecutando ambos programas con el mandato `time`. La versión convencional debería permitir la especificación de distintos tamaños a la hora de acceder al archivo para poder así evaluar la influencia del tamaño de los accesos en el tiempo de ejecución de la versión convencional del programa.

3.7.3. Código fuente de apoyo

Para facilitar la realización de la práctica, se recomienda proporcionar a los alumnos un archivo que contenga ejemplos de programas que usan la técnica de proyección de archivos, como el existente en la página web del libro (`practica-3.7.tgz`).

3.7.4. Recomendaciones generales

Para la primera parte de la práctica, se recomienda que el alumno utilice funciones estándar de manejo de caracteres y cadenas para facilitar la programación de los ejercicios propuestos en la misma.

3.7.5. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- `memoria.txt`: Memoria de la práctica, donde se incluirán los comentarios razonados que se han solicitado en distintas partes de la práctica.
- `cola.c`: Archivo fuente de C, donde se incluirá el programa `cola`.
- `buscar.c`: Archivo fuente de C, donde se incluirá el programa `buscar`.
- `amayuscula.c`: Archivo fuente de C, donde se incluirá el programa `amayuscula`.
- `genera_vector.c`: Archivo fuente de C, donde se incluirá el programa `genera_vector`.
- `imprime_vector.c`: Archivo fuente de C, donde se incluirá el programa `imprime_vector`.
- `ordena_vector.c`: Archivo fuente de C, donde se incluirá el programa `ordena_vector`.
- `ordena_y_suma_vectores_pr.c`: Archivo fuente de C, donde se incluirá el programa `ordena_y_suma_vectores_pr`.
- `ordena_y_suma_vectores_th.c`: Archivo fuente de C, donde se incluirá el programa `ordena_y_suma_vectores_th`.

3.7.6. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

3.8. PRÁCTICA: USO DE BIBLIOTECAS DINÁMICAS

3.8.1. Objetivos de la práctica

El objetivo principal es que el alumno se familiarice con el concepto de biblioteca dinámica y llegue a conocer usos avanzados de este mecanismo.

NIVEL: Avanzado.

HORAS ESTIMADAS: 26.

3.8.2. Descripción de la funcionalidad que debe desarrollar el alumno

La práctica plantea una serie de ejercicios que permitirán al alumno afianzar, de forma aplicada, sus conocimientos sobre el uso de bibliotecas dinámicas.

Aplicación que procesa diversos tipos de archivos

En este primer ejercicio se plantea el desarrollo de un programa, denominado `procesa_archivos`, que, usando la técnica de la carga explícita de bibliotecas dinámicas, tenga la capacidad de procesar distintos tipos de archivos, siendo capaz de «aprender» en tiempo de ejecución cómo procesar un nuevo tipo de archivo. Evidentemente, lo de menos en esta práctica es qué procesamiento real se llevaría sobre cada tipo de archivo. Lo importante es demostrar la capacidad para poder añadir dinámicamente al programa en ejecución la funcionalidad requerida para procesar nuevos tipos de archivos.

Siguiendo una estrategia bastante típica, el tipo de un archivo vendrá determinado por la extensión que aparezca en su nombre (`.html`, `.txt`, `.zip`, etc.). Existirá un archivo de configuración, llamado `config`, donde se especificará una línea por cada tipo de archivo que actualmente se sepa procesar. Cada línea contendrá la extensión que caracteriza a los archivos de ese tipo, así como el nombre del archivo que contiene la biblioteca dinámica que incluye la funcionalidad para procesar ese tipo de archivo:

```
html bibhtml.so
text bibtext.so
. . . . .
```

Toda biblioteca dinámica encargada de procesar un determinado tipo de archivo debe incluir una función denominada `procesar`, que recibe como único argumento el nombre del archivo que se pretende procesar y que llevará a cabo el procesamiento específico de ese tipo de archivos.

A continuación se describe el modo de operación del programa que, dado que lo que se pretende es enfatizar en los aspectos relacionados con el manejo de bibliotecas dinámicas, será muy sencillo:

- El programa consistirá en un bucle que, repetidamente, lee de la entrada estándar el nombre de un archivo y realiza su procesamiento.
- Por cada nombre de archivo leído, se debe extraer su extensión y consultar en el archivo de configuración para determinar cómo procesarlo.
- Si no hay una entrada asociada a esa extensión, el programa imprimirá un mensaje de error y pasará a leer nuevamente de la entrada estándar.

- En caso de que sí exista la entrada, se cargará la biblioteca especificada y se invocará a la rutina procesar de esa biblioteca que realizará el tratamiento adecuado, pasándole como parámetro el nombre del archivo.

Para hacer que el programa pueda manejar un nuevo tipo de archivos, una vez desarrollada la biblioteca dinámica correspondiente, sólo es necesario modificar el archivo de configuración para reflejar este nuevo tipo. Ni siquiera es necesario reiniciar el programa `procesa_archivos`: éste inmediatamente tendrá acceso a esta nueva funcionalidad al encontrar en el archivo de configuración la nueva entrada añadida. De manera similar, se podría eliminar el tratamiento de un determinado tipo de archivos o modificarlo sin afectar al programa. Bastaría, simplemente, con actualizar adecuadamente el archivo de configuración.

Dado que, como se comentó previamente, el objetivo de la práctica es el desarrollo de la infraestructura de procesamiento dinámico, pero no el procesamiento real, se recomienda generar varias bibliotecas dinámicas que incluyan funciones `procesar` que, simplemente, impriman el nombre de la biblioteca y el del archivo recibido como argumento.

Planificador de tareas dinámico

Se pide el desarrollo de un programa, denominado `planificador`, que se encargue de ejecutar cíclicamente un conjunto de tareas de distinta prioridad. Mediante el uso de la técnica de la carga explícita de bibliotecas dinámicas, se podrán añadir y eliminar tareas en tiempo de ejecución sin necesidad de reiniciar el planificador. Se debe resaltar que, como ocurría en la práctica anterior, lo de menos en esta práctica es qué labor van a llevar a cabo las tareas planificadas. El objetivo principal es diseñar la infraestructura que permite esta carga dinámica de tareas. A continuación se describe el modo de operación del planificador:

- Cada tarea estará implementada en una biblioteca dinámica. Cada biblioteca dinámica debe incluir dos funciones: `carga`, que recibirá como parámetro un entero indicando su prioridad y que será invocada por el planificador cuando se quiera incluir esta tarea, y `descarga`, que no tendrá parámetros y que será llamada cuando se desee eliminarla. La biblioteca dinámica debe incluir también una función con un único argumento de tipo `void *`, que será la que lleve a cabo la tarea programada. Además de estas funciones, la biblioteca puede contener todas las funciones adicionales que se considere necesario.
- El planificador gestionará una lista de tareas. Cada tarea quedará identificada por una referencia a la función que ejecuta la tarea, que reside en una biblioteca dinámica, y una referencia a un argumento (de tipo puntero genérico `void *`), que también hará referencia a un objeto de la biblioteca dinámica. Cada vez que el planificador quiera ejecutar una determinada tarea, invocará a la función asociada a la misma pasándole el argumento almacenando en la entrada correspondiente de la lista.
- La lista de tareas estará inicialmente vacía. El planificador ofrecerá funciones para insertar una nueva tarea o para eliminar una existente. Los prototipos de estas funciones podrían ser como los siguientes:

```
void *insertar(void (*tarea)(void *, void *argumento, int prioridad);
               void eliminar(void *descriptor);
```

La rutina de inserción devuelve un valor de tipo `void *`, que representa un descriptor de esa tarea insertada, y recibe como primer parámetro la dirección de la función que implementa la tarea, que será una función `void` con un parámetro de tipo `void *`. El segundo

parámetro es una referencia al argumento que será pasado a la función cuando sea invocada y el último representa la prioridad que se le dará a esta tarea. En cuanto a la función `eliminar`, recibe como parámetro el descriptor de la tarea devuelto por la rutina `insertar` que incluyó anteriormente esta tarea en la lista. Hay que resaltar que, dado que desde la biblioteca dinámica se tiene que poder acceder a estas funciones, habría que generar el programa usando la opción `-rdynamic` del montador para asegurar esta visibilidad.

- Se usará un FIFO (tubería con nombre) para indicarle al planificador en tiempo de ejecución que debe incluir o eliminar una tarea. En su fase de arranque, el planificador creará este FIFO y lo abrirá en modo de lectura-escritura para evitar bloquearse en la apertura.
- Se desarrollarán dos programas: `insertar_tarea` y `eliminar_tarea`. El programa `insertar_tarea` recibirá como argumento el nombre de la biblioteca dinámica que contiene la tarea y la prioridad con la que se pretende que ejecute. El programa `eliminar_tarea` recibirá como único argumento el nombre de la biblioteca dinámica que se desea eliminar. Estos programas deben escribir en el FIFO estos valores, así como el tipo de operación que se pretende hacer (insertar una nueva tarea o eliminar una existente, respectivamente).
- Si la lista de tareas está vacía, ya sea inicialmente o más adelante, el planificador se quedará bloqueado leyendo del FIFO.
- Cuando el planificador recibe por el FIFO una petición de inserción, debe cargar la biblioteca e invocar su función de carga, que, a su vez, deberá instalar la función que realiza la tarea dentro de la lista del planificador.
- En el caso de recibir una solicitud de eliminación, el planificador debe invocar la función de descarga, que, por su parte, desinstalará la función de la lista del planificador. Por último, el planificador debe eliminar la biblioteca de su mapa de memoria.
- Siempre que la lista de tareas no esté vacía, el planificador invocará por orden de prioridad las sucesivas tareas y a continuación consultará el FIFO, de forma no bloqueante (puede usarse la llamada `select`), para ver si hay peticiones pendientes. En caso de que las haya, las procesa. En cualquier caso, después de esto, el planificador vuelve a ejecutar la lista de tareas, y así sucesivamente.

Para realizar las pruebas de la práctica se recomienda el desarrollo de bibliotecas dinámicas muy sencillas, de forma que la tarea consista simplemente en escribir su nombre, su argumento y su prioridad.

Por último, hay que resaltar que el modo de operación del programa desarrollado tiene cierta similitud, hasta cierto punto, con el esquema usado en Linux para permitir la carga dinámica de módulos en el *kernel* (mandatos `insmod` y `rmmmod`).

Interposición de bibliotecas dinámicas

En ocasiones puede ser necesario reescribir una función que ya está definida en una biblioteca dinámica que usa el programa, de manera que todas las llamadas a esa función que realice el programa (ya sea desde el propio código del programa o desde una de sus bibliotecas dinámicas) confluyan en la nueva versión de la función. A veces, se pretende que esa nueva versión oculte a la versión anterior. Sin embargo, en otras ocasiones, se desea que la nueva versión realice un determinado trabajo previo y luego invoque a la versión original. Este último ejercicio plantea el uso del mecanismo de las bibliotecas dinámicas para resolver este tipo de situaciones. En general, la estrategia se basará en definir la nueva versión de la función en una biblioteca dinámica e interponerla de manera que intercepte las llamadas a la biblioteca original.

En primer lugar, se comenta a grandes rasgos cómo se realiza la resolución de referencias a símbolos externos en un programa que usa bibliotecas dinámicas. El ejecutable tiene asociadas

una lista ordenada de las bibliotecas dinámicas que usa. En el caso de un programa en C, siempre estará incluida la biblioteca dinámica de dicho lenguaje (`libc`) en la última posición de la lista, sin necesidad de que se haya especificado en el mandato de montaje. Cuando se referencia un símbolo global dentro de un programa, esta referencia se resuelve usando la primera biblioteca de la lista que lo tenga definido, incluso aunque la referencia se haya hecho desde una biblioteca que estaba al final de la lista.

Como primer ejemplo, se propone la creación de una biblioteca dinámica que contenga el módulo de gestión de memoria dinámica desarrollado en una práctica previa de este capítulo (la biblioteca se denominará `mialloc.so`). Para ello, se debe generar especificando la opción `-shared`. A continuación se realizará un programa de prueba que haga uso de las funciones de reserva dinámica y se enlazará con la biblioteca dinámica generada. Al ejecutarlo, se podrá observar que se está haciendo uso de nuestras propias versiones de las funciones, quedando oculta la versión original de las mismas no sólo para el código del programa de prueba, sino para cualquier biblioteca que use el programa. Para poder observar este último aspecto, se recomienda incluir una llamada a `strdup` en el programa de prueba. Esta función de la biblioteca de C invoca a su vez a la función `malloc` para reservar espacio. Cuando se ejecute el programa de prueba, se podrá observar que esta llamada indirecta a `malloc` también la captura nuestra biblioteca.

Una vez realizado este ejemplo, a continuación se plantea un ejercicio basado en estos conceptos. Concretamente, se pretende el desarrollo de una biblioteca dinámica, denominada `traza_malloc.so`, que supervise las llamadas a las funciones de memoria dinámica que realiza un programa (`malloc`, `realloc` y `free`) y genere una traza de las mismas en un archivo para su posterior análisis una vez que haya concluido la ejecución del programa. Esta funcionalidad podría ser muy útil a la hora de depurar un programa que usa memoria dinámica, labor que resulta siempre muy difícil.

Como se puede apreciar, en este caso no se pretende ocultar la versión original de estas funciones, sino capturarlas, escribir la traza correspondiente en el archivo e invocar la función original. Para lograr este reenvío de la llamada se usará la función `dlsym`, especificando el valor `RTLD_NEXT` como primer parámetro. La biblioteca dinámica debe encargarse de crear el archivo donde se almacenarán las trazas. El alumno debe determinar qué información de traza se incluirá por cada llamada, de manera que sea adecuada para poder detectar en el análisis posterior errores en el uso de la memoria dinámica por parte del programa, como, por ejemplo, zonas que se reservan, pero nunca se liberan («goteras» de memoria) o intentos de liberar o redimensionar zonas que no se han reservado previamente. Puede resultar interesante que el alumno diseñe un algoritmo para analizar esta información de traza y detectar qué problemas aparecen en el uso de memoria dinámica por parte de un determinado programa.

Para terminar, hay que resaltar que las técnicas de interposición de bibliotecas que se han presentado en este apartado se han aplicado enlazando el programa con la biblioteca que realiza la interposición. Sin embargo, en muchos sistemas UNIX (como Solaris y Linux) existe la posibilidad de usarlas directamente sobre ejecutables sin necesidad de enlazarlos con la nueva biblioteca (téngase en cuenta que, para poder enlazar de nuevo el ejecutable, necesitamos disponer de su código objeto, lo que no es siempre posible). Para ello, se usa la variable de entorno `LD_PRELOAD`. Si se define esta variable de entorno de manera que contenga el nombre de una biblioteca dinámica, cuando se ejecute el programa se cargará automáticamente dicha biblioteca pudiendo interceptar las llamadas que considere oportuno. Para poder observar de forma práctica esta posibilidad, se recomienda realizar dos experimentos:

- Definir la variable de entorno `LD_PRELOAD` para que haga referencia a la biblioteca `mialloc.so`. A partir de ese momento, cualquier programa que se ejecute (incluidos los mandatos del sistema) usará nuestra biblioteca de memoria dinámica en vez de la del sistema.

- Definir la variable de entorno LD_PRELOAD para que se especifique la biblioteca traza_malloc.so. A partir de ese momento, cualquier programa que se ejecute usará la biblioteca de intercepción y, por tanto, se monitorizará su uso de la memoria dinámica.

Este mecanismo ofrece muchas posibilidades, permitiéndonos interferir en el comportamiento de programas aun sin disponer de su código fuente. Como ejemplo de esta poderosa característica, plantea una biblioteca de intercepción que capture las llamadas de *sockets* (primitivas send, sendto, recv y recvfrom) para incluir cifrado o compresión en la transmisión de datos. En el emisor se captura la llamada, se cifra o comprime y se invoca a la primitiva original y en el receptor se realiza el procesamiento complementario.

3.8.3. Código fuente de apoyo

Para facilitar la realización de la práctica se recomienda proporcionar a los alumnos un archivo que contenga ejemplos de programas que usan la técnica de carga explícita de bibliotecas dinámicas, como el existente en la página web del libro (practica-3.8.tgz).

3.8.4. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- memoria.txt: Memoria de la práctica, donde se incluirán los comentarios razonados de las distintas cuestiones que se han planteado en distintas partes de la práctica.
- procesa_archivos.c: Archivo fuente de C, donde se incluirá el programa procesa_archivos.
- planificador.c: Archivo fuente de C, donde se incluirá el programa planificador.
- insertar_tarea.c: Archivo fuente de C, donde se incluirá el programa insertar_tarea.
- eliminar_tarea.c: Archivo fuente de C, donde se incluirá el programa eliminar_tarea.
- traza_malloc.c: Archivo fuente de C, donde se incluirá el código de la biblioteca traza_malloc.so.

3.8.5. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

3.9. PRÁCTICA: MONITOR DEL USO DE MEMORIA DE UN PROGRAMA

3.9.1. Objetivos de la práctica

El principal objetivo es llevar a la práctica algunos de los conceptos estudiados en el tema teórico de gestión de memoria. Concretamente, la práctica abarcará aspectos de gestión de memoria tales como los siguientes:

- Estructuras de datos del gestor de memoria: tablas de regiones, tablas de páginas y tablas de marcos.
- Tratamiento del fallo de página.
- Algoritmos de reemplazo.
- Análisis del uso de memoria de los programas.

NIVEL: Diseño.

HORAS ESTIMADAS: 32.

3.9.2. Descripción de la práctica

La práctica va a consistir en el desarrollo de una aplicación (*memon*) que permita conocer el comportamiento de un programa con respecto a su uso de memoria dependiendo de la cantidad de memoria física disponible y del algoritmo de reemplazo utilizado.

Como se irá viendo paulatinamente, el desarrollo de este monitor va a implicar enfrentarse a situaciones muy similares a las que tiene que afrontar el gestor de memoria de un sistema operativo real. Además, el monitor que se pretende desarrollar tiene interés por sí mismo, ya que se trata de una herramienta que, por un lado, permite comparar diversos algoritmos de reemplazo, y por otro, facilita el análisis de cómo usan los programas reales la memoria dependiendo de sus características específicas. Nótese que el monitor no tendrá un carácter «intrusivo» en la ejecución del programa: el programa se completará realizando la labor para la que estaba programado. Simplemente, la ejecución habrá sido más lenta debido a la sobrecarga de la monitorización.

Por último, hay que resaltar que, por simplicidad, el estudio del uso de memoria de los programas no va a abarcar la región de pila, centrándose en las regiones de código, datos con y sin valor inicial, *heap* y las correspondientes a archivos proyectados, ya sea con proyección privada o compartida.

En el resto de este apartado se va a describir la idea en la que se fundamenta el modo de operación del monitor.

Cualquier programador que trabaje en el entorno C-UNIX conoce y «teme» el clásico mensaje: *Segmentation fault (core dumped)*. Detrás de este mensaje está la detección por parte del sistema operativo de un acceso a memoria inválido (ya sea un acceso a una dirección no asignada o un intento de realizar una operación no permitida sobre una determinada dirección de memoria) y la consiguiente generación de la señal *SEGV*, que, al no ser normalmente capturada por el programa, causa la terminación anómala del mismo.

Los programas convencionales generalmente no capturan esta señal. Sin embargo, su captura y tratamiento, junto con el uso de la llamada al sistema *mprotect*, conforman el punto de arranque de esta práctica.

La llamada al sistema *mprotect* permite establecer unos determinados permisos de acceso (o sea, especificar la protección, de ahí proviene su nombre) sobre un rango de direcciones del mapa del proceso que corresponda con un número entero de páginas. Se puede especificar permiso de lectura (*PROT_READ*), de ejecución (*PROT_EXEC*) y de escritura (*PROT_WRITE*). Como es habitual en UNIX, se puede especificar una combinación de permisos uniéndolos con el operador *|*. Además, se pueden quitar todos los permisos especificando *PROT_NONE*.

Nótese que con *mprotect* podemos modificar los permisos originales (y razonables) que estableció el sistema operativo. Así, por ejemplo, podríamos hacer que una determinada página de código tuviera permisos de escritura. Para más detalles sobre esta llamada, se recomienda consultar el manual. Sin embargo, conviene resaltar un último aspecto: los permisos no son acumulativos. Por ejemplo, si sobre una página que tenía previamente permiso de lectura se

realiza un `mprotect` especificando sólo `PROT_WRITE`, la página sólo quedará con permiso de escritura.

Mediante la captura de la señal `SEGV` y el uso de `mprotect`, el monitor puede supervisar la ejecución de un programa suponiendo que existe cierto número de marcos disponibles para el mismo y que se usa un determinado algoritmo de reemplazo. Esta supervisión se basa en las siguientes pautas:

- Inicialmente, se inhabilita el acceso a la página mediante el uso de `mprotect`. Podría decirse que «cubrimos» la página para que no pueda accederse.
- Cuando el programa intenta acceder a la página, se produce la señal `SEGV`. Es en el tratamiento de esta señal donde el monitor es consciente de que el programa ha accedido a la página y, por tanto, puede anotar este hecho. Es importante resaltar que, como parte de este tratamiento, el monitor debe devolver los permisos originales a la página mediante `mprotect` (o sea, «destapar» la página). Si no fuera así, el programa entraría en un bucle infinito, ya que, después de tratar la señal, se vuelve a ejecutar de nuevo la instrucción que la causó y, por tanto, si no se ha habilitado el acceso vuelve a producirse la señal. Esta repetición de la ejecución de la instrucción después del tratamiento de la señal es lo que hace que el monitor no sea «intrusivo», puesto que todas las instrucciones del programa original se ejecutan. Es interesante hacer notar que la ejecución de una misma instrucción puede generar varias señales de este tipo: tanto cuando se lee la instrucción como cuando se accede a cada uno de los operandos en memoria que pueda tener la misma, ya que pueden residir en páginas distintas.
- Nótese que puede ocurrir que el programa que se monitoriza sea erróneo y genere un error de acceso, ya sea por intentar acceder a una dirección inválida o debido a que intenta realizar una operación no permitida sobre una dirección válida (por ejemplo, intenta escribir sobre la región de código). Esta situación también causaría una señal `SEGV`, pero el monitor debe darse cuenta de que realmente se trata de un error de acceso, por lo que debe terminar la monitorización (no tiene sentido continuar supervisando un programa erróneo). Para poder distinguir si el acceso es válido, el monitor debe mantener estructuras de datos que reflejen el estado del mapa del programa que se supervisa. Estas estructuras van a ser similares, hasta cierto punto, a las tablas de regiones y tablas de páginas usadas por el gestor de memoria de un sistema operativo.
- Con respecto al número de marcos disponibles para la ejecución del proceso, este valor va a establecer el número máximo de páginas que pueden tener sus permisos habilitados. Siguiendo con el símil, este número determina cuántas páginas como mucho pueden estar «destapadas». Cada vez que se produce una señal `SEGV`, además de actualizar sus estadísticas de accesos, el monitor comprobará si el número de páginas habilitadas ha llegado a este límite (o sea, si se ha llenado la memoria). Si ocurre esta situación, habrá que inhabilitar mediante `mprotect` («expulsar») la página seleccionada por el algoritmo de reemplazo elegido. Nótese que esta gestión implica que el monitor debe almacenar una estructura de datos, similar a la tabla de marcos de un sistema operativo, que refleje qué marcos están libres y cuáles usados y por qué página.

Como resultado de la aplicación de estas pautas, el monitor puede obtener exactamente los mismos resultados que ocurrirían en un sistema real que dedicara el número de marcos especificados a la ejecución del proceso y que usara el algoritmo de reemplazo indicado.

El monitor presenta muchas similitudes con el gestor de memoria de un sistema operativo. De hecho, el instructor podrá darse cuenta de que se trata de una aplicación del principio de concreción en el que se basa el *hardware* virtual del *minikernel*: un fallo de página real producido por un acceso

inválido a una página con acceso inhabilitado (generado por la MMU del procesador), el sistema operativo lo convierte en una señal *SEGV*, que el monitor trata como un fallo de página del programa que supervisa. A continuación, para incidir en esta similitud entre el monitor y el gestor de memoria de un sistema operativo, se exponen de forma comparada diversos aspectos de su modo de operación:

- En un sistema operativo real con memoria virtual basada en paginación por demanda, cuando se crea una región todas sus páginas se marcan como *no residentes*. De manera similar, en el monitor se dejarán inaccesibles mediante el uso de *mprotect*.
- Cuando se accede a una página no residente en un sistema operativo real, se produce un fallo de página. En este caso se generará la señal *SEGV*. El tratamiento de *SEGV* tendrá muchos aspectos en común con el tratamiento de un fallo de página real.
- Cuando se trae una página a memoria en un sistema real, se marca como residente y, por tanto, los siguientes accesos no causarán fallos de página. En el monitor se usará *mprotect* para habilitar los próximos accesos.
- En un sistema real, la memoria física disponible limita el número de páginas que pueden estar residentes. Si hay un fallo y no hay ningún marco libre, se aplica un algoritmo de reemplazo que elige una página residente que se expulsa de memoria. De manera similar, el monitor prohibirá el acceso a la página seleccionada como «victima» por el algoritmo de reemplazo. Nótese que el monitor se encargará de que en cada momento sólo pueda estar habilitado el acceso a, como mucho, tantas páginas como marcos tenga la memoria física.
- Ambos usan estructuras de datos de similares características:
 - Una tabla de regiones que contenga las características de cada región. Téngase en cuenta que, por ejemplo, es necesario saber qué rango de direcciones ocupa cada región para poder diferenciar dentro del tratamiento de *SEGV* si se trata realmente de un fallo de página o en verdad se ha producido un acceso inválido (o sea, un verdadero *SEGV*).
 - Una tabla de páginas por cada región. Es necesario saber el estado de cada página de la región. Por ejemplo, hace falta conocer qué páginas están residentes y cuáles no.
 - Una tabla de marcos. Esta estructura permite conocer qué marcos están libres y cuáles ocupados, especificando qué página contiene (para poder invalidarla en caso de reemplazo).

Para llevar a cabo su labor, el monitor cuenta con la colaboración de un módulo de apoyo que se encargará en cada momento de informar al monitor de qué regiones forman el mapa del proceso y cuáles son sus características (dirección inicial, tamaño, protección, si está vinculada a un archivo o es anónima y si es compartida o privada). Este módulo se dedica a interceptar las llamadas del programa que puedan afectar al mapa de memoria del proceso para informar de ello al monitor. En concreto, el entorno de apoyo detecta e informa cuando se crea una nueva región en el mapa, cuando se elimina una región o cuando cambia el tamaño de una región existente. Hay que resaltar que el sistema operativo real se encargará de llevar a cabo todas estas operaciones sobre el mapa del proceso. El módulo de apoyo simplemente detecta cuando se producen y avisa al monitor del hecho.

Por tanto, una vez arrancado el programa cuya ejecución se pretende supervisar, el monitor sólo tomará el control debido a cuatro posibles eventos: se ha producido una señal *SEGV*, se ha creado o eliminado una región o bien ha cambiado su tamaño.

3.9.3. Organización del software del monitor

Dada la extensión y complejidad del monitor planteado, es conveniente organizarlo de manera modular. Antes de plantear una posible estructura del software del monitor, es conveniente co-

mentar que se recomienda no proporcionar al alumno el código fuente del módulo de apoyo, ya que consideramos que su disponibilidad no aporta ningún beneficio al alumno, pudiendo incluso causarle cierta confusión. Con respecto a los otros módulos que se describirán en este apartado, se propone adoptar una opción «minimalista» en el sentido de proporcionar al alumno una versión inicial de los mismos prácticamente vacía. El objetivo de esta estrategia es que el alumno se vea forzado a diseñar completamente las estructuras de datos requeridos por el monitor. Hay que recordar, sin embargo, que, dado que el instructor dispone de la solución completa de la práctica, puede decidir proporcionar al alumno una versión inicial más elaborada para disminuir la complejidad de la práctica y acortar su tiempo de desarrollo. A continuación se propone una posible organización del *software* del monitor.

Módulo principal (archivo `memon.c`)

Este módulo contendrá el programa principal del monitor. En la versión inicial proporcionada al alumno, sólo incluirá la lógica requerida para capturar la señal `SEGV` debido a la dificultad que presenta poder obtener la dirección que causó el fallo. Una vez obtenida dicha dirección, la rutina de tratamiento invoca directamente a la función `fallo_pagina` del módulo `fallo`.

Módulo de apoyo

Como se comentó previamente, este módulo se encarga de controlar la evolución del mapa del proceso informando al monitor cuando sea oportuno. Este módulo proporciona la función `ejecutar_programa` que será invocada por el monitor cuando, una vez iniciadas sus estructuras de datos, pretenda arrancar la ejecución del programa que se pretende supervisar. Asimismo, se dedica a interceptar las llamadas del programa a monitorizar que puedan afectar al mapa de memoria del proceso para informar de ello al monitor. En concreto, el entorno de apoyo detecta e informa de las siguientes situaciones:

- Creación de una nueva región en el mapa. Este evento se produce cuando el entorno de apoyo detecta que se ha creado una nueva región e invocará a la rutina `creacion_region` del módulo `mapa` para informarle de este evento y de las características de la nueva región. La creación de una nueva región puede estar asociada a distintas situaciones:
 - En la creación del programa que se desea supervisar (realizada en la rutina de apoyo `ejecutar_programa`) se crean las regiones iniciales del proceso y se informa de ello al monitor.
 - El programa a supervisar realiza una llamada `mmap`.
 - El programa a supervisar realiza una primera reserva de memoria dinámica que causa la creación del *heap*. Nótese que en esta práctica se considera al *heap* como una región inicialmente vacía que es independiente de la región de datos sin valor inicial.
- Eliminación de una región. El entorno ha detectado que se ha eliminado una región e informa al monitor invocando la función `eliminacion_region` del módulo `mapa` para informarle de este evento. Este evento va a ser consecuencia de que el programa ha ejecutado la llamada `munmap` o de que ha terminado su ejecución.
- Cambio del tamaño de una región existente. El entorno detecta un cambio de tamaño en una región y le informa el monitor llamando a su función `cambio_tam_region`. Este cambio de tamaño estará asociado a la evolución de la región del *heap*.

A continuación se incluyen los prototipos de estas funciones usadas por el entorno de apoyo para notificar los cambios en el mapa del proceso:

```
/* se informa de que se ha creado una región
   dir: dirección de comienzo de la región
   nodoi: archivo al que está vinculada (0 si es anónima)
   prot: permisos de acceso a la región
   tamano: tamaño de la región
   compartida: ¿es una región de tipo compartida?
*/
void creacion_region(void *dir, int nodoi, int prot,
                     int tamano, int compartida);

/* se informa de que se ha eliminado una región
   dir: dirección de comienzo de la región
*/
void eliminacion_region(const void *dir);

/* se informa de que ha cambiado el tamaño de una región
   dir: dirección de comienzo de la región
   tamano: nuevo tamaño de la región
*/
void cambio_tam_region(void *dir, int tam);
```

En el archivo apoyo.h, además del prototipo de ejecutar_programa, se ofrecen unas macros que facilitan la gestión de la máscara de protección usada por mprotect.

Módulo marcos

Este módulo contendrá la gestión de la tabla de marcos, incluyendo los algoritmos de reemplazo. Aunque inicialmente se proporciona una versión vacía de este módulo, se recomienda que ofrezca, entre otras, rutinas para crear la tabla de marcos, para reservar uno libre, para liberar uno ocupado, así como los algoritmos de reemplazo correspondientes.

Módulo mapa

Este módulo deberá incluir las operaciones relacionadas con la gestión del mapa del proceso tanto de sus regiones como de las páginas contenidas en las mismas. Por tanto, contendrá las definiciones de la tablas de regiones y de páginas junto con las funciones que las gestionan. En este módulo se implementarán las funciones que invocará el módulo de apoyo para informar sobre la evolución del mapa del proceso, creacion_region, cambio_tam_region y eliminacion_region, así como otras funciones que se considere oportuno.

Módulo fallo

Este módulo tendrá que incluir la rutina de tratamiento del «fallo de página» (denominada fallo_pagina), que se encargará de tratar este evento y de actualizar las estadísticas de acuerdo con el mismo. Esta rutina será invocada directamente desde la función de tratamiento de la señal SEGV incluida en el módulo principal.

3.9.4. Descripción de la funcionalidad que debe desarrollar el alumno

El monitor recibirá como argumentos el algoritmo de reemplazo que debe utilizar, el número de marcos disponibles (o sea, el tamaño de la memoria física) y el nombre del programa que se pretende ejecutar de forma supervisada junto con sus argumentos.

La información sobre el algoritmo de reemplazo estará contenida en el propio nombre del programa. Así, aunque sólo habrá un ejecutable, denominado memon, existirán enlaces a este ejecutable, de manera que el nombre del enlace haga referencia al algoritmo de reemplazo (memon_FIFO para un algoritmo FIFO y memon_reloj para el algoritmo del reloj).

Supóngase, por ejemplo, que se desea monitorizar en un sistema con ocho marcos y usando un algoritmo de reemplazo FIFO, el programa prueba que recibe como argumento los archivos (vector_resultado y vector_operando). Se ejecutaría el siguiente mandato:

```
memon_FIFO 8 prueba vector_resultado vector_operando
```

Si se pretende usar el algoritmo del reloj, habrá que ejecutar lo siguiente:

```
memon_reloj 8 prueba vector_resultado vector_operando
```

Con respecto a la salida producida por el monitor, una vez que esté completada la funcionalidad del mismo, ésta podría ser como la siguiente:

```
Fallos de página 42
Fallos no forzados 28
Fallos forzados 14
Fallos sin reemplazo 10
Fallos con reemplazo 32
Fallos sin lectura 3
Fallos con lectura archivo 27
Fallos con lectura swap 12
Escrituras en archivo 5
Escrituras en swap 11
```

En el resto de este apartado se expone de forma evolutiva qué funcionalidad se pide concretamente en esta práctica, describiendo tres versiones sucesivas del monitor con una complejidad incremental.

Versión inicial. Aplicación directa de la idea básica

Esta primera versión va a plasmar las ideas planteadas en el apartado anterior para construir un monitor que obtenga algunas estadísticas básicas sobre el uso de memoria de un programa usando un algoritmo de reemplazo sencillo como el FIFO. Esta primera versión del monitor deberá obtener las siguientes estadísticas:

- Número total de fallos de página causados por el programa.
- Número de fallos «no forzados» (causados por la falta de memoria física) y «forzados» (no son causados por la falta de memoria física, se deben simplemente al uso de paginación por demanda).
- Número de fallos que conllevan reemplazo (no se encuentra un marco libre) y número de fallos que no generan reemplazo.

Para realizar esta primera versión se deberían definir las estructuras de datos que representan las regiones, las páginas y los marcos. Con respecto a la tabla de marcos, simplemente, se debe recordar que en cada entrada, como mínimo, tendría que aparecer información sobre si está ocupado el marco correspondiente y, en caso de estarlo, qué página contiene. Por lo que se refiere a la tabla de regiones, debería incluir en cada entrada toda la información sobre la región que proporciona la llamada `creacion_region` del módulo de apoyo. Por último, sobre la tabla de páginas, hay que incluir en la definición de la entrada aquellos campos requeridos por la funcionalidad reducida de esta primera versión (por ejemplo, la información necesaria para distinguir los fallos forzados y no forzados).

En el módulo principal habrá que tratar los argumentos recibidos y realizar la iniciación de las estructuras de datos. Una vez que realiza esta labor, se encargará de arrancar el programa que se desea monitorizar llamando a la función `ejecutar_programa` del módulo de apoyo. Aunque queda a criterio del alumno, podría ser interesante incluir en este módulo la impresión de las estadísticas finales obtenidas por el monitor, así como el código que averigua cuál es el tamaño de la página (mediante la llamada `sysconf`) y lo carga en una variable global para evitar tener que estar haciendo la llamada al sistema continuamente.

En el módulo mapa habrá que implementar las rutinas que son invocadas desde el módulo de apoyo. En el caso de la creación, el monitor debe anotar las características de la región e inhabilitar el acceso a sus páginas. Cuando se trata de una eliminación, debe anotar este hecho en sus estructuras de datos liberando, además, los marcos que contuvieran páginas de una región siempre que ésta fuera de tipo privado. Nótese que, siguiendo la estrategia que usa la mayoría de los sistemas operativos, cuando se elimina una región compartida del mapa del proceso, no se van a eliminar de la memoria física sus páginas residentes, dando así oportunidad para que puedan usarlas otros procesos que comparten esa misma región (se mantiene esta estrategia aunque, evidentemente, esto no puede ocurrir en la práctica, ya que sólo hay un proceso). Es importante resaltar que la operación de eliminar una región no conlleva el uso de `mprotect`, puesto que el sistema operativo ya eliminó la región y si se hace un `mprotect` daría error. Por último, con respecto a la operación de cambio de tamaño, su tratamiento dependerá de si se trata de una expansión de la región o de una contracción. Si el tamaño aumenta, la zona añadida a la región tiene un tratamiento similar, hasta cierto punto, al que se realiza en la creación de una nueva región (hay que inhabilitar el acceso a esta zona expandida). En caso de que disminuya, el tratamiento de la zona que desaparece tendrá puntos en común con el que se realiza en la eliminación de una región (se deben liberar los marcos que contenían esta zona que ha desaparecido).

En el módulo marcos habrá que implementar la gestión de la tabla de marcos, así como el algoritmo de reemplazo FIFO.

En el módulo fallo habrá que realizar la primera versión de la rutina de fallo de página, que deberá tener la estructura típica de una rutina de este tipo. Por lo que se refiere a las estadísticas, sólo se calcularán los fallos totales, los fallos forzados y no forzados y los fallos con y sin reemplazo. Además, habrá que detectar los accesos a direcciones de memoria inválidas. En caso de que se produzcan, se sacará un mensaje por la salida de error y se terminará inmediatamente la ejecución del programa.

Por último, hay que resaltar que el monitor nos permite apreciar en la práctica la influencia del número de marcos asignados a un proceso (su conjunto residente) con el número de fallos de página. Para ello, es recomendable ejecutar el monitor sobre un programa de prueba variando el número de marcos disponible:

- Aumentar el número de marcos hasta que no haya fallos forzados.
- Disminuir el número de marcos para comprobar cómo se dispara exponencialmente el número de fallos de página (como nos enseña la teoría, lo que está ocurriendo es que el

conjunto de trabajo del proceso no cabe en el conjunto residente). ¿Qué ocurre cuando especificamos un solo marco? Intente analizar lo que está ocurriendo.

Versión intermedia. Control de la modificación de las páginas

Se puede mejorar la funcionalidad del monitor y conseguir estadísticas más detalladas del uso de memoria si logramos llevar la cuenta de cuáles de las páginas residentes han sido modificadas y cuáles no.

Se trata, por tanto, de gestionar una información equivalente a la que proporciona el bit de modificado de una MMU. La estrategia que se va a utilizar no es nueva. Es la misma que se ha usado en sistemas operativos reales cuando la MMU del procesador no incluía un bit de modificado. A continuación se describe esta estrategia:

- Cuando en un fallo de página (o sea, en el tratamiento del `SEGV` en el caso de nuestro monitor) se trae una página a memoria (o sea, se habilita su acceso con `mprotect`), se le especifica una protección que corresponde con la de la región, pero quitándole el permiso de escritura.
- Si se produce un fallo y se comprueba que la página ya estaba residente, esto implicaría que se ha intentado escribir sobre ella. Por tanto, se considerará activado el bit de modificado y se establecerá directamente la protección original de la región. Nótese que antes de activar el bit de modificado y restaurar la protección original habría que comprobar que realmente la región tiene permiso de escritura, ya que, en caso contrario, se trataría de un acceso inválido (por ejemplo, un programa que escribe sobre su región de código). En caso de que se produzca, se sacará un mensaje por la salida de error y se terminará inmediatamente la ejecución del programa. Un aspecto importante es que el monitor no incluirá en ninguna de sus estadísticas este fallo, ya que se trata de un fallo «artificial» usado para la gestión del bit de modificado.

Conviene resaltar que con la inclusión de este nuevo mecanismo, una única escritura en memoria puede causar dos fallos. En el primero, «se trae la página a memoria», pero no se habilita permiso de escritura. Al repetirse la misma instrucción (nótese que después de tratar `SEGV` se repite la misma instrucción, ya que el contador de programa sigue apuntando a la instrucción que causó el fallo), produce un segundo fallo, que el monitor no incluye en sus estadísticas, activándose el bit de modificado. La gestión de este bit de modificado permite obtener más estadísticas del comportamiento del programa:

- Número de fallos de página que no implican lectura, número de fallos que producen una lectura de archivo y número de fallos que provocan una lectura del *swap*.
- Número de escrituras a archivo y número de escrituras a *swap*.

Hay que tener en cuenta que una página modificada de una región privada se escribe en el *swap* cuando se expulsa, mientras que si se trata de una página de una región compartida siempre se usa el archivo como soporte.

Para implementar esta funcionalidad, en el módulo de gestión del mapa se deberán incluir campos adicionales en la definición de la entrada de la tabla de páginas (como mínimo, el propio bit de modificado) y añadir nuevas funciones si se considera oportuno.

En la rutina de fallo habrá que incluir la lógica de gestión del bit de modificación e implementar las nuevas estadísticas. Para ello habrá que tener en cuenta si la página pertenece a una región anónima o vinculada a archivo y si es privada o compartida.

Asimismo, habrá que detectar los errores debidos a operaciones de memoria no permitidas. Concretamente, los accesos de escritura a regiones que no lo permiten.

Versión final. Algoritmo del reloj

De todos son conocidas las limitaciones del algoritmo FIFO y las buenas prestaciones del algoritmo del reloj a pesar de su relativa simplicidad. Por tanto, en esta última versión de *memon* se plantea la inclusión de este nuevo algoritmo de reemplazo. Además de su programación, la implementación de este algoritmo presenta una dificultad adicional: requiere el uso de un bit de referencia.

Hasta ahora no se había planteado este requisito dado que el algoritmo FIFO no usa este bit. Para conseguir implementar este bit de referencia tenemos que recurrir nuevamente a un algoritmo similar al usado para el bit de modificado. Como curiosidad, es interesante resaltar que la MMU del procesador VAX donde se implementó el UNIX BSD original no tenía bit de referencia y, por tanto, se usó una estrategia similar a la descrita a continuación:

- Cuando se «trae» una página a memoria, se marca como residente y referenciada.
- Cuando el algoritmo del reloj solicita desactivar el bit de referencia de una página, habrá que deshabilitar el acceso con *mprotect*, pero manteniendo la página como residente.
- Si llega un fallo y la página está residente y no referenciada, se activa el bit de referencia y se habilita el acceso oportuno. Como ocurría con la implementación del bit de modificado, el monitor no incluirá en ninguna de sus estadísticas este fallo, ya que se trata de un fallo «artificial».

Nótese que esta estrategia tiene que funcionar de manera conjunta con la correspondiente a la gestión del bit de modificado.

La inclusión del algoritmo del reloj implica, en primer lugar, modificar el archivo *marcos.c* para incluir el algoritmo del reloj.

Además, en el módulo de gestión del mapa se deberán incluir campos adicionales en la definición de la entrada de la tabla de páginas (como mínimo, el propio bit de referencia) y añadir nuevas funciones si se considera oportuno. Asimismo, en la rutina de fallo habrá que incluir la lógica de gestión del bit de referencia.

3.9.5. Código fuente de apoyo

Para facilitar la realización de la práctica, se dispone en la página web del libro del archivo *practica 3.9.tgz*, que contiene el código fuente de apoyo de la práctica. Al extraer su contenido, se crea el directorio *practica 3.9*, donde se debe desarrollar la práctica. Dentro de este directorio se encuentran los siguientes archivos:

- *Makefile*: *Makefile* del monitor. Genera un ejecutable denominado *memon*.
- *memon.c*: archivo principal del monitor.
- *apoyo.o*: archivo objeto que contiene las funciones de apoyo.
- *apoyo.h*: archivo que contiene los prototipos de las funciones de apoyo, así como unas macros de utilidad.
- *marcos.h*: archivo que contiene los prototipos del módulo de gestión de marcos.
- *marcos.c*: archivo del módulo de gestión de marcos.
- *mapa.h*: archivo que contiene los prototipos del módulo de gestión del mapa.

- mapa.c: archivo del módulo de gestión del mapa del proceso.
- fallo.c: archivo que contiene la rutina que trata el fallo de página.
- memon_FIFO: enlace al ejecutable memon. Será el nombre usado para ejecutar el monitor cuando se desea aplicar el algoritmo FIFO.
- memon_reloj: enlace al ejecutable memon. Será el nombre usado para ejecutar el monitor cuando se desea aplicar el algoritmo del reloj.

3.9.6. Recomendaciones generales

Se recomienda usar los programas desarrollados en las otras prácticas del tema como ejemplos de programas para monitorizar.

3.9.7. Entrega de documentación

Se recomienda que el alumno entregue un archivo que contenga la memoria de la práctica. Dada la envergadura de la práctica y la libertad que tiene el alumno a la hora de desarrollarla, en este archivo el alumno deberá incluir una descripción del diseño de la misma. Además, el alumno deberá entregar todos los archivos de la práctica donde se habrá incluido la funcionalidad pedida.

3.9.8. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.

4

Comunicación y sincronización de procesos

En este capítulo se presentan prácticas relacionadas con la comunicación y sincronización de procesos. El objetivo fundamental es que el alumno entienda el problema fundamental que plantea la ejecución de procesos concurrentes que cooperan entre sí, así como la necesidad de comunicar y sincronizar correctamente dichos procesos. También se pretende que el alumno comprenda el problema de los interbloqueos y conozca cómo detectarlos.

4.1. CONCEPTOS BÁSICOS

Los procesos que ejecutan de forma concurrente en un sistema se pueden clasificar como procesos independientes o cooperantes. Un proceso **independiente** es aquel que ejecuta sin requerir la ayuda o cooperación de otros procesos. Un claro ejemplo de procesos independientes son los diferentes intérpretes de mandatos que se ejecutan de forma simultánea en un sistema. Los procesos son **cooperantes** cuando están diseñados para trabajar conjuntamente en alguna actividad, para lo que deben ser capaces de comunicarse e interactuar entre ellos.

Tanto si los procesos son independientes como cooperantes, pueden producirse una serie de interacciones entre ellos. Estas interacciones pueden ser de dos tipos:

- Interacciones motivadas porque los procesos **comparten** o **compiten** por el acceso a recursos físicos o lógicos. Esta situación aparece en los distintos tipos de procesos anteriormente comentados. Por ejemplo, dos procesos totalmente independientes pueden competir por el acceso a disco. En este caso, el sistema operativo deberá encargarse de que los dos procesos accedan ordenadamente sin que se cree ningún conflicto. Esta situación también aparece cuando varios procesos desean modificar el contenido de un registro de una base de datos. Aquí es el gestor de la base de datos el que se tendrá que encargar de ordenar los distintos accesos al registro.
- Interacción motivada porque los procesos se **comunican** y **sincronizan** entre sí para alcanzar un objetivo común. Por ejemplo, un compilador se puede construir mediante dos procesos: el compilador propiamente dicho, que se encarga de generar código ensamblador, y el proceso ensamblador, que obtiene código en lenguaje máquina a partir del ensamblador. En este ejemplo puede apreciarse la necesidad de comunicar y sincronizar a los dos procesos.

Estos dos tipos de interacciones obligan al sistema operativo a incluir mecanismos y servicios que permitan la comunicación y la sincronización entre procesos.

4.1.1. Problemas clásicos de comunicación y sincronización

La interacción entre procesos se plantea en una serie de situaciones clásicas de comunicación y sincronización. Estas situaciones, junto con sus problemas, se describen a continuación para demostrar la necesidad de comunicar y sincronizar procesos. Algunos de estos problemas constituirán el núcleo fundamental de las prácticas propuestas en este capítulo.

4.1.1.1. El problema de la sección crítica

Éste es uno de los problemas que con mayor frecuencia aparece cuando se ejecutan procesos concurrentes tanto si son cooperantes como independientes. Considerese un sistema compuesto por n procesos $\{P_1, P_2, \dots, P_N\}$ en el que cada uno tiene un fragmento de código, que se denomina **sección crítica**. Dentro de la sección crítica, los procesos pueden estar accediendo y modificando variables comunes, registros de una base de datos, un archivo, en general cualquier recurso compartido. La característica más importante de este sistema es que cuando un proceso se encuentra ejecutando código de la sección crítica, ningún otro proceso puede ejecutar en su sección.

Para resolver el problema de la sección crítica es necesario utilizar algún *mecanismo de sincronización* que permita a los procesos cooperar entre ellos sin problemas. Este mecanismo debe proteger el código de la sección crítica y su funcionamiento básico es el siguiente:

- Cada proceso debe solicitar permiso para entrar en la sección crítica mediante algún fragmento de código, que se denomina de forma genérica *entrada en la sección crítica*.
- Cuando un proceso sale de la sección crítica debe indicarlo mediante otro fragmento de código, que se denomina *salida de la sección crítica*. Este fragmento permitirá que otros procesos entren a ejecutar el código de la sección crítica.

La estructura general, por tanto, de cualquier mecanismo que pretenda resolver el problema de la sección crítica es la siguiente:

```
Entrada en la sección crítica
Código de la sección crítica
Salida de la sección crítica
```

Cualquier solución que se utilice para resolver este problema debe cumplir los tres requisitos siguientes:

- **Exclusión mutua**: si un proceso está ejecutando código de la sección crítica, ningún otro proceso lo podrá hacer.
- **Progreso**: si ningún proceso está ejecutando dentro de la sección crítica, la decisión de qué proceso entra en la sección se hará sobre los procesos que desean entrar. Los procesos que no quieren entrar no pueden formar parte de esta decisión. Además, esta decisión debe realizarse en tiempo finito.
- **Espera acotada**: debe haber un límite en el número de veces que se permite que los demás procesos entren a ejecutar código de la sección crítica después de que un proceso haya efectuado una solicitud de entrada y antes de que se conceda la suya.

4.1.1.2. Problema del productor-consumidor

El problema del productor-consumidor es uno de los problemas más habituales que surge cuando se programan aplicaciones utilizando procesos concurrentes. En este tipo de problemas, uno o más procesos, que se denominan *productores*, generan cierto tipo de datos que son utilizados o consumidos por otros procesos, que se denominan *consumidores*. Un claro ejemplo de este tipo de problemas es el del compilador que se describió anteriormente. En este ejemplo el compilador hace las funciones de productor al generar el código ensamblador que consumirá el proceso ensamblador para generar el código máquina. En la Figura 4.1 se representa la estructura clásica de este tipo de procesos.

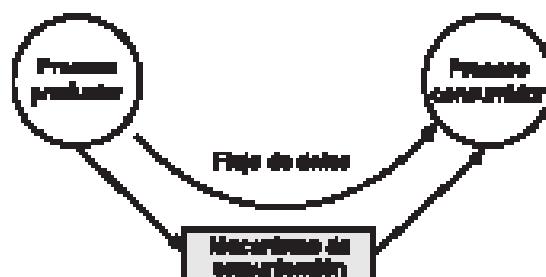


Figura 4.1. Estructura clásica de procesos productor-consumidor.

En esta clase de problemas es necesario disponer de algún mecanismo de comunicación que permita a los procesos productor y consumidor intercambiar información. Ambos procesos, además, deben sincronizar su acceso al mecanismo de comunicación para que la interacción entre ellos no sea problemática: cuando el mecanismo de comunicación se llene, el proceso productor se deberá quedar bloqueado hasta que haya hueco para seguir insertando elementos. A su vez, el proceso consumidor deberá quedarse bloqueado cuando el mecanismo de comunicación esté vacío, ya que en este caso no podrá continuar su ejecución al no disponer de información a consumir. Por tanto, este tipo de problema requiere servicios para que los procesos puedan comunicarse y servicios para que se sincronicen a la hora de acceder al mecanismo de comunicación.

4.1.1.3. El problema de los lectores-escritores

En este problema existe un determinado objeto (véase Figura 4.2), que puede ser un archivo, un registro dentro de un archivo, etc., que va a ser utilizado y compartido por una serie de procesos concurrentes. Algunos de estos procesos sólo van a acceder al objeto sin modificarlo, mientras que otros van a acceder al objeto para modificar su contenido. Esta actualización implica leerlo, modificar su contenido y escribirlo. A los primeros procesos se les denomina *lectores* y a los segundos se les denomina *escritores*. En este tipo de problemas existe una serie de restricciones que han de seguirse:

- Sólo se permite que un escritor tenga acceso al objeto al mismo tiempo. Mientras el escritor esté accediendo al objeto, ningún otro proceso lector ni escritor podrá acceder a él.
- Se permite, sin embargo, que múltiples lectores tengan acceso al objeto, ya que ellos nunca van a modificar el contenido del mismo.

En este tipo de problemas es necesario disponer de servicios de sincronización que permitan a los procesos lectores y escritores sincronizarse adecuadamente en el acceso al objeto.

4.1.1.4. Comunicación cliente-servidor

En el modelo cliente-servidor, los procesos llamados servidores ofrecen una serie de servicios a otros procesos que se denominan clientes (véase Figura 4.3). El proceso servidor puede residir en la misma máquina que el cliente o en una distinta, en cuyo caso la comunicación deberá realizarse a través de una red de interconexión. Muchas aplicaciones y servicios de red, como el correo electrónico y la transferencia de archivos, se basan en este modelo.

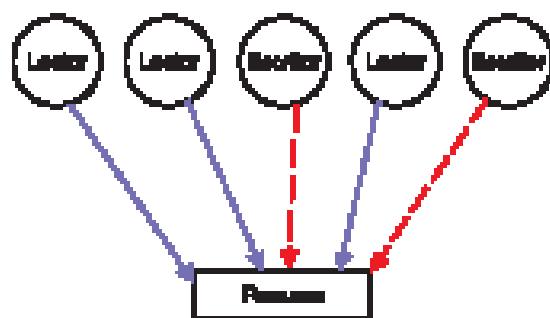


Figura 4.2. Procesos lectores y escritores.

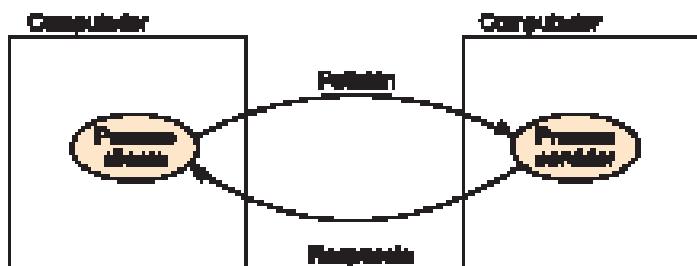


Figura 4.3. Comunicación cliente-servidor.

En este tipo de aplicaciones es necesario que el sistema operativo ofrezca servicios que permitan comunicarse a los procesos cliente y servidor. Cuando los procesos ejecutan en la misma máquina, se pueden emplear técnicas basadas en memoria compartida o archivos. Sin embargo, este modelo de comunicación suele emplearse en aplicaciones que ejecutan en computadores que no comparten memoria y, por tanto, se usan técnicas basadas en paso de mensajes.

4.1.2. Mecanismos y servicios de comunicación

En esta sección se presentan los mecanismos y servicios que ofrece POSIX para la comunicación y sincronización de procesos y que se utilizarán en la realización de las prácticas que se proponen en este capítulo.

4.1.2.1. Tuberías (pipes)

Una tubería es un mecanismo de comunicación y sincronización. Desde el punto de vista de su utilización, es como un sendarchivo mantenido por el sistema operativo. Conceptualmente, cada proceso ve la tubería como un conducto con dos extremos, uno de los cuales se utiliza para escribir o insertar datos y el otro para extraer o leer datos de la tubería. La escritura se realiza mediante el servicio que se utiliza para escribir datos en un archivo. De igual forma, la lectura se lleva a cabo mediante el servicio que se emplea para leer de un archivo.

El flujo de datos en la comunicación empleando tuberías es unidireccional y FIFO, esto quiere decir que los datos se extraen de la tubería (mediante la operación de lectura) en el mismo orden en el que se insertaron (mediante la operación de escritura). La Figura 4.4 representa dos procesos que se comunican de forma unidireccional utilizando una tubería.

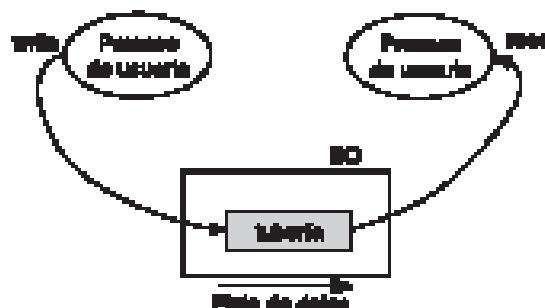


Figura 4.4. Comunicación unidireccional utilizando una tubería.

Un *pipe* en POSIX no tiene nombre y, por tanto, sólo puede ser utilizado entre los procesos que lo heredan a través de la llamada `fork()`. A continuación se describen los servicios que permiten crear y acceder a los datos de un *pipe*.

Creación de un pipe

El servicio que permite crear un *pipe* es el siguiente:

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

Esta llamada devuelve dos descriptores de archivos (véase la Figura 4.5) que se utilizan como identificadores:

- `filedes[0]`, descriptor de archivo que se emplea para leer del *pipe*.
- `filedes[1]`, descriptor de archivo que se utiliza para escribir en el *pipe*.

La llamada `pipe` devuelve 0 si fue bien y -1 en caso de error.

Cierre de un pipe

El cierre de cada uno de los descriptores que devuelve la llamada `pipe` se consigue mediante el servicio `close`, que también se emplea para cerrar cualquier archivo. Su prototipo es:

```
int close(int fd);
```

El argumento de `close` indica el descriptor de archivo que se desea cerrar. La llamada devuelve 0 si se ejecutó con éxito. En caso de error, devuelve -1.

Escrivir en un pipe

El servicio para escribir datos en un *pipe* en POSIX es el siguiente:

```
int write(int fd, char *buffer, int n);
```

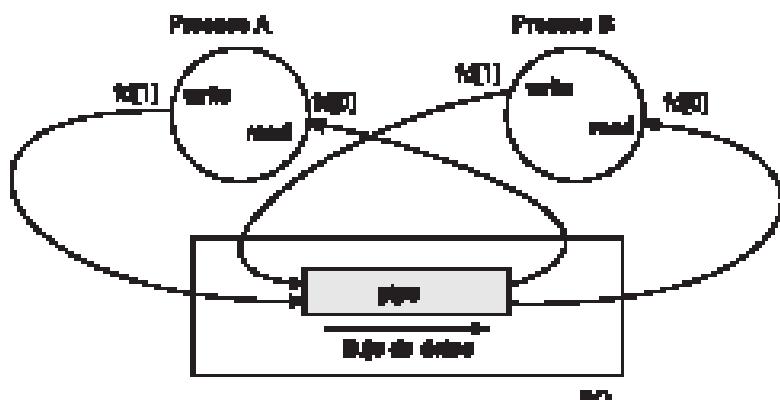


Figura 4.5. Tuberías POSIX entre dos procesos.

Este servicio también se emplea para escribir datos en un archivo. El primer argumento representa el descriptor de archivo que se emplea para escribir en un *pipe*. El segundo argumento especifica el *buffer* de usuario donde se encuentran los datos que se van a escribir al *pipe*. El último argumento indica el número de bytes a escribir. Los datos se escriben en el *pipe* en orden FIFO. La semántica de esta llamada es la siguiente:

- Si la tubería se encuentra llena o se llena durante la escritura, la operación bloquea al proceso escritor hasta que se pueda completar.
- Si no hay ningún proceso con la tubería abierta para lectura, la operación devuelve el correspondiente error. Este error se genera mediante el envío al proceso que intenta escribir de la señal **SIGPIPE**.
- Una operación de escritura sobre una tubería se realiza de forma atómica, es decir, si dos procesos intentan escribir de forma simultánea en una tubería sólo uno de ellos lo hará, el otro se bloqueará hasta que finalice la primera escritura.

Lectura de un pipe

Para leer datos de un *pipe* se utiliza el siguiente servicio, también empleado para leer datos de un archivo:

```
int read(int fd, char *buffer, int n);
```

El primer argumento indica el descriptor de lectura del *pipe*. El segundo argumento especifica el *buffer* de usuario donde se van a situar los datos leídos del *pipe*. El último argumento indica el número de bytes que se desean leer del *pipe*. La llamada devuelve el número de bytes leídos. En caso de error, la llamada devuelve -1. Las operaciones de lectura siguen la siguiente semántica:

- Si la tubería está vacía, la llamada bloquea al proceso en la operación de lectura hasta que algún proceso escriba datos en la misma.
- Si la tubería almacena M bytes y se quieren leer n bytes, entonces:
 - Si $M > n$, la llamada devuelve n bytes y elimina de la tubería los datos solicitados.
 - Si $M < n$, la llamada devuelve M bytes y elimina los datos disponibles en la tubería.
- Si no hay escritores y la tubería está vacía, la operación devuelve fin de archivo (la llamada `read` devuelve cero). En este caso, la operación no bloquea al proceso.
- Al igual que las escrituras, las operaciones de lectura sobre una tubería son atómicas. En general, la atomicidad en las operaciones de lectura y escritura sobre una tubería se asegura siempre que el número de datos involucrados en las anteriores operaciones sea menor que el tamaño de la misma.

4.1.2.2. Semáforos

Un semáforo es un mecanismo de sincronización que se utiliza generalmente en sistemas con memoria compartida, bien sea un monoprocesador o un multiprocesador. Su uso en un multicomputador depende del sistema operativo en particular. Un semáforo es un objeto con un valor entero al que se le puede asignar un valor inicial no negativo y al que sólo se puede acceder utilizando dos operaciones atómicas: `wait` y `signal` (también llamadas `down` o `up`, respectivamente). Las definiciones de estas dos operaciones son las siguientes:

```

wait(s) {
    s = s - 1;
    if (s < 0)
        Bloquear al proceso;
}

signal(s) {
    s = s + 1;
    if (s <= 0)
        Desbloquear a un proceso bloqueado en la operación wait;
}

```

El número de procesos que en un instante determinado se encuentran bloqueados en una operación `wait` viene dado por el valor absoluto del semáforo si es negativo. Cuando un proceso ejecuta la operación `signal`, el valor del semáforo se incrementa. En el caso de que haya algún proceso bloqueado en una operación `wait` anterior, se desbloqueará a un solo proceso.

Las operaciones `wait` y `signal` son dos operaciones genéricas que deben particularizarse en cada sistema operativo. A continuación se presentan los servicios que ofrece el estándar POSIX para trabajar con semáforos.

En POSIX, un semáforo se identifica mediante una variable del tipo `sem_t`. El estándar POSIX define dos tipos de semáforos:

- **Semáforos sin nombre.** Permiten sincronizar a los procesos ligeros que ejecutan dentro de un mismo proceso o a los procesos que lo heredan a través de la llamada `fork`.
- **Semáforos con nombre.** En este caso, el semáforo lleva asociado un nombre que sigue la convención de nombrado que se emplea para archivos. Con este tipo de semáforos se pueden sincronizar procesos sin necesidad de que tengan que heredar el semáforo utilizando la llamada `fork`.

Creación de un semáforo sin nombre

Todos los semáforos en POSIX deben iniciarse antes de su uso. La función `sem_init` permite iniciar un semáforo sin nombre. El prototipo de este servicio es el siguiente:

```
int sem_init(sem_t *sem, int shared, int val);
```

Con este servicio se crea y se asigna un valor inicial a un semáforo sin nombre. El primer argumento identifica la variable de tipo semáforo que se quiere utilizar. El segundo argumento indica si el semáforo se puede utilizar para sincronizar procesos ligeros o cualquier otro tipo de proceso. Si `shared` es 0, el semáforo sólo puede utilizarse entre los procesos ligeros creados dentro del proceso que inicia el semáforo. Si `shared` es distinto de 0, entonces se puede utilizar para sincronizar procesos que lo hereden por medio de la llamada `fork`. El tercer argumento representa el valor que se asigna inicialmente al semáforo.

Destrucción de un semáforo sin nombre

Con este servicio se destruye un semáforo sin nombre previamente creado con la llamada `sem_init`. Su prototipo es el siguiente:

```
int sem_destroy(sem_t *sem)
```

Creación y apertura de un semáforo con nombre

El servicio `sem_open` permite crear o abrir un semáforo con nombre. La función que se utiliza para invocar este servicio admite dos modalidades, según se utilice para crear el semáforo o simplemente abrir uno existente. Estas modalidades son las siguientes:

```
sem_t *sem_open(char *name, int flag, mode_t mode, int val);
sem_t *sem_open(char *name, int flag);
```

Un semáforo con nombre posee un nombre, un dueño y derechos de acceso similares a los de un archivo. El nombre de un semáforo es una cadena de caracteres que sigue la convención de nombrado de un archivo. La función `sem_open` establece una conexión entre un semáforo con nombre y una variable de tipo semáforo.

El valor del segundo argumento determina si la función `sem_open` accede a un semáforo previamente creado o si crea un nuevo. Un valor 0 en `flag` indica que se quiere utilizar un semáforo que ya ha sido creado, en este caso no es necesario los dos últimos parámetros de la función `sem_open`. Si `flag` tiene un valor `O_CREAT`, requiere los dos últimos argumentos de la función. El tercer parámetro especifica los permisos del semáforo que se va a crear, de la misma forma que ocurre en la llamada `open` para archivos. El cuarto parámetro especifica el valor inicial del semáforo.

POSIX no requiere que los semáforos con nombre se correspondan con entradas de directorio en el sistema de archivos, aunque sí pueden aparecer.

Cierre de un semáforo con nombre

Cierra un semáforo con nombre rompiendo la asociación que tenía un proceso con un semáforo. El prototipo de la función es:

```
int sem_close(sem_t *sem);
```

Borrado de un semáforo con nombre

Elimina del sistema un semáforo con nombre. Esta llamada pospone la destrucción del semáforo hasta que todos los procesos que lo están utilizando lo hayan cerrado con la función `sem_close`. El prototipo de este servicio es:

```
int sem_unlink(char *name);
```

Operación wait

La operación `wait` en POSIX se consigue con el siguiente servicio:

```
int sem_wait(sem_t *sem);
```

Operación signal

Este servicio se corresponde con la operación `signal` sobre un semáforo. El prototipo de este servicio es:

```
int sem_post(sem_t *sem);
```

Todas las funciones que se han descrito devuelven un valor 0 si la función se ha ejecutado con éxito o -1 en caso de error.

4.1.2.3. Mutex y variables condicionales

Los *mutex* y las variables condicionales son mecanismos especialmente concebidos para la sincronización de procesos ligeros. Un *mutex* es el mecanismo de sincronización de procesos ligeros más sencillo y eficiente. Los *mutex* se emplean para obtener acceso exclusivo a recursos compartidos y para asegurar la exclusión mutua sobre secciones críticas.

Sobre un *mutex* se pueden realizar dos operaciones atómicas básicas:

- **lock:** intenta bloquear el *mutex*. Si el *mutex* ya está bloqueado por otro proceso, el proceso que realiza la operación se bloquea. En caso contrario, se bloquea el *mutex* sin bloquear al proceso.
- **unlock:** desbloquea el *mutex*. Si existen procesos bloqueados en él, se desbloqueará a uno de ellos, que será el nuevo proceso que adquiera el *mutex*. La operación *unlock* sobre un *mutex* debe ejecutarla el proceso ligero que adquirió con anterioridad el *mutex* mediante la operación *lock*. Esto es diferente a lo que ocurre con las operaciones *wait* y *signal* sobre un semáforo.

El siguiente segmento de pseudocódigo utiliza un *mutex* para proteger el acceso a una sección crítica.

```
lock(m); /* solicita la entrada en la sección crítica */
< sección crítica >
unlock(n); /* salida de la sección crítica */
```

En la Figura 4.6 se representa de forma gráfica una situación en la que dos procesos ligeros intentan acceder de forma simultánea a ejecutar código de una sección crítica utilizando un *mutex* para protegerla.

Dado que las operaciones *lock* y *unlock* son atómicas, sólo un proceso conseguirá bloquear el *mutex* y podrá continuar su ejecución dentro de la sección crítica. El segundo proceso se bloqueará hasta que el primero libere el *mutex* mediante la operación *unlock*.

Una **variable condicional** es una variable de sincronización asociada a un *mutex* que se utiliza para bloquear a un proceso hasta que ocurra algún suceso. Las variables condicionales tienen dos operaciones atómicas para esperar y señalizar:

- **c_wait:** bloquea al proceso que ejecuta la llamada y le expulsa del *mutex* dentro del cual se ejecuta y al que está asociado la variable condicional, permitiendo que algún otro proceso adquiera el *mutex*. El bloqueo del proceso y la liberación del *mutex* se realiza de forma atómica.
- **c_signal:** desbloquea a uno o varios procesos suspendidos en la variable condicional. El proceso que se despierta compite de nuevo por el *mutex*.

A continuación se va a describir una situación típica en la que se utilizan los *mutex* y las variables condicionales de forma conjunta. Supóngase que una serie de procesos compiten por el acceso a una sección crítica. En este caso es necesario un *mutex* para proteger la ejecución de dicha sección crítica. Una vez dentro de la sección crítica, puede ocurrir que un proceso no pueda continuar su ejecución dentro de la misma debido a que no se cumple una determinada condición; por ejemplo, se quiere insertar elementos en un *buffer* común y éste se encuentra lleno. En esta situación, el proceso debe bloquearse, puesto que no puede continuar su ejecución. Además, debe liberar el

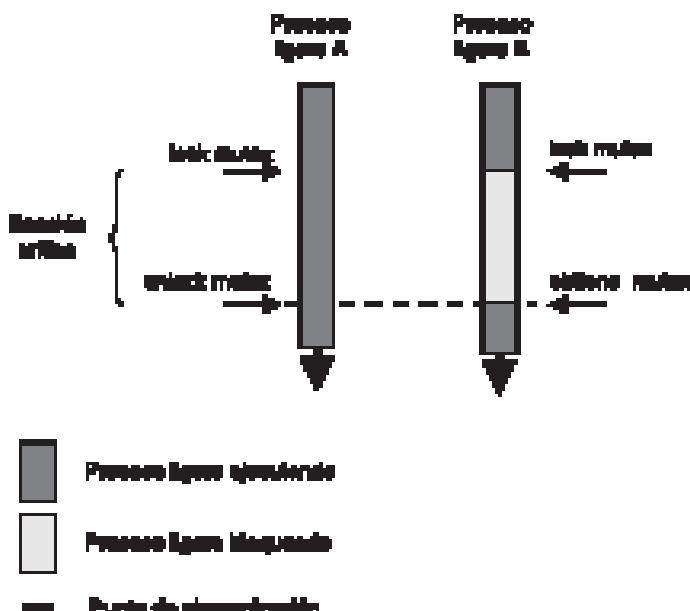


Figura 4.6. Ejemplo de omisiones en una sección crítica.

añadir para permitir que otro proceso entre en la sección crítica y pueda modificar la situación que bloqueó al proceso, en este caso eliminar un elemento del buffer común para hacer hueco.

Para conseguir este funcionamiento es necesario utilizar una o más variables compartidas que se utilizarán como predicado lógico y que el proceso consultará para decidir su bloqueo o no. El fragmento de código que se debe emplear en este caso es el siguiente:

```
lock(m);
/* código de la sección crítica */
while (condición == FALSE)
    c_wait(c, m);
/* resto de la sección crítica */
unlock(n);
```

En el fragmento anterior, *m* es el mutex que se utiliza para proteger el acceso a la sección crítica y *c* la variable condicional que se emplea para bloquear el proceso y abandonar la sección crítica. Cuando el proceso que está ejecutando dentro de la sección evalúa la condición y ésta es falsa, se bloquea mediante la operación *c_wait* y libera el mutex permitiendo que otro proceso entre en ella.

El proceso bloqueado permanecerá en esta situación hasta que algún otro proceso modifique alguna de las variables compartidas que le permitan continuar. El fragmento de código que debe ejecutar este otro proceso debe seguir el modelo siguiente:

```
lock(m);
/* código de la sección crítica */
/* se modifica la condición y ésta se hace TRUE */
condición = TRUE;
c_signal(c);
unlock(n);
```

En este caso, el proceso que hace cierta la condición ejecuta la operación `c_signal` sobre la variable condicional despertando a un proceso bloqueado en dicha variable. Cuando el proceso ligero que espera en una variable condicional se desbloquea, vuelve a competir por el `mutex`. Una vez adquirido de nuevo el `mutex`, debe comprobar si la situación que le despertó y que le permitía continuar su ejecución sigue cumpliéndose, de ahí la necesidad de emplear una estructura de control de tipo `while`. Es necesario volver a evaluar la condición, ya que entre el momento en el que la condición se hizo cierta y el instante en el que comienza a ejecutar de nuevo el proceso bloqueado en la variable condicional puede haber ejecutado otro proceso que, a su vez, puede haber hecho falsa la condición. En la Figura 4.7 se representa de forma gráfica el uso de `mutex` y variables condicionales entre dos procesos tal y como se ha descrito anteriormente.

A continuación se describen los servicios POSIX que permiten utilizar `mutex` y variables condicionales. Para utilizar un `mutex`, un programa debe declarar una variable de tipo `pthread_mutex_t` (definido en el archivo de cabecera `pthread.h`) e iniciarla antes de utilizarla.

Iniciar un `mutex`

Esta función permite iniciar una variable de tipo `mutex`. Su prototipo es el siguiente:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                      pthread_mutexattr_t *attr);
```

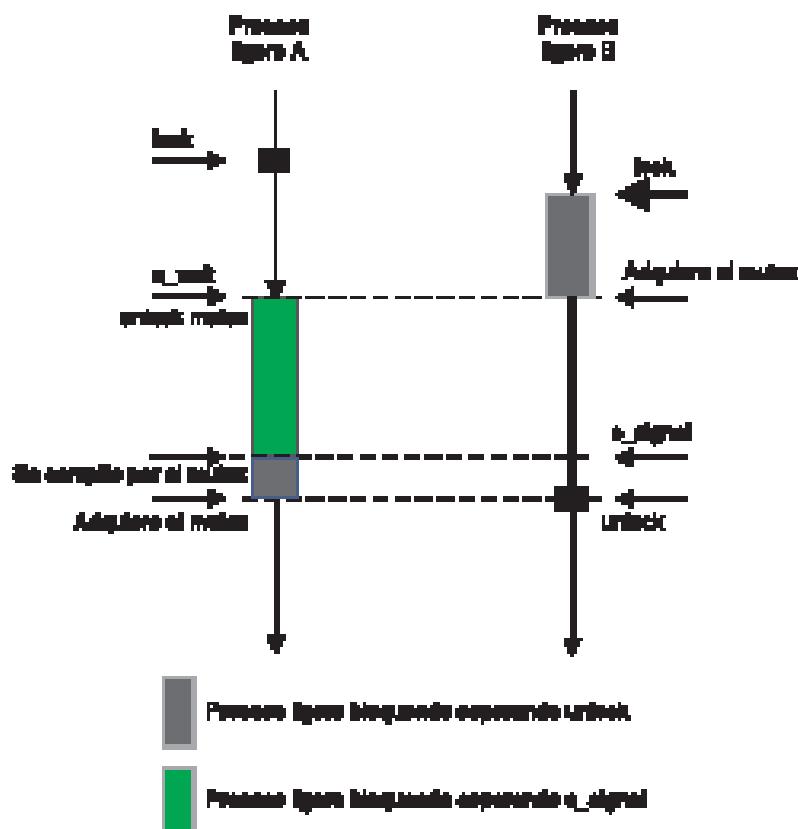


Figura 4.7. *Uso de mutex y variables condicionales.*

El segundo argumento especifica los atributos con los que se crea el *mutex* inicialmente; en caso de que este segundo argumento sea NULL, se tomarán los atributos por defecto.

Destruir un mutex

Permite destruir un objeto de tipo *mutex*. El prototipo de la función que permite invocar este servicio es:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Operación lock

Este servicio se corresponde con la operación *lock* descrita en la sección anterior. Esta función intenta obtener el *mutex*. Si el *mutex* ya se encuentra adquirido por otro proceso, el proceso ligero que ejecuta la llamada se bloquea. Su prototipo es:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Operación unlock

Este servicio se corresponde con la operación *unlock* y permite al proceso ligero que la ejecuta liberar el *mutex*. El prototipo es:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Iniciar una variable condicional

Para emplear en un programa una variable condicional es necesario declarar una variable de tipo *pthread_cond_t* e iniciarla antes de usarla mediante el servicio *pthread_cond_init*, cuyo prototipo se muestra a continuación:

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *attr);
```

Esta función inicia una variable de tipo condicional. El segundo argumento especifica los atributos con los que se crea inicialmente la variable condicional. Si el segundo argumento es NULL, la variable condicional toma los atributos por defecto.

Destruir una variable condicional

Permite destruir una variable de tipo condicional. Su prototipo es:

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Operación c_wait sobre una variable condicional

Este servicio se corresponde con la operación *c_wait* sobre una variable condicional. Su prototipo es:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

Esta función suspende al proceso ligero hasta que otro proceso ejecute una operación `c_signal` sobre la variable condicional pasada como primer argumento. De forma atómica, se libera el mutex pasado como segundo argumento. Cuando el proceso se despierte volverá a competir por el mutex.

Operación `c_signal` sobre una variable condicional

Este servicio se corresponde con la operación `c_signal` sobre una variable condicional. Su prototipo es:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Se desbloquea a un proceso suspendido en la variable condicional pasada como argumento a esta función. Esta función no tiene efecto si no hay ningún proceso ligero esperando sobre la variable condicional. Para desbloquear a todos los procesos ligeros suspendidos en una variable condicional se emplea el servicio:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

4.1.3. Interbloqueos

Un interbloqueo supone un bloqueo permanente de un conjunto de procesos que compiten por recursos o bien se comunican o sincronizan entre sí. Los interbloqueos que aparecen cuando se utilizan mecanismos de comunicación y sincronización se deben a un mal uso de los mismos. A continuación se van a presentar ejemplos de mala utilización de mecanismos de comunicación y sincronización que llevan a los procesos a un estado de interbloqueo.

Considérese en primer lugar una situación en la que se utilizan dos semáforos P y Q, ambos con un valor inicial 1. Si dos procesos utilizan estos dos semáforos de la siguiente manera se puede producir un interbloqueo:

Proceso A	Proceso B
<code>wait (P);</code>	<code>wait (Q);</code>
<code>wait (Q);</code>	<code>wait (P);</code>
<code>...</code>	<code>...</code>
<code>signal (P);</code>	<code>signal (Q);</code>
<code>signal (Q);</code>	<code>signal (P);</code>

Para modelizar los interbloqueos se suele recurrir a la construcción de un grafo de asignación de recursos. En este grafo existen dos tipos de nodos: los procesos se representan mediante cuadrados y los recursos mediante círculos (véase la Figura 4.8). Cuando un proceso solicita un recurso, se dibuja en el grafo un arco dirigido que parte del proceso y que va hacia el recurso, y cuando se concede un recurso a un proceso, el arco anterior se cambia por otro que parte del recurso y que va hacia el proceso. El interbloqueo se detecta cuando aparece un ciclo en el grafo de asignación de recursos.

La Figura 4.8 presenta el interbloqueo que se produce en el ejemplo anterior cuando los procesos ejecutan las operaciones `wait` según el siguiente orden:

Proceso A	Proceso B
1 <code>wait (P);</code>	2 <code>wait (Q);</code>
3 <code>wait (Q);</code>	4 <code>wait (P);</code>

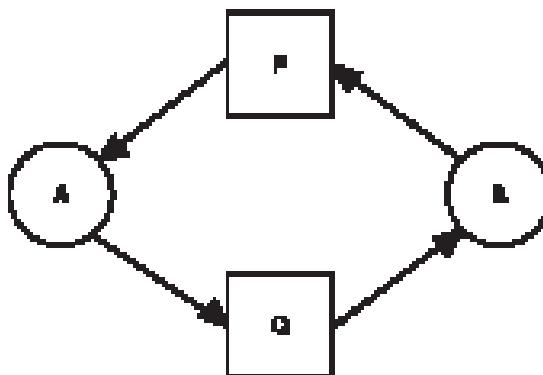


Figura 4.8. Ejemplo de interbloqueo utilizando semáforos.

4.2. PRÁCTICA: INTÉRPRETE DE MANDATOS CON TUBERÍAS

El objetivo de esta práctica es completar la funcionalidad de la Práctica 2.7, incluyendo la funcionalidad necesaria para la ejecución de secuencias de mandatos conectados a través de tuberías (*pipes*).

NIVEL: Avanzado.

HORAS ESTIMADAS: 16.

4.2.1. Descripción de la funcionalidad que debe desarrollar el alumno

La práctica a desarrollar tiene que completar la funcionalidad de la Práctica 2.7, cuyo objetivo era el desarrollo de un intérprete de mandatos sencillo. Para el desarrollo de esta práctica, debe utilizarse el mismo material de apoyo y la misma función `obtener_mandato`, descrita en esa práctica y que permite obtener el mandato o secuencia de mandatos a ejecutar. El alumno debe ampliar la Práctica 2.7 para permitir la ejecución de mandatos conectados a través de pípes. No hay ninguna restricción en el número de mandatos que se pueden ejecutar.

4.2.1.1. Ejecución de mandatos con tuberías

Para facilitar el desarrollo de esta práctica, a continuación se presenta un programa que permite ejecutar el mandato `ls | wc`. La ejecución de este mandato supone la ejecución de los programas `ls` y `wc` de UNIX y su conexión mediante una tubería. El código que permite la ejecución de este mandato es el que se muestra en el siguiente programa:

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

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

    /* se crea la tubería */
```

```

        if (pipe(fd) < 0) {
            perror("Error al crear la tubería");
            exit(0);
        }

        pid = fork();
        switch (pid) {
            case -1: /* error */
                perror("Error en el fork");
                exit(0);
            case 0: /* proceso hijo ejecuta ls */
                close(fd[0]);
                close(STDOUT_FILENO);
                dup(fd[1]);
                close(fd[1]);
                execvp("ls", "ls", NULL);
                perror("Error en el exec");
                break;
            default: /* proceso padre ejecuta wc */
                close(fd[1]);
                close(STDIN_FILENO);
                dup(fd[0]);
                close(fd[0]);
                execvp("wc", "wc", NULL);
                perror("Error en el exec");
        }
    }
}

```

El proceso hijo (véase Figura 4.9) redirige su salida estándar a la tubería. Por su parte, el proceso padre redirecciona su entrada estándar a la tubería. Con esto se consigue que el proceso que ejecuta el programa *ls* escriba sus datos de salida en la tubería y el proceso que ejecuta el programa *wc* lea sus datos de la tubería.

Los pasos que realiza el proceso hijo para redirigir su salida estándar a la tubería son los siguientes:

- Cierra el descriptor de lectura de la tubería, *fd[0]*, ya que no lo utiliza (*close(fd[0])*).
- Cierra la salida estándar, que inicialmente en un proceso referencia el terminal (*close(STDOUT_FILENO)*). Esta operación libera el descriptor de archivo 1, es decir, el descriptor *STDOUT_FILENO*.
- Duplica el descriptor de escritura de la tubería mediante la sentencia *dup(fd[1])*. Esta llamada devuelve y consume un descriptor, que será el de número más bajo disponible, en este caso el descriptor 1, que coincide en todos los procesos como el descriptor de salida estándar. Con esta operación se consigue que el descriptor de archivo 1 y el descriptor almacenado en *fd[1]* sirvan para escribir datos en la tubería. De esta forma se ha conseguido redirigir el descriptor de salida estándar en el proceso hijo a la tubería.
- Se cierra el descriptor *fd[1]*, ya que el proceso hijo no lo va a utilizar en adelante. Recuérdese que el descriptor 1 sigue siendo válido para escribir datos en la tubería.
- Cuando el proceso hijo invoca el servicio *exec* para ejecutar un nuevo programa, se conserva en el BCP la tabla de descriptores de archivos abiertos, y en este caso, el descriptor de salida estándar 1 está referenciando a la tubería. Cuando el proceso comienza a ejecutar el código del programa *ls*, todas las escrituras que se hagan sobre el descriptor de salida estándar se harán realmente sobre la tubería.

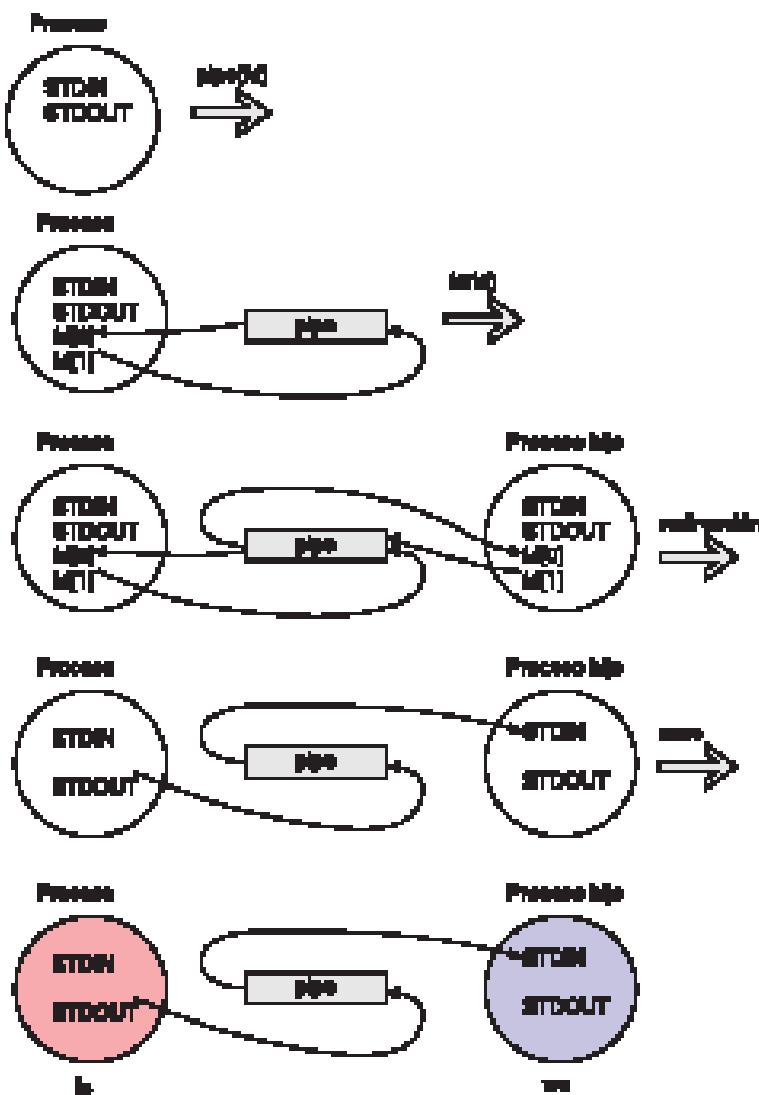


Figura 4.9. Ejecución de mandatos con tuberías.

El servicio `dup` de POSIX duplica un descriptor de archivo abierto. Su prototipo es:

```
int dup(int fd);
```

El servicio `dup` duplica el descriptor de archivo `fd`. La llamada devuelve el nuevo descriptor de archivo. Este descriptor de archivo referencia al mismo archivo al que referencia `fd`.

4.2.2. Desarrollo de la práctica

El desarrollo de esta práctica tiene que hacerse a partir de la solución realizada para la Práctica 2.7. Es decir, el alumno completará sobre la solución de la Práctica 2.7 la funcionalidad descrita en la sección anterior.

4.2.3. Entrega de documentación

El alumno deberá entregar los siguientes archivos:

- memoria.txt: Memoria de la práctica.
- main.c: Código fuente del ~~minishell~~, implementando todas las funcionalidades que se requieren en la Práctica 2.7 y en ésta.

4.2.4. Bibliografía

- F. García, J. Carretero, J. Fernández y A. Calderón. *Lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.
- J. Carretero, F. García, P. de Miguel y F. Costoya. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- K. A. Robbins y S. Robbins. *UNIX Programación Práctica*. Prentice-Hall, 1997.
- M. J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, 1985.

4.3. PRÁCTICA: PRODUCTOR-CONSUMIDOR UTILIZANDO PROCESOS LIGEROS

4.3.1. Objetivo de la práctica

El objetivo de esta práctica es que el alumno se familiarice con el uso de procesos ligeros, así como la forma de comunicar esos procesos mediante memoria compartida. También se pretende que el alumno aprenda a resolver el problema del productor consumidor en un caso concreto.

NIVEL: Avanzado.

HORAS ESTIMADAS: 10.

4.3.2. Descripción de la funcionalidad que debe desarrollar el alumno

La práctica consiste en modificar un código fuente en C que implemente un problema de tipo productor-consumidor. La descripción de este código es la siguiente:

- La función que implementa el consumidor genera un número aleatorio cada cierto tiempo e introduce este número en una cola circular. En caso de que esta cola estuviera llena, el consumidor esperaría a que exista una posición libre dentro de la cola circular.
- La función que implementa el productor examina la cola circular periódicamente: si ésta está vacía, espera a que haya algún elemento nuevo en ella; en caso contrario, extrae el elemento más antiguo de la cola circular, y pasado un cierto tiempo, que se considera «tiempo de proceso», lo imprime por pantalla.
- La cola circular se implementa mediante una estructura que contiene un vector de tamaño N donde se almacenan los elementos de la cola; asimismo, consta de dos enteros, `inicio_cola` y `fin_cola`, que almacenan índices del vector de elementos. El índice `inicio_cola` indica la posición del vector donde almacenar el próximo elemento a introducir en la cola circular. El índice `fin_cola` indica la posición de vector donde se encuentra el próximo elemento a extraer de la cola circular. Cuando alguno de los dos

índices alcance el valor máximo del vector, el siguiente elemento al que deberá transitar será el primer elemento del vector. De esta forma se consigue que la cola sea circular. Cuando los índices `inicio_cola` y `fin_cola` apuntan al mismo elemento del vector indican que la cola circular está vacía. En cambio, cuando el índice `fin_cola` es el siguiente elemento en la cola al apuntado por el índice `inicio_cola` indica que la cola circular está llena. (Recuerde que el elemento de la cola siguiente al que está en la posición N es el que está en la posición 1.)

El alumno deberá realizar las pertinentes modificaciones sobre el código que se proporciona para que éste pueda realizar las siguientes acciones:

- El programa deberá lanzar dos procesos ligeros, uno que ejecute la función `consumidor` y otro que ejecute la función `productor`. Para ello se debe usar la función `pthread_create`.
- El programa deberá garantizar la exclusión mutua en el acceso a la cola circular que comparten los dos procesos, es decir, hay que evitar que, en ningún caso, los dos procesos accedan a la vez a la cola circular. Para realizarlo se recomienda el uso de `mutex`.
- El programa deberá, asimismo, evitar la espera activa que realizan los dos procesos mientras comprueban si la cola circular está vacía o llena según el caso. Es decir, el proceso deberá esperar a que se cumpla la condición requerida de forma que deje la CPU libre para otros procesos. Para ello se recomienda el uso de variables de condición.

4.3.2.1. Código fuente de apoyo

Para facilitar la realización de la práctica se dispone del archivo `practica_4.3.tgz`, disponible en la página web del libro, que contiene código fuente de apoyo. Al extraer su contenido se crea el directorio `practica_4.3`, donde se debe desarrollar la práctica. Dentro de este directorio se habrán incluido los siguientes archivos:

- `Makefile`. Archivo fuente para la herramienta `make`. No debe ser modificado. Con él se consigue la recopilación automática de los ficheros fuente cuando se modifique. Basta con ejecutar el mandato `make` para que el programa se compile de forma automática.
- `procsyne.c`. Archivo de C a modificar donde se incluirá el código del productor, consumidor y la cola circular.

4.3.2.2. Recomendaciones generales

Es importante estudiar previamente el funcionamiento de los procesos ligeros en POSIX. Para evitar problemas es recomendable ejecutar las modificaciones en el orden en el cual se han descrito anteriormente; además, el programa no funcionará de forma adecuada al menos hasta haber completado los dos primeros pasos, siendo el tercer paso vital para evitar un uso desmesurado de la CPU por parte del programa.

El código que se suministra puede ejecutarse tal cual, pero no realiza ninguna tarea hasta que no se hayan realizado las oportunas modificaciones.

A continuación se listan las funciones que se pueden necesitar a la hora de desarrollar la práctica:

```
pthread_create
pthread_mutex_init pthread_mutex_lock pthread_mutex_unlock
pthread_cond_init  pthread_cond_wait  pthread_cond_signal
```

Para poder usar estas funciones es necesario incluir el archivo de cabecera del sistema `<pthread.h>` y enlazar el programa con la biblioteca `libpthread`.

4.3.3. Entrega de documentación

El alumno debe entregar los siguientes archivos:

- memoria.txt: Memoria de la práctica.
- procsyne.c: Archivo que contiene el programa.

4.3.4. Bibliografía

- F. García, J. Carretero, J. Fernández y A. Calderón. *Lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.
- J. Carretero, F. García, P. de Miguel y F. Costoya. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- K. A. Robbins y S. Robbins. *UNIX Programación Práctica*. Prentice-Hall, 1997.
- M. J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, 1985.
- K. A. Robbins y S. Robbins. *UNIX programación práctica. Guía para la concurrencia, la comunicación y los multihilos*. Prentice-Hall, 1997.

4.4. DISEÑO E IMPLEMENTACIÓN DE SEMÁFOROS EN EL MINIKERNEL

4.4.1. Objetivo de la práctica

El principal objetivo es que el alumno pueda conocer de forma práctica cómo funcionan los semáforos. A la funcionalidad típica de los semáforos se le ha añadido una serie de características que hacen que el alumno tenga que enfrentarse con distintos patrones de sincronización entre procesos.

- NIVEL: Diseño.
- HORAS ESTIMADAS: 10.

4.4.2. Descripción de la funcionalidad que debe desarrollar el alumno

Se pretende ofrecer a las aplicaciones un servicio de sincronización basado en semáforos. Antes de pasar a describir la funcionalidad que van a tener estos semáforos, hay que aclarar que su semántica está ideada para permitir practicar con distintas situaciones que aparecen frecuentemente en la programación de un sistema operativo. Las principales características de estos semáforos son las siguientes:

- El número de semáforos disponibles es fijo (constante `NUM_SEM`).
- Cada semáforo tiene asociado un nombre que consiste en una cadena de caracteres con un tamaño máximo igual a `MAX_NOM_SEM` (incluyendo el carácter nulo de terminación de la cadena).

- Cuando se crea un semáforo, el proceso obtiene un descriptor (similar al descriptor de archivo de POSIX) que le permite acceder al mismo. Si ya existe un semáforo con ese nombre, se devuelve un error. En caso de que no exista colisión, se debe comprobar si se ha alcanzado el número máximo de semáforos en el sistema. Si esto ocurre, se debe bloquear al proceso hasta que se elimine algún semáforo. La operación que crea el semáforo también lo deja abierto para poder ser usado.
- Para poder usar un semáforo ya existente se debe abrir especificando su nombre. El proceso obtiene un descriptor asociado al mismo.
- Cada proceso tiene asociado un conjunto de descriptores asociados a los semáforos que está usando. El número de descriptores por proceso está limitado a NUM_SEM_PROC. Si al abrir o crear un semáforo no hay ningún descriptor libre, se devuelve un error.
- Las primitivas de uso del semáforo (signal y wait) tienen básicamente la semántica convencional. Ambas reciben como parámetro un descriptor de semáforo. La única característica un poco especial es que la primitiva signal incluye como argumento el número de unidades que se incrementa el semáforo. Por tanto, esta llamada puede causar el desbloqueo de varios procesos.
- Cuando un proceso no necesita usar un semáforo, lo cierra. El semáforo se eliminará realmente cuando no haya ningún proceso que lo utilice, o sea, no haya ningún descriptor asociado al semáforo. En el momento de la liberación real es cuando hay que comprobar si había algún proceso bloqueado esperando para crear un semáforo debido a que se había alcanzado el número máximo de semáforos en el sistema.
- Cuando un proceso termina, ya sea voluntaria o involuntariamente, el sistema operativo debe cerrar todos los semáforos que usaba el proceso.

La interfaz de los servicios de semáforos va a ser muy similar a la interfaz de semáforos POSIX descrita anteriormente y es la siguiente:

- `int crear_sem(char *nombre, unsigned int val_ini)`. Crea el semáforo con el nombre y valor inicial especificado. Devuelve un entero que representa un descriptor para acceder al semáforo. En caso de error, devuelve un número negativo.
- `int abrir_sem(char *nombre)`. Devuelve un descriptor asociado a un semáforo ya existente o un número negativo en caso de error.
- `int wait_sem(unsigned int semid)`. Realiza la típica labor asociada a esta primitiva. En caso de error, devuelve un número negativo.
- `int signal_sem(unsigned int semid, unsigned int valor)`. Esta función permite incrementar el contador del semáforo en el valor especificado. En caso de error, devuelve un número negativo.
- `int cerrar_sem(unsigned int semid)`. Cierra el semáforo especificado, devolviendo un número negativo en caso de error.

Notese que todas las primitivas devuelven un número negativo en caso de error. Si lo considera oportuno, el alumno puede codificar el tipo de error usando distintos valores negativos. Así, por ejemplo, en la llamada `abrir_sem` pueden producirse, al menos, dos tipos de error: que no haya ningún descriptor libre y que no exista el semáforo.

Por último, hay que resaltar que el diseño de los semáforos debe tratar adecuadamente una situación como la que se especifica a continuación:

- El proceso P_1 está bloqueado en `crear_sem`, ya que se ha alcanzado el número máximo de semáforos. En la cola de procesos listos hay dos procesos (P_2 y P_3).

- P_2 realiza una llamada a `cerrar_sem`, que desbloquea a P_1 , que pasa al final de la cola de procesos listos.
- P_2 termina y pasa a ejecutar el siguiente proceso P_3 .
- P_3 llama a `crear_sem`: ¿qué ocurre?

Hay que asegurarse de que sólo uno de los dos procesos ($P_1 \circ P_2$) puede crear el semáforo, mientras que el otro se deberá quedar bloqueado. Se admiten como correctas las dos posibilidades. Se proporciona una pista al alumno: una forma de implementar la alternativa en la que P_1 se queda bloqueado y P_3 puede crear el semáforo es usar un bucle en vez de una sentencia condicional a la hora de bloquearse en `crear_sem` si no hay un semáforo libre.

4.4.3. Entrega de documentación

Consulte los apartados 1.3.3, 1.3.4, 1.3.5 y 1.3.6, para obtener información sobre el código fuente de apoyo y la documentación a entregar.

4.4.4. Bibliografía

- F. García, J. Carretero, J. Fernández y A. Calderón. *Lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.
- J. Carretero, F. García, P. de Miguel y F. Costoya. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- K. A. Robbins y S. Robbins. *UNIX programación práctica. Guía para la concurrencia, la comunicación y los multihilos*. Prentice-Hall, 1997.
- D. P. Bovet y M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.

4.5. DISEÑO E IMPLEMENTACIÓN DE MUTEX EN EL MINIKERNEL

4.5.1. Objetivo de la práctica

El objetivo principal de esta práctica es que el alumno pueda ver de forma práctica la diferencia que hay entre los semáforos y los mutex.

- NIVEL: Diseño.
- HORAS ESTIMADAS: 6.

4.5.2. Descripción de la funcionalidad que debe desarrollar el alumno

Se pretende completar la práctica anterior añadiendo un mecanismo de `mutex` que conviva estrechamente con los semáforos planteados en esa práctica previa. De hecho, el `mutex` se va a considerar un tipo especial de semáforo que va a compartir gran parte de la funcionalidad y de las estructuras de datos de los semáforos generales implementados previamente. Concretamente, se van a implementar dos tipos de `mutex`:

- Mutex no recursivos: Si un proceso que posee un `mutex` intenta bloquearlo de nuevo, se le devuelve un error, ya que se está produciendo un caso trivial de interbloqueo.

- **Mutex recursivos:** Un proceso que posee un *mutex* puede bloquearlo nuevamente todas las veces que quiera. Sin embargo, el *mutex* sólo quedará liberado cuando el proceso lo desbloquee tantas veces como lo bloqueó.

Se debería generalizar la estructura que contiene los datos de un semáforo definida en la práctica anterior para que pueda dar cobijo tanto a los datos de un semáforo de tipo general como a los de los dos tipos de *mutex*. Para ello se podría usar un registro variante como el proporcionado por un tipo *unión* de C. A continuación se presentan, de forma comparativa con los semáforos, las principales características de este mecanismo de sincronización:

- En el número de semáforos disponibles (*NUM_SEM*) y en el número de descriptores por proceso (*NUM_SEM_PROC*) se van a incluir todos los mecanismos de sincronización.
- El nombre del *mutex* tendrá las mismas características que el del semáforo. De hecho, compartirán el mismo espacio de nombres, o sea, que un semáforo y un *mutex* no pueden tener el mismo nombre.
- La creación del *mutex* (función *crear_mutex*) tendrá las mismas características que la de un semáforo. Debe comprobar la posible colisión de nombres (tanto con un semáforo como con otro *mutex*) y si se ha alcanzado el número máximo de semáforos (incluyendo tanto semáforos como *mutex*) en el sistema, en cuyo caso bloques al proceso. La operación de creación iniciará el estado del *mutex* como abierto y devolverá un descriptor que permita usarlo. Se recibirá como parámetro de qué tipo es el *mutex* (recursivo o no).
- La primitiva para abrir un *mutex* va tener también las mismas características que la de los semáforos.
- Las primitivas *lock* y *unlock* tienen el comportamiento convencional explicado en el Apartado 4.1.2.3.
- La semántica asociada al cierre de un *mutex* tiene una importante diferencia con respecto a los semáforos. En este caso, si el proceso que cierra el *mutex* lo tiene bloqueado, habrá que desbloquearlo implícitamente.
- Por lo demás, la semántica asociada al cierre es muy similar a la de un semáforo. El *mutex* se eliminará realmente cuando no haya ningún descriptor asociado al mismo. En el momento de la liberación real, se comprueba si había algún proceso bloqueando esperando para crear un semáforo o un *mutex*.
- Cuando un proceso termina, ya sea voluntaria o involuntariamente, el sistema operativo debe cerrar tanto los semáforos como los *mutex* que usaba el proceso.

La interfaz de los servicios de *mutex* va a ser la siguiente:

- `int crear_mutex(char *nombre, int tipo)`. Crea el *mutex* con el nombre y tipo especificados. Devuelve un entero que representa un descriptor para acceder al *mutex*. En caso de error, devuelve un número negativo. Habrá que definir dos constantes, que deberían incluirse tanto en el archivo de cabecera usado por los programas de usuario (*servicios.h*) como en el usado por el sistema operativo (*kernel.h*), para facilitar la especificación del tipo de *mutex*:

```
#define NO_RECURSIVO 0
#define RECURSIVO 1
```

- `int abrir_mutex(char *nombre)`. Devuelve un descriptor asociado a un *mutex* ya existente o un número negativo en caso de error.
- `int lock(unsigned int mutexid)`. Realiza la típica labor asociada a esta primitiva. En caso de error, devuelve un número negativo.

- `int unlock(unsigned int mutexid)`. Realiza la típica labor asociada a esta primitiva. En caso de error, devuelve un número negativo.
- `int cerrar_mutex(unsigned int mutexid)`. Cierra el mutex especificado, devolviendo un número negativo en caso de error.

Notese que todas las primitivas devuelven un número negativo en caso de error. El alumno debería estudiar exhaustivamente los distintos tipos de error que pueden darse en cada primitiva. Por último, hay que resaltar dos aspectos:

- Dado que tanto los semáforos de tipo general como ambos tipos de mutex comparten mucha funcionalidad, el alumno debería intentar organizar su código de manera adecuada para evitar duplicidades.
- Hay que llamar la atención sobre la diferencia que existe entre los semáforos y los mutex a la hora de cerrarlos. En el caso de los mutex, si el mutex que se pretende cerrar está bloqueando por el proceso, hay que desbloquearlo, ya que, en caso contrario, se quedaría bloqueando indefinidamente. Esto no ocurre así con los semáforos. Para entenderlo, supóngase un proceso que hace una operación `wait` sobre un semáforo que impide el paso al resto y que a continuación lo cierra. Ese semáforo no queda definitivamente «fuera de servicio», ya que otro proceso puede hacer una operación `signal` sobre el mismo. La clave está en que los mutex tienen asociado el concepto de «dueño», aquel proceso que lo tiene bloqueando, mientras que los semáforos no. Esta diferencia es trascendental a la hora de analizar el problema del interbloqueo, como se podrá apreciar en la siguiente práctica.

4.5.3. Entrega de documentación

Consulte las Secciones 1.3.3, 1.3.4, 1.3.5 y 1.3.6 para obtener información sobre el código fuente de apoyo y la documentación a entregar.

4.5.4. Bibliografía

- F. García, J. Carretero, J. Fernández y A. Calderón. *Lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.
- J. Carretero, F. García, P. de Miguel y F. Costoya. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- K. A. Robbins y S. Robbins. *UNIX programación práctica. Guía para la concurrencia, la comunicación y los multihilos*. Prentice-Hall, 1997.
- D. P. Bovet y M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.

4.6. DETECCIÓN DE INTERBLOQUEOS EN LOS MUTEX DEL MINIKERNEL

4.6.1. Objetivo de la práctica

El objetivo principal es intentar llevar a la práctica uno de los temas que tradicionalmente se han considerado como más teóricos dentro de los sistemas operativos: los interbloqueos.

- **NIVEL:** Diseño.
- **HORAS ESTIMADAS:** 10.

4.6.2. Interbloqueos para mutex frente a interbloqueos para semáforos

En la práctica se pretende implementar un algoritmo de detección de interbloqueos para los mutex desarrollados en la práctica anterior. En primer lugar, recordando el comentario final vertido en el enunciado de la misma, hay que resaltar la diferencia que existe entre estudiar los interbloqueos para los mutex frente a hacerlo para los semáforos.

Con respecto al mutex, una vez creado, los procesos lo usan de forma exclusiva hasta que se destruye. Con este tipo de primitiva pueden producirse situaciones de interbloqueo como la que aparece en el siguiente ejemplo:

Proceso P ₁	Proceso P ₂
lock (M ₁)	lock (M ₂)
lock (M ₁)	lock (M ₂)
.....
unlock (M ₁)	unlock (M ₂)
unlock (M ₁)	unlock (M ₂)

Puede darse un interbloqueo si los dos procesos consiguen bloquear el primer mutex y se quedan esperando por el segundo. Nótese que este interbloqueo se produciría con independencia de lo que puedan hacer otros procesos existentes en el sistema.

En cuanto a los semáforos generales, su patrón de comportamiento es muy diferente. Así, considérese qué ocurriría si en el ejemplo anterior se sustituyesen los mutex por semáforos iniciados con un contador igual a 1:

Proceso P ₁	Proceso P ₂
wait (S ₁)	wait (S ₂)
wait (S ₁)	wait (S ₂)
.....
signal (S ₁)	signal (S ₂)
signal (S ₁)	signal (S ₂)

Aparentemente, los procesos mantienen el mismo comportamiento. Sin embargo, hay un aspecto sutil muy importante. En este caso, el comportamiento puede depender de otros procesos externos que podrían incluir llamadas a signal sobre cualquiera de los dos semáforos. Por tanto, el posible interbloqueo detectado en el ejemplo anterior podría no darse en este caso siempre que algún proceso externo lo rompiera.

Por todo ello, la práctica se centra en la detección de interbloqueos sólo en mutex.

4.6.3. Descripción de la funcionalidad que debe desarrollar el alumno

En los mutex implementados en la práctica anterior no se controlaba la posibilidad de que se produzca interbloqueo, con la excepción del caso trivial de un proceso bloqueándose a sí mismo.

Los algoritmos de detección de interbloqueo se basan en mantener un grafo de asignación de recursos que refleja qué recursos tiene asignado cada proceso y cuáles tiene pedidos, pero no se le han concedido porque están asignados a otro proceso. Si se detecta un ciclo en el grafo, implica que hay un interbloqueo y que, por tanto, hay que aplicar un algoritmo de recuperación de ese estado.

En el caso específico de los mutex, el grafo reflejaría por cada proceso qué mutex tiene bloqueados y por qué mutex está esperando, si es que lo está haciendo. *A priori*, puede parecer

que es necesario empezar a definir nuevas estructuras de datos que implementen este grafo. La buena noticia es que va a ser suficiente con las estructuras ya existentes, puesto que éstas ya reflejan estas relaciones entre los procesos y los mutex.

Por lo que se refiere al algoritmo de detección, se ejecutará cada vez que un proceso realiza una llamada `lock` y se encuentra que el mutex está bloqueado por otro proceso. La detección del ciclo en el grafo se puede llevar a cabo siguiendo las pautas que se explican a continuación:

- Un proceso P solicita un `lock` sobre M y el sistema operativo detecta que ya está bloqueado por otro proceso. Se arranca el algoritmo de detección de interbloqueos comenzando por el proceso P :
 1. Se obtiene qué `mutex` tiene bloqueados P .
 2. Por cada `mutex` se averigua la lista de procesos que están esperando por él mismo.
 3. Se repiten los dos pasos anteriores para cada uno de los procesos de esta lista, y así sucesivamente.
- Este proceso repetitivo se aplica hasta que:
 - Aparece de nuevo el `mutex` M en el paso 1: hay un interbloqueo.
 - Se termina de recorrer todo el grafo alcanzable desde P sin aparecer M : no hay interbloqueo.

Se recomienda solicitar al alumno que calcule el grado de complejidad del algoritmo que desarrolla con respecto al número de procesos y `mutex` existentes en el sistema.

Con respecto a la estrategia de recuperación que se aplicará una vez detectado el interbloqueo, se pueden plantear dos alternativas que dan lugar a dos versiones de la práctica:

- Una menos drástica, que consiste simplemente en devolver un valor de error en la primitiva `lock` correspondiente. Nótese que, en este caso, el resto de los procesos implicados en el interbloqueo no van a poder continuar hasta que este proceso desbloquee alguno de los `mutex`. Retomando el ejemplo anterior:

Proceso P_1	Proceso P_2
<code>lock (M₁)</code>	<code>lock (M₂)</code>
<code>lock (M₂)</code>	<code>lock (M₁)</code>
.....
<code>unlock (M₂)</code>	<code>unlock (M₁)</code>
<code>unlock (M₁)</code>	<code>unlock (M₂)</code>

Supóngase la siguiente situación:

- P_1 bloquea M_1
- P_2 bloquea M_2
- P_1 intenta bloquear M_2 pero, como lo tiene otro proceso, se aplica el algoritmo de detección que comprueba que no hay ningún ciclo.
- P_2 intenta bloquear M_1 pero, como lo tiene otro proceso, se aplica el algoritmo de detección que encuentra un ciclo. Se devuelve un error en esa llamada `lock`. Nótese que, de todas formas, P_1 seguirá esperando hasta que P_2 desbloquee el `mutex` M_2 . En el peor de los casos, eso ocurrirá cuando termine P_2 , aunque lo más lógico es

que el programador controle si la llamada `lock` devuelve un error y, si es así, termine el proceso.

```
if [lock(M1)<0] {
    printf ("Error bloquando M1\n");
    terminar_proceso();
}
.
.
.
```

- Una más agresiva, en la que se va a «matar» directamente al proceso que ha invocado la llamada `lock`. Con esta estrategia se liberan inmediatamente los `mutex` que tenía bloqueados el proceso, quedando disponibles para el resto de los procesos. Así, en el ejemplo, se abortaría la ejecución de P_2 y P_1 pasaría inmediatamente al estado de listo para ejecutar.

4.6.4. Entrega de documentación

Consulte los Apartados 1.3.3, 1.3.4, 1.3.5 y 1.3.6, para obtener información sobre el código fuente de apoyo y la documentación a entregar.

4.6.5. Bibliografía

- F. García, J. Carretero, J. Fernández y A. Calderón. *Lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.
- J. Carretero, F. García, P. de Miguel y F. Costoya. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- K. A. Robbins y S. Robbins. *UNIX programación práctica. Guía para la concurrencia, la comunicación y las multihilos*. Prentice-Hall, 1997.
- D. R. Bovet y M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.

5

Entrada/salida, archivos y directorios

En este capítulo se presentan las prácticas relacionadas con entrada/salida, archivos y directorios. El capítulo tiene tres objetivos básicos: mostrar al lector algunos conceptos básicos desde el punto de vista de usuario, mostrar los servicios que da el sistema operativo y proponer un conjunto de prácticas que permita cubrir los aspectos básicos y de diseño de los sistemas de entrada/salida, archivos, directorios y del servidor de archivos. De esta forma, se pueden adaptar las prácticas del tema a distintos niveles de conocimiento.

5.1. CONCEPTOS BÁSICOS DE ENTRADA/SALIDA

El sistema de E/S es la parte del sistema operativo que se ocupa de facilitar el manejo de los dispositivos de E/S ofreciendo una visión lógica simplificada de los mismos que pueda ser usada por otros componentes del sistema operativo (como el sistema de archivos) o incluso por el usuario. Mediante esta visión lógica, se ofrece a los usuarios un mecanismo de abstracción que oculta todos los detalles relacionados con los dispositivos físicos, así como del funcionamiento real de los mismos. El sistema operativo debe controlar el funcionamiento de todos los dispositivos de E/S para alcanzar los siguientes objetivos:

- Facilitar el manejo de los dispositivos periféricos. Para ello debe ofrecer una interfaz entre los dispositivos y el resto del sistema que sea sencilla y fácil de utilizar.
- Optimizar la E/S del sistema, proporcionando mecanismos de incremento de prestaciones donde sea necesario.
- Proporcionar dispositivos virtuales que permitan conectar cualquier tipo de dispositivo físico sin que sea necesario remodelar el sistema de E/S del sistema operativo.
- Permitir la conexión de dispositivos nuevos de E/S, solventando de forma automática su instalación usando mecanismos del tipo *plug & play*.

En el modelo general de conexión de periféricos a una computadora se distinguen dos elementos:

- **Periféricos o dispositivos de E/S.** Elementos que se conectan a la unidad central de proceso a través de las unidades de entrada/salida. Son el componente mecánico que se conecta a la computadora.
- **Controladores de dispositivos o unidades de E/S.** Se encargan de hacer la transferencia de información entre la memoria principal y los periféricos. Son el componente electrónico a través del cual se conecta el dispositivo de E/S. Tienen una conexión al bus de la computadora y otra para el dispositivo (generalmente mediante cables internos o externos).

El controlador es el componente más importante desde el punto de vista del sistema operativo, ya que constituye la interfaz del dispositivo con el bus de la computadora y es el componente que se ve desde la UCP. Su programación se lleva a cabo mediante una interfaz de muy bajo nivel que proporciona acceso a una serie de registros del controlador, incluidos en el mapa de E/S de la computadora, que se pueden acceder mediante instrucciones de máquina de E/S. Hay tres registros importantes en casi todos los controladores: **registro de datos, estado y control**. El registro de datos sirve para el intercambio de datos. En él irá el controlador cargando los datos leídos y de él irá extrayendo los datos para su escritura en el periférico. Un bit del registro de estado sirve para indicar que el controlador puede transferir una palabra. En las operaciones de lectura esto significa que ha cargado en el registro de datos un nuevo valor, mientras que en las de escritura significa que necesita un nuevo dato. Otros bits de este registro sirven para que el controlador indique los problemas que ha encontrado en la ejecución de la última operación de E/S. El registro de control sirve para indicarle al controlador las operaciones que ha de realizar. Los distintos bits de este registro indican distintas acciones que ha de realizar el periférico. Para empezar una operación de E/S, la UCP tiene que escribir sobre los registros anteriores los datos de la operación a través de una dirección de E/S o de memoria asignada únicamente al controlador. Este modelo vale tanto para los terminales o la pantalla como para los discos.

Las características del controlador son muy importantes, ya que definen el aspecto del periférico para el sistema operativo. Atendiendo a las características del *hardware* de los dispositivos, se pueden observar los siguientes aspectos distintivos:

- **Dirección de E/S.** En general, hay dos modelos de direccionamiento de E/S: los que usan puertos y los que proyectan los registros en memoria.
- **Unidad de transferencia.** Los dispositivos suelen usar unidades de transferencia de tamaño fijo. Hay dos modelos clásicos de dispositivos: de caracteres y de bloques.
- **Interacción computadora-controlador.** La computadora tiene que interaccionar con la controladora para realizar las operaciones de E/S y saber cuándo terminan.

La forma más habitual de comunicación entre la UCP y los controladores es usar interrupciones. Empleando **E/S dirigida por interrupciones**, el procesador envía la orden de E/S al controlador de dispositivo y no espera a que éste se encuentre listo para enviar o transmitir los datos, sino que se dedica a otras tareas hasta que llega una interrupción del dispositivo que indica que se ha realizado la operación solicitada.

El modelo de interrupciones está íntimamente ligado a la arquitectura del procesador. Casi todas las UCP actuales incluyen interrupciones vectorizadas y enmascarables. Es decir, un rango de interrupciones entre 0 y 255, por ejemplo, alguna de las cuales se pueden inhibir temporalmente para no recibir interrupciones de su vector correspondiente. Cada interrupción se asigna a un dispositivo, o un rango de ellos en caso de un controlador SCSI o una cadena de dispositivos tipo *daisy chain*, que usa el vector correspondiente para indicar eventos de E/S a la UCP. Cuando se programa una operación en un dispositivo, como, por ejemplo, una búsqueda en un disco, éste contesta con un ACK indicando que la ha recibido, lo que no significa que haya terminado. En este caso, existe concurrencia entre la E/S y el procesador, puesto que éste se puede dedicar a ejecutar código de otro proceso optimizando de esta forma el uso del procesador. Al cabo de un cierto tiempo, cuando el disco ha efectuado la búsqueda y las cabezas del disco están sobre la posición deseada, levanta una interrupción (poniendo un 1 en el vector correspondiente). La rutina de tratamiento de la interrupción se encargará de leer o enviar el dato al controlador. Obsérvese que tanto la tabla de interrupciones como la rutina de tratamiento de la interrupción se consideran parte del sistema operativo. Esto suele ser así por razones de seguridad; en concreto, para evitar que los programas que ejecuta un usuario puedan perjudicar a los datos o programas de otros usuarios.

Las computadoras incluyen varias señales de solicitud de interrupción, cada una de las cuales tiene una determinada prioridad. En caso de activarse al mismo tiempo varias de estas señales, se tratará la de mayor prioridad, quedando las demás a la espera de ser atendidas. Además, la computadora incluye un mecanismo de **inhibición** selectiva que permite detener todas o determinadas señales de interrupción. Las señales inhibidas no son atendidas hasta que pasen a estar desinhibidas. La información de inhibición de las interrupciones suele incluirse en la parte del registro de estado, que solamente es modificable en nivel de núcleo, por lo que su modificación queda restringida al SO.

5.2. MANEJADORES DE DISPOSITIVOS: EL RELOJ Y EL TERMINAL

En este apartado se presentan de forma breve cuáles son las principales funciones de los manejadores de dos de los dispositivos más característicos de una computadora: el reloj y el terminal.

5.2.1. El manejador del reloj

La labor principal del manejador del reloj es el tratamiento de sus interrupciones. Asimismo, también se encarga de realizar su iniciación y llevar a cabo las llamadas al sistema relacionadas

con el mismo. Con independencia de cuál sea el sistema operativo específico, se pueden identificar las siguientes operaciones como las funciones principales del manejador del reloj: mantenimiento de la fecha y de la hora, gestión de temporizadores, obtención de contabilidad y estadísticas y soporte para la planificación de procesos.

Mantenimiento de la fecha y de la hora

En el arranque del equipo, el sistema operativo debe leer el reloj, mantenido por una batería, para obtener la fecha y hora actual. A partir de ese momento, el sistema operativo se encargará de actualizar la hora según se vayan produciendo las interrupciones.

La principal cuestión referente a este tema es cómo se almacena internamente la información de la fecha y la hora. En la mayoría de los sistemas, la fecha se representa como el número de unidades de tiempo transcurridas desde una determinada fecha en el pasado. Sea cuál sea la información almacenada para mantener la fecha y la hora, es muy importante que se le dedique un espacio de almacenamiento suficiente para que se puedan representar fechas en un futuro a medio o incluso a largo plazo.

El servicio POSIX para obtener la fecha y hora es `time`, cuyo prototipo es el siguiente:

```
time_t time (time_t *t);
```

Esta función devuelve el número de segundos transcurridos desde el 1 de enero de 1970 en UTC. Si el argumento no es nulo, también lo almacena en el espacio apuntado por el mismo.

Gestión de temporizadores

En numerosas ocasiones un programa necesita esperar un cierto plazo de tiempo antes de realizar una determinada acción. El sistema operativo ofrece servicios que permiten a los programas establecer temporizaciones y se encarga de notificarlas cuando se cumple el plazo especificado.

Contabilidad y estadísticas

Puesto que la rutina de interrupción se ejecuta periódicamente, desde ella se puede realizar un muestreo de diversos aspectos del estado del sistema llevando a cabo funciones de contabilidad y estadística. Una de las funciones de este tipo presentes en la mayoría de los sistemas operativos es la contabilidad del uso del procesador por parte de cada proceso. En cada interrupción se detecta qué proceso está ejecutando y a éste se le carga el uso del procesador en ese intervalo. Generalmente, el sistema operativo distingue a la hora de realizar esta contabilidad si el proceso estaba ejecutando en modo usuario o en modo sistema.

El servicio `times` de POSIX devuelve información sobre el tiempo de ejecución de un proceso y de sus procesos hijos. El prototipo de la función es el siguiente:

```
clock_t times (struct tms *info);
```

Esta función rellena la zona apuntada por el puntero recibido como argumento con información sobre el uso del procesador, en modo usuario y en modo sistema, tanto del propio proceso como de sus procesos hijos. Además, devuelve un valor relacionado con el tiempo real en el sistema (típicamente, el número de interrupciones de reloj que se han producido desde el arranque del sistema).

Soporte a la planificación de procesos

La mayoría de los algoritmos de planificación de procesos tienen en cuenta de una forma u otra el tiempo y, por tanto, implican la ejecución de ciertas acciones de planificación dentro de la rutina de interrupción. En el caso de un algoritmo *round-robin*, en cada interrupción de reloj se le des cuenta el tiempo correspondiente a la rodaja asignada al proceso. Cuando se produce la interrupción de reloj que consume la rodaja, se realiza la replanificación.

5.2.2. El manejador del terminal

En este apartado se van analizar las labores principales de un manejador de terminal. Este dispositivo está formado típicamente por un teclado que permite introducir información y una pantalla que posibilita su visualización. Por tanto, se presentarán primero los aspectos relacionados con el *software* que maneja la entrada para, a continuación, estudiar el *software* que gestiona la salida.

Software de entrada

La lectura del terminal está dirigida por interrupciones. Cada vez que se produce una interrupción del teclado, el manejador debe recoger de su controlador información sobre la tecla pulsada.

Una característica que debe proporcionar todo manejador de terminal es el «teclado anticipado» (en inglés, *type ahead*). Esta característica permite que el usuario teclee información antes de que el programa la solicite, lo que proporciona al usuario una forma de trabajo mucho más cómoda. Para implementar este mecanismo, se requiere que el manejador use una zona de almacenamiento intermedio de entrada para guardar los caracteres tecleados hasta que los solicite un proceso.

Otro aspecto importante es la edición de los datos de entrada. Cuando un usuario teclea unos datos puede, evidentemente, equivocarse y se requiere, por tanto, un mecanismo que le permita corregir el error cometido. Surge entonces la cuestión de quién se debe encargar de proporcionar esta función de edición de los datos introducidos. Se presentan, al menos, dos alternativas:

- El manejador ofrece un modo de operación en el que proporciona unas funciones de edición relativamente sencillas, generalmente orientadas a líneas de texto individuales. Éste suele ser el modo de trabajo por defecto, puesto que satisface las necesidades de la mayoría de las aplicaciones.
- El manejador ofrece otro modo de operación en el que no se proporciona ninguna función de edición. La aplicación recibe directamente los caracteres tecleados y será la encargada de realizar las funciones de edición que considere oportunas.

Por lo que se refiere al modo orientado a línea, hay que resaltar que una solicitud de lectura de una aplicación no se puede satisfacer si el usuario no ha introducido una línea completa, aunque se hayan tecleado caracteres suficientes. Nótese que esto se debe a que, hasta que no se completa una línea, el usuario todavía tiene la posibilidad de editarla usando los caracteres de edición correspondientes.

En el modo orientado a línea existen unos caracteres que tienen asociadas funciones de edición, pero éstos no son los únicos caracteres que reciben un trato especial por parte del manejador. Existe un conjunto de caracteres especiales que normalmente no se le pasan al programa que lee del terminal, como ocurre con el resto de los caracteres, sino que activa alguna función del manejador. Estos caracteres se pueden clasificar en las siguientes categorías:

- *Carácteres de edición.* Tienen asociadas función de edición tales como borrar el último carácter tecleado, borrar la línea en curso o indicar el fin de la entrada de datos. Estos caracteres sólo se procesan si el terminal está en modo línea.
- *Carácteres para el control de procesos.* Todos los sistemas proporcionan al usuario algún carácter para abortar la ejecución de un proceso o detenerla temporalmente.
- *Carácteres de control de flujo.* El usuario puede desear detener momentáneamente la salida que genera un programa para poder revisarla y, posteriormente, dejar que continúe apareciendo en la pantalla. El manejador gestiona caracteres especiales que permiten realizar estas operaciones.
- *Carácteres de escape.* A veces, el usuario quiere introducir como entrada de datos un carácter que está definido como especial. Se necesita un mecanismo para indicar al manejador que no trate dicho carácter, sino que lo pase directamente a la aplicación. Para ello, generalmente, se define un carácter de escape cuya misión es indicar que el carácter que viene a continuación no debe procesarse.

Por último, resaltar que la mayoría de los sistemas ofrecen la posibilidad de cambiar qué carácter está asociado a cada una de estas funciones o incluso desactivar dichas funciones si se considera oportuno.

POSIX especifica dos funciones destinadas a obtener los atributos de un terminal y a modificarlos, respectivamente. Sus prototipos son los siguientes:

```
int tcgetattr (int descriptor, struct termios *atrib);
int tcsetattr (int descriptor, int opción, struct termios *atrib);
```

La función `tcgetattr` obtiene los atributos del terminal asociado al descriptor especificado. La función `tcsetattr` modifica los atributos del terminal que corresponde al descriptor pasado como parámetro.

Software de salida

La salida en un terminal no es algo totalmente independiente de la entrada. Por defecto, el manejador hace eco de todos los caracteres que va recibiendo en las sucesivas interrupciones del teclado. Así, la salida que aparece en el terminal es una mezcla de lo que escriben los programas y del eco de los datos introducidos por el usuario. Esta opción se puede desactivar, en cuyo caso el manejador no escribe en la pantalla los caracteres que va recibiendo.

A diferencia de la entrada, la salida no está orientada a líneas de texto, sino que se escriben directamente los caracteres que solicita el programa, aunque no constituyan una línea.

5.3. PRÁCTICA: MONITORIZACIÓN DE ENTRADA/SALIDA EN WINDOWS: PORTMON

5.3.1. Objetivos de la práctica

El objetivo de esta práctica es que el alumno se familiarice con una herramienta de monitorización de puertos de entrada/salida serie y paralela y que sea capaz de observar los eventos del sistema significativos para estos objetos. Para ello se usará la herramienta Portmon, un monitor muy

sencillo, pero eficiente, que se puede conseguir de forma gratuita, junto con mucho material de ayuda, en la página web del libro.

NIVEL: Introducción.

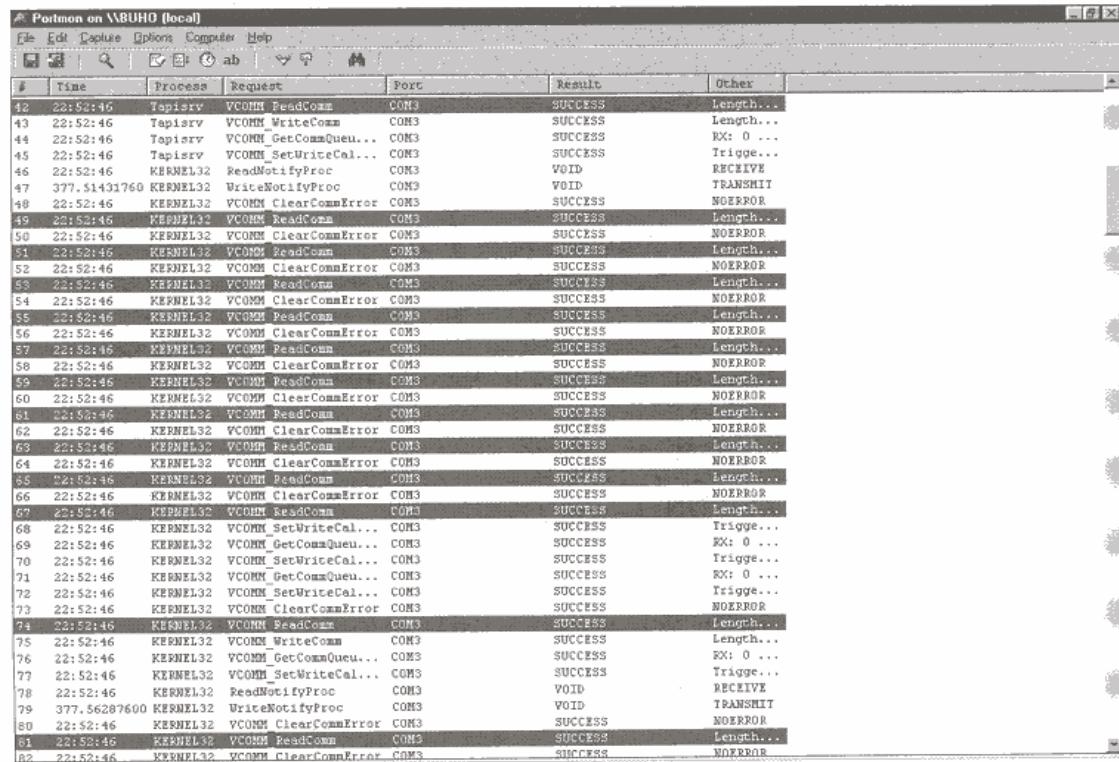
HORAS ESTIMADAS: 6.

5.3.2. Descripción de la funcionalidad que debe desarrollar el alumno

Portmon es un monitor de los puertos de entrada/salida serie y paralelos. Una vez ejecutado, registra todas las operaciones relacionadas con estos puertos. Por ejemplo, cada vez que se pulsa el ratón o el teclado, hay entradas por los puertos serie y Portmon captura todas esas acciones. El resultado es una pantalla como la de la Figura 5.1, en la que, como se puede apreciar, hay varias columnas: la primera columna (#) es el código de la operación (un número que se asigna a cada operación), la segunda columna (Time) es la hora y la tercera columna (Process) es el programa que accedió al recurso. En la Figura 5.1 se muestra la salida generada por el monitor cuando se establece una conexión de un módem a través de un puerto serie.

La práctica tiene dos partes:

1. **Instalación del monitor Portmon** en la máquina de prácticas. El alumno deberá ir a la página web del libro y traer la versión del monitor que corresponda con su sistema operativo (Windows 9x, Windows NT/2000). Nota: Para poder instalar la utilidad en Windows 2000, y para poder ejecutarla, se necesitan permisos de administrador.



The screenshot shows the Portmon application window titled "Portmon on \\\BUHU (local)". The window has a menu bar with File, Edit, Capture, Options, Computer, Help, and a toolbar with various icons. The main area is a table with columns: #, Time, Process, Request, Port, Result, and Other. The table lists numerous entries from 42 to 82, detailing operations like ReadComm, WriteComm, SetWriteCal, GetCommQueu, and ClearCommError on COM3, all resulting in SUCCESS with NOERROR. Some entries show additional details like Length, EK, or Trigge.

#	Time	Process	Request	Port	Result	Other
42	22:52:46	Tapisrv	VCOMM_ReadComm	COM3	SUCCESS	Length...
43	22:52:46	Tapisrv	VCOMM_WriteComm	COM3	SUCCESS	Length...
44	22:52:46	Tapisrv	VCOMM_SetWriteCal...	COM3	SUCCESS	EK: 0 ...
45	22:52:46	Tapisrv	VCOMM_SetWriteCal...	COM3	SUCCESS	Trigge...
46	22:52:46	KERNEL32	ReadNotifyProc	COM3	VOID	RECEIVE
47	377.51431760	KERNEL32	WriteNotifyProc	COM3	VOID	TRANSMIT
48	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
49	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
50	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
51	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
52	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
53	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
54	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
55	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
56	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
57	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
58	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
59	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
60	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
61	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
62	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
63	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
64	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
65	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
66	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
67	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
68	22:52:46	KERNEL32	VCOMM_SetWriteCal...	COM3	SUCCESS	Trigge...
69	22:52:46	KERNEL32	VCOMM_GetCommQueu...	COM3	SUCCESS	EK: 0 ...
70	22:52:46	KERNEL32	VCOMM_SetWriteCal...	COM3	SUCCESS	Trigge...
71	22:52:46	KERNEL32	VCOMM_GetCommQueu...	COM3	SUCCESS	EK: 0 ...
72	22:52:46	KERNEL32	VCOMM_SetWriteCal...	COM3	SUCCESS	Trigge...
73	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
74	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
75	22:52:46	KERNEL32	VCOMM_WriteComm	COM3	SUCCESS	Length...
76	22:52:46	KERNEL32	VCOMM_GetCommQueu...	COM3	SUCCESS	EK: 0 ...
77	22:52:46	KERNEL32	VCOMM_SetWriteCal...	COM3	SUCCESS	Trigge...
78	22:52:46	KERNEL32	ReadNotifyProc	COM3	VOID	RECEIVE
79	377.56287600	KERNEL32	WriteNotifyProc	COM3	VOID	TRANSMIT
80	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR
81	22:52:46	KERNEL32	VCOMM_ReadComm	COM3	SUCCESS	Length...
82	22:52:46	KERNEL32	VCOMM_ClearCommError	COM3	SUCCESS	NOERROR

Figura 5.1. Pantalla de salida de Portmon.

2. Activación del monitor Portmon durante cinco minutos, captura y elaboración de los datos de salida.

La primera parte de la práctica es muy sencilla, ya que la aplicación se descarga de la web del libro (*portmon.zip*) en un archivo comprimido tipo ZIP. Basta con extraer los archivos y ya están listos para funcionar. El monitor es un programa de apenas 90 KB, pero con una gran utilidad para aquellos usuarios interesados en el funcionamiento interno del sistema operativo Windows. Haga doble clic en el archivo *Portmon.exe* para ejecutar el programa.

La segunda parte tiene cuatro pasos:

- 2.1. **Control del número de operaciones** de entrada/salida que hacen determinadas aplicaciones al abrirse. Para ello, pulse el ratón y el teclado varias veces. En cada caso anote el número de operaciones antes y después y calcule las operaciones realizadas sobre los recursos de entrada/salida.
- 2.2. **Filtrado de operaciones** para estudiar sólo determinados tipos. Para ello, el alumno deberá filtrar las operaciones para que aparezcan sólo las operaciones *ReadComm* y *WriteComm*. Para filtrar operaciones, pulse *Edit -> Filter/Highlight* y seleccione *Highlight* de esas operaciones. Se verán rojas en la pantalla. El alumno deberá capturar una de estas pantallas con *PrintScreen* e incluirla en su memoria.
- 2.3. **Registro de la monitorización** en un archivo de texto *Portmon.log*. Ábralo como una hoja de cálculo de Excel y aplique un autofiltro sobre la columna *Request*. Basándose en el filtrado, el alumno deberá responder a las preguntas siguientes:
 - ¿Cuántas operaciones se han hecho en total?
 - Incluya en su memoria la lista de las operaciones ejecutadas durante la monitorización.
 - Realice un gráfico como el que se muestra en la Figura 5.2. Dicho gráfico muestra todas las operaciones realizadas y el número de las mismas. ¿Qué conclusiones se pueden sacar de este estudio?

5.3.3. Recomendaciones generales

Antes de empezar con la monitorización de todo el sistema de archivos, se recomienda monitorizar acciones sencillas. El ratón y el teclado pueden ser buenos ejemplos. De esta forma será capaz

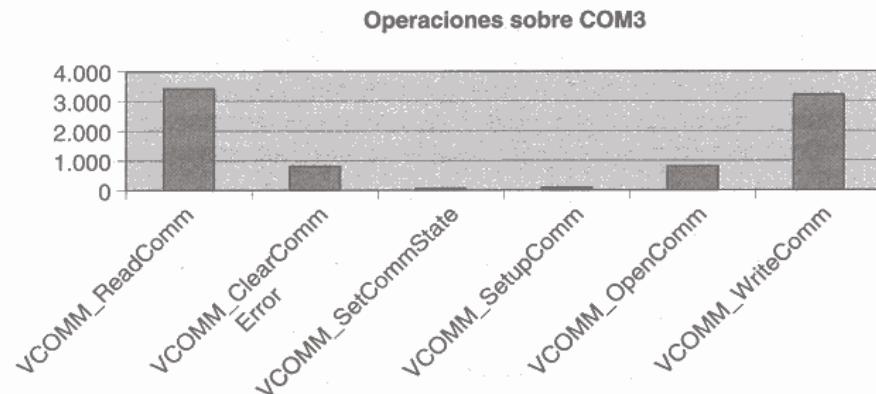


Figura 5.2. Número y tipo de operaciones de entrada/salida.

de comprender mejor el funcionamiento de Portmon y de realizar un mejor análisis de los resultados finales. El directorio se elige mediante la opción `Edit->Filter->Include`.

Si quiere observar el efecto de determinadas operaciones, como, por ejemplo, la impresión de un archivo a una impresora conectada a un puerto paralelo, envíe algo a dicho dispositivo. Otra buena prueba es conectar algo por el módem.

El monitor tiene una buena herramienta de ayuda en línea, úsela.

5.3.4. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- `Memoria.doc`: Memoria de la práctica.
- `Portmon.log`: Archivo que contiene el resultado de la sesión de monitorización.

5.3.5. Bibliografía

- D. Solomon y M. Russinovich. *Inside Windows 2000*. 3.^a edición, Microsoft Press, 2000.
- R. Nagar. *Windows NT File System Internals*. O'Reilly and Associates, 1997.

5.4. PRÁCTICA: IMPLEMENTACIÓN DE UN MANEJADOR DE RELOJ EN EL MINIKERNEL

5.4.1. Objetivo de la práctica

Esta práctica permite que el alumno pueda ver de forma aplicada algunas de las funciones asociadas con el reloj del sistema.

NIVEL: Diseño.

HORAS ESTIMADAS: 3.

5.4.2. Descripción de la funcionalidad que debe desarrollar el alumno

En el capítulo dedicado a los procesos ya se plantearon prácticas que implicaban incluir cierta funcionalidad asociada al reloj del sistema. Concretamente, se trataron dos de las funciones típicas proporcionadas por el manejador de reloj:

- La gestión de temporizadores, requerida por la llamada `dormir`.
- El soporte a la planificación de procesos, necesario tanto para el algoritmo *round-robin* como para el algoritmo de planificación similar al disponible en Linux.

En esta práctica se plantea completar la rutina añadiendo otras dos funcionalidades típicas:

- El mantenimiento de la fecha y la hora.
- La contabilidad del uso del procesador por parte de un proceso.

Con respecto al primer aspecto, como ya se comentó en el primer capítulo, el sistema operativo lee en su arranque la fecha y la hora del reloj CMOS del sistema, y a partir de ese momento, se encarga él mismo de actualizarla en cada interrupción de reloj, de acuerdo con la frecuencia de interrupción del mismo. La práctica plantea implementar la siguiente llamada al sistema:

```
int fechayhora(int *milisegundos);
```

Esta llamada devuelve como resultado el número de segundos transcurridos desde el 1 de enero de 1970. Además, si recibe como argumento un puntero que no sea nulo, almacena en el espacio apuntado por el mismo el número de milisegundos transcurridos dentro del último segundo.

Observe que para obtener la fecha y la hora en un formato más convencional (o sea, día, mes, año, horas, minutos y segundos), los programas de usuario, después de invocar esta llamada, pueden usar la función de biblioteca `localtime` para realizar la conversión.

Por lo que se refiere a la contabilidad, se va a implementar una función «inspirada» en la llamada `times` de POSIX, que tendrá el siguiente prototipo:

```
int tiempos_proceso(struct tiempos_ejec *t_ejec);
```

Siendo el tipo `struct tiempos_ejec`:

```
struct tiempos_ejec {
    int usuario;
    int sistema;
};
```

Esta llamada devuelve el número de interrupciones de reloj que se han producido desde que arrancó el sistema. Además, si recibe como argumento un puntero que no sea nulo, almacena en el espacio apuntado por el mismo cuántas veces en la interrupción de reloj se ha detectado que el proceso estaba ejecutando en modo usuario (campo `usuario`) y cuántas en modo sistema (campo `sistema`). Nótese que la definición del tipo `struct tiempos_ejec` debería estar disponible tanto para el sistema operativo como para las aplicaciones. Por tanto, se debería incluir tanto en el archivo de cabecera usado por los programas de usuario (`servicios.h`) como en el usado por el sistema operativo (`kernel.h`).

Con respecto a los tres tipos de valores de tiempo que puede devolver esta llamada, hay que comentar los siguientes:

- Sobre el valor devuelto como retorno de la llamada, está relacionado con el tiempo real en el sistema. Normalmente, no se usa de forma absoluta, sino que se comparan los valores tomados por dos llamadas realizadas en distintos instantes para medir el tiempo transcurrido entre esos dos momentos.
- Por lo que se refiere a los tiempos estimados de ejecución, se trata de un muestreo que se realiza en cada interrupción de reloj y puede proporcionar datos aproximados sobre cuánto tiempo lleva ejecutando un proceso y en qué modo. Nótese que debería usarse la función `viene_de_usuario` proporcionada por el módulo HAL para poder distinguir las «muestras».

Por último, se recomienda solicitar al alumno que calcule qué rango de representación proporcionan los distintos valores de tiempo usados en esta práctica.

5.5. PRÁCTICA: IMPLEMENTACIÓN DE UN MANEJADOR DE TERMINAL BÁSICO EN EL MINIKERNEL

5.5.1. Objetivo de la práctica

Esta práctica permite que el alumno pueda ver de forma aplicada cómo funciona un manejador de terminal básico y, especialmente, las dificultades que presenta conseguir una sincronización correcta dentro de este módulo.

NIVEL: Diseño.

HORAS ESTIMADAS: 6.

5.5.2. Descripción de la funcionalidad que debe desarrollar el alumno

Se pretende implementar un servicio básico de lectura del terminal que ofrezca la siguiente llamada al sistema:

```
int leer_caracter();
```

Esta función lee un carácter del terminal y lo devuelve como resultado de la función. Puede parecer sorprendente que la función devuelva un entero en vez de un carácter, pero, por motivos de homogeneidad, se ha seguido el criterio de que todas las llamadas devuelvan un entero. Téngase en cuenta, asimismo, que la clásica función `getchar` de C, en la que se «inspira» esta función, también devuelve un entero.

Las principales características de este manejador del teclado son:

- Ofrece un modo de operación orientado a carácter.
- Asociado al terminal existirá un *buffer* con un tamaño de `TAM_BUF_TERM` caracteres, donde se guardan los caracteres tecleados hasta que algún proceso los solicite.
- Si una petición de lectura encuentra que hay datos disponibles, se satisface inmediatamente. En caso contrario, el proceso queda bloqueado esperando la llegada de un carácter.
- La rutina de interrupción del terminal se encarga de introducir en el *buffer* los datos leídos y desbloquear a los procesos cuando sea oportuno. Si al introducir un carácter se encuentra que el *buffer* está lleno, la rutina ignora el carácter recibido.
- La rutina de interrupción debe encargarse de realizar el eco de los caracteres que recibe.

Como puede observarse, el manejador constituye un ejemplo paradigmático del problema del productor-consumidor:

- La interrupción del terminal «produce» caracteres.
- La llamada al sistema los «consume».

Hay que tener un especial cuidado a la hora de controlar los posibles problemas de sincronización en la operación del manejador. Téngase en cuenta que la situación es compleja, ya que puede haber partes del código de la llamada `leer_caracter` que requieran que el nivel de interrupción sea igual a 3 (o sea, todas las interrupciones inhibidas), mientras que otros fragmentos pueden requerir sólo nivel 2 (o sea, sólo las interrupciones de terminal inhibidas).

Por último, hay que resaltar que el diseño del manejador debe tratar adecuadamente una situación como la que se especifica a continuación:

- El proceso P_1 está bloqueado en `leer_caracter`, ya que no hay caracteres disponibles. En la cola de procesos listos hay dos procesos (P_2 y P_3).
- Llega una interrupción del terminal que desbloquea a P_1 , que pasa al final de la cola de procesos listos.
- P_2 termina y pasa a ejecutar el siguiente proceso P_3 .
- P_3 llama a `leer_caracter`: ¿qué ocurre?

Hay que asegurarse de que sólo uno de los dos procesos (P_1 o P_3) puede leer el carácter tecleado, mientras que el otro se deberá quedar bloqueado. Se admiten como correctas las dos posibilidades. Se proporciona una pista al alumno: una forma de implementar la alternativa en la que P_1 se queda bloqueado y P_3 se lleva el carácter es usar un bucle en vez de una sentencia condicional a la hora de bloquearse en `leer_caracter` si no hay caracteres disponibles.

5.6. PRÁCTICA: DISEÑO Y PROGRAMACIÓN DE UN MANEJADOR DE TERMINAL AVANZADO EN EL MINIKERNEL

5.6.1. Objetivo de la práctica

El principal objetivo es que el alumno pueda conocer de forma práctica la complejidad que lleva asociada un manejador de terminal real.

NIVEL: Diseño.

HORAS ESTIMADAS: 18.

5.6.2. Descripción de la funcionalidad que debe desarrollar el alumno

En la práctica anterior se ha desarrollado un módulo básico de gestión de la entrada del terminal. Esta práctica plantea desarrollar un manejador más complejo que ofrezca, dentro de lo que cabe, funciones similares a las presentes en un sistema UNIX convencional. Las características principales de este módulo serán las siguientes:

- En la práctica anterior se usaba un *buffer* para el terminal con un tamaño muy pequeño (`TAM_BUF_TERM` igual a ocho bytes). Dado que se implementa un modo de operación más sofisticado, parece razonable usar un tamaño mayor (por ejemplo, de treinta y dos bytes).
- La entrada está orientada a líneas. Una solicitud de lectura no se satisface hasta que se haya introducido una línea completa.
- La llamada para leer del terminal es:

```
int leer(void *buf, int tam);
```

Esta llamada devuelve el número de bytes leídos. El sistema operativo copia la información leída en la zona que comienza en la dirección apuntada por `buf`. El segundo parámetro, `tam`, indica cuántos caracteres se solicita leer.

- El comportamiento de la lectura sigue la misma pauta que en UNIX. Así, si en la llamada se solicitan S caracteres y el usuario introduce una línea con L caracteres (incluyendo el carácter de fin de línea '`\n`'), se presentan los dos siguientes casos:

- Si $S \geq L$, se copian los L caracteres introducidos incluyendo el de fin de línea.
- Si $S < L$, se copian los S primeros caracteres de la línea. El resto quedan disponibles para las siguientes peticiones de lectura.

En ningún caso se añade un carácter nulo ('`\0`') al final del *buffer*.

- De manera similar a UNIX, algunos caracteres tendrán asociado un tratamiento especial. Concretamente, se definen los siguientes caracteres especiales:

- Indicador de fin de línea. Este carácter causa que se complete la línea en curso quedando disponible para su lectura. El propio carácter de fin de línea ('`\n`') formará parte de la línea.
- Borrado del anterior carácter tecleado. Este carácter provoca que el anterior deje de formar parte de la línea. Por defecto, se usará el carácter Crtl-H (código ASCII 8 en decimal), aunque, como se comentará más adelante, se podrá redefinir.
- Borrado de la línea. Este carácter provoca la eliminación de todos los caracteres que forman parte de la línea en curso. Por defecto, se usará el carácter Crtl-U (código ASCII 21 en decimal), aunque también se podrá redefinir.
- Carácter de final de entrada de datos. La línea en curso queda disponible para ser leída aunque no se haya tecleado el carácter de fin de línea. En la línea no se incluirá este carácter. Si se pulsa al principio de una línea, leer devuelve 0 bytes, lo que, por convención, se usa para indicar que ha terminado la entrada de datos. Por defecto, se usará el carácter Crtl-D (código ASCII 4 en decimal), aunque también se podrá redefinir.
- Carácter que permite abortar el proceso actual. Por defecto, se usará el carácter Crtl-B (código ASCII 2 en decimal), aunque también se podrá redefinir. Nótese que en este caso no se ha usado el típico de UNIX (crtl-C), ya que queremos que siga activo para «matar» al *minikernel*.
- Carácter de escape. Permite al usuario indicar que el carácter que se tecleará justo a continuación no deberá interpretarse como especial, aunque lo sea. Por defecto, se usará el carácter '\', aunque también se podrá redefinir.

Por simplicidad, no se hará eco de estos caracteres especiales, excepto del carácter indicador de fin de línea.

- Como se comentó previamente, se puede redefinir la asociación entre un carácter y una función especial. Para ello, se ofrecen dos nuevos servicios (hasta cierto punto, similares a las funciones POSIX `tcgetattr` y `tcsetattr`):

```
int obtener_car_control(struct car_cont *car);
int fijar_car_control(struct car_cont *car);
```

Siendo el tipo struct `car_cont`:

```
struct car_cont {
    char cc[5];
};
```

Cada posición del vector corresponde con un carácter especial siguiendo el criterio:

- Posición 0: carácter de borrado del último carácter.
- Posición 1: carácter de borrado de línea.

- Posición 2: carácter de fin de datos.
- Posición 3: carácter para abortar procesos.
- Posición 4: carácter de escape.

Nótese que esta definición de tipo debería estar disponible tanto para el sistema operativo como para las aplicaciones. Por tanto, se debería incluir tanto en el archivo de cabecera usado por los programas de usuario (*servicios.h*) como en el usado por el sistema operativo (*kernel.h*).

- Puesto que no hay una llamada específica para redefinir cada carácter especial, si un programa quiere cambiar un carácter específico tendrá que obtener primero las definiciones actuales usando *obtener_car_control*, modificar el carácter correspondiente y usar *fijar_car_control* para activar la nueva definición. Así, por ejemplo, el siguiente programa establece que el carácter para abortar es el Ctrl-A (código ASCII 1 en decimal).

```
#include "servicios.h"

int main(){
    struct car_cont defs;

    obtener_car_control(&defs);
    defs.cc[3]='\001';
    fijar_car_control(&defs);
    .....
}
```

- En su modo de operación por defecto, el manejador realizará el eco de cada carácter tecleado (excepto de los especiales, como se especificó previamente). Se proporcionará un nuevo servicio que permita activar y desactivar el eco:

```
int eco(int estado);
```

Si *estado* es igual a cero, se desactiva el *eco*, y si es distinto de cero, se reactiva.

- Si se llena el *buffer* asociado al terminal, la rutina de tratamiento de la interrupción ignorará el carácter recibido, excepto si se trata de un carácter especial que no haya que almacenar en el *buffer*.

Para terminar, hay que resaltar que, aunque los aspectos «estéticos» de la edición en el terminal no son trascendentales en el desarrollo de esta práctica, se debería estimular al alumno para que intente implementar algún efecto de esta índole, ya que proporciona una interfaz de usuario más vistosa. Así, por ejemplo, lo importante es asegurar que cuando se teclea un carácter de borrado, no se le entregue a la aplicación el carácter previamente tecleado. Sin embargo, si se consigue además que ese carácter desaparezca de la pantalla, el efecto resultante es más impactante. Una pista: escriba en pantalla la secuencia formada por un carácter \b, un espacio en blanco y otro carácter \b.

5.7. CONCEPTOS BÁSICOS DE ARCHIVOS

Desde el punto de vista de los usuarios y las aplicaciones, los archivos y directorios son los elementos centrales del sistema. Cualquier usuario genera y usa información a través de las apli-

caciones que ejecuta en el sistema. En todos los sistemas operativos de propósito general, las aplicaciones y sus datos se almacenan en archivos no volátiles, lo que permite su posterior reutilización. Cuando en un sistema existen múltiples archivos, es necesario dotar al usuario de algún mecanismo para estructurar el acceso a los mismos. Estos mecanismos son los directorios, agrupaciones lógicas de archivos que siguen criterios definidos por sus creadores o manipuladores. Para facilitar el manejo de los archivos, casi todos los sistemas de directorios permiten usar nombres lógicos, que, en general, son muy distintos de los descriptores físicos que usa internamente el sistema operativo.

Las aplicaciones de una computadora necesitan almacenar información en soporte permanente, tal como discos o cintas. Dicha información, cuyo tamaño puede variar desde unos pocos bytes hasta terabytes, puede tener accesos concurrentes desde varios procesos. Además, dicha información tiene su ciclo de vida que normalmente no está ligado a una única aplicación.

Todos los sistemas operativos proporcionan una unidad de almacenamiento lógico, que oculta los detalles del sistema físico de almacenamiento, denominada **archivo**. *Un archivo es una unidad de almacenamiento lógico no volátil que agrupa un conjunto de información relacionada entre sí bajo un mismo nombre.* Desde el punto de vista del usuario, el archivo es la única forma de gestionar el almacenamiento secundario, por lo que es importante en un sistema operativo definir cómo se nombran los archivos, qué operaciones hay disponibles sobre los archivos, cómo perciben los usuarios los archivos, etc. Internamente, todos los sistemas operativos dedican una parte de sus funciones, agrupada en el **sistema de archivos**, a gestionar los archivos. En este componente del sistema operativo se define cómo se estructuran los archivos, cómo se identifican, cómo se implementan, acceden, protegen, etc.

Desde el punto de vista del sistema operativo, un archivo se caracteriza por tener una serie de atributos. Dichos atributos varían de unos sistemas operativos a otros, pero habitualmente incluyen los siguientes: nombre, identificador único, tipo de archivo, mapa del archivo, protección, tamaño del archivo, información temporal e información de control del archivo. El sistema operativo debe proporcionar, al menos, una estructura de archivo genérico que dé soporte a todos los tipos de archivos mencionados anteriormente, un mecanismo de nombrado, facilidades para proteger los archivos y un conjunto de servicios que permita explotar el almacenamiento secundario y el sistema de E/S de forma sencilla y eficiente. Dicha estructura debe incluir los atributos deseados para cada archivo especificando cuáles son visibles y cuáles están ocultos a los usuarios.

Una de las características principales de un sistema operativo, de cara a los usuarios, es la forma en que se nombran los archivos. Todo objeto archivo debe tener un **nombre** a través del que se pueda acceder a él de forma inequívoca. El tipo de nombres que se usan para los archivos varía de un sistema operativo a otro, aunque todos ellos permiten asignar a los archivos nombres formados por combinaciones de caracteres alfanuméricos y algunos caracteres especiales. La longitud de los nombres puede ser variable. Por ejemplo, MS-DOS permite nombres con una longitud máxima de ocho caracteres, mientras UNIX permite nombres de hasta 4.096 caracteres. Asimismo, algunos sistemas operativos, como MS-DOS o Windows, no distinguen entre mayúsculas y minúsculas, mientras otros, como UNIX, sí lo hacen. Por las razones anteriores, un nombre de archivo como CATALINAPEREZ no sería válido en MS-DOS, pero sí en UNIX, y los nombres CATALINA y catalina denominarían al mismo archivo en MS-DOS, pero a dos archivos distintos en UNIX. Muchos sistemas operativos permiten añadir una o más **extensiones** al nombre de un archivo. Dichas extensiones se suelen separar del nombre con un punto (ejemplo, .txt) y sirven para indicar al sistema operativo, a las aplicaciones o a los usuarios características del contenido del archivo. En UNIX y Windows NT, un archivo puede tener cualquier número de extensiones y de cualquier tamaño.

Desde el punto de vista del usuario, la información de un archivo puede estructurarse como una lista de caracteres, un conjunto de registros secuencial o indexado, etc. Los sistemas operati-

vos más populares, como UNIX o Windows, proporcionan una estructura interna de archivo y una interfaz de programación muy sencilla, pero polivalentes, permitiendo a las aplicaciones construir cualquier tipo de estructura para sus archivos sin que el sistema operativo sea consciente de ello. La traslación desde las direcciones lógicas de un archivo a direcciones físicas de los dispositivos que albergan el archivo es distinta en cada sistema operativo, pero se basa en el mapa del archivo almacenado como parte de los atributos del archivo.

Para poder utilizar la información almacenada en un archivo, las aplicaciones deben acceder a la misma y almacenarla en memoria. Hay distintas formas de acceder a un archivo, pero dependiendo de que se pueda saltar de una posición a otra de un archivo, se distinguen dos métodos de acceso principales: acceso secuencial y acceso directo. El método de **acceso secuencial** sólo permite leer los bytes del archivo en orden, empezando por el principio. No puede saltar de una posición del archivo a otra o leerlo desordenado. Con el método de **acceso directo, o aleatorio**, el archivo se considera como un conjunto de registros, que se pueden acceder desordenadamente moviendo el apuntador de acceso al archivo a uno u otro registro.

5.8. CONCEPTOS BÁSICOS DE DIRECTORIOS

Un sistema de archivos puede ser muy grande. Para poder acceder a los archivos con facilidad, todos los sistemas operativos proporcionan formas de organizar los nombres de archivos mediante **directorios**. Un directorio puede almacenar otros atributos de los archivos tales como ubicación, propietario, etc., dependiendo de cómo haya sido diseñado. Habitualmente, los directorios se implementan como archivos, pero se tratan de forma especial y existen servicios especiales del sistema operativo para su manipulación.

Un **directorio** es un *objeto que relaciona de forma unívoca el nombre de usuario de un archivo y el descriptor interno del mismo usado por el sistema operativo*. Los directorios sirven para organizar y proporcionar información acerca de la organización de los archivos en los sistemas de archivos. Para evitar ambigüedades, un mismo nombre no puede identificar nunca a dos archivos distintos, aunque varios nombres se pueden referir al mismo archivo. Habitualmente, un directorio contiene tantas entradas como archivos son accesibles a través de él, siendo la función principal de los directorios presentar una **visión lógica** simple al usuario, escondiendo los detalles de implementación del sistema de directorios.

Cuando se abre un archivo, el sistema operativo busca en el sistema de directorios hasta que encuentra la entrada correspondiente al nombre del archivo. A partir de dicha entrada, el sistema operativo obtiene el identificador interno del archivo y, posiblemente, algunos de los atributos del mismo. Esta información permite pasar del nombre de usuario al objeto archivo que maneja el sistema operativo.

Al igual que un archivo, un directorio es un objeto y debe ser representado por una estructura de datos. Actualmente, los directorios almacenan únicamente el identificador del descriptor de archivo (en UNIX, el nodo-i) y colocan los atributos del objeto archivo dentro de la estructura de datos asociada a su descriptor. Con esta solución, la entrada de directorio es una estructura de datos muy sencilla que contiene únicamente el nombre del archivo asociado a ella y el identificador del descriptor de archivo, número de nodo-i, usado por el sistema operativo.

Independientemente de cómo se defina la entrada de un directorio, es necesario organizar todas las entradas de directorio para poder manejar los archivos existentes en un sistema de almacenamiento de forma sencilla y eficiente. Actualmente, los sistemas de directorios tienen una estructura de árbol que representa todos los directorios y subdirectorios del sistema partiendo de un **directorio raíz**, existiendo un camino único (**path**) que atraviesa el árbol desde la raíz hasta un archivo determinado. Los nodos del árbol son directorios que contienen un conjunto de subdirec-

torios o archivos. Las hojas del árbol son siempre archivos. Normalmente, cada usuario tiene su propio directorio **home** a partir del cual se cuelgan sus subdirectorios y archivos y en el que le sitúa el sistema operativo cuando entra a su cuenta. Este directorio lo decide el administrador, o los procesos de creación de nuevos usuarios, cuando el usuario se da de alta en el sistema y está almacenado junto con otros datos del usuario en una base de datos o archivo del sistema operativo. En el caso de UNIX, por ejemplo, el archivo `/etc/password` contiene una línea por usuario del sistema similar a la siguiente:

```
miguel:*:Miguel:/users/miguel:/etc/bin
```

donde el directorio **home** es `/users/miguel`. MS-DOS y Windows NT tienen registros de usuarios que también definen el directorio **home**.

Los usuarios pueden subir y bajar por el árbol de directorios, mediante servicios del sistema operativo, siempre que tengan los permisos adecuados. Por ello, tanto el propio usuario como el sistema operativo deben conocer dónde están en cada instante. Para solventar este problema se definió el concepto de **directorio de trabajo**, que es el directorio en el que un usuario se encuentra en un instante determinado. Para crear o borrar un archivo o directorio se puede indicar su nombre relativo al directorio de trabajo o completo desde la raíz a las utilidades del sistema que llevan a cabo estas operaciones.

La especificación del nombre de un archivo en un árbol de directorios, o en un grafo acíclico, toma siempre como referencia el directorio raíz (`/` en UNIX, `\` en MS-DOS). A partir de este directorio, es necesario viajar por los sucesivos subdirectorios hasta encontrar el archivo deseado. Para ello, el sistema operativo debe conocer el nombre completo del archivo a partir del directorio raíz. Hay dos posibles formas de obtener dicho nombre:

- Que el usuario especifique el nombre completo del archivo, denominado **nombre absoluto**.
- Que el usuario especifique el nombre de forma relativa, denominado **nombre relativo**, a algún subdirectorio del árbol de directorios.

El **nombre absoluto** de un archivo *proporciona todo el camino a través del árbol de directorios desde la raíz hasta el archivo*. Un nombre absoluto es *autocontenido*, en el sentido de que proporciona al sistema de archivos toda la información necesaria para llegar al archivo, sin que tenga que suponer o añadir ninguna información de entorno del proceso o interna al sistema operativo. Algunas aplicaciones necesitan este tipo de nombres para asegurar que sus programas funcionan independiente del lugar del árbol de directorios en que estén situados. Por ejemplo, un compilador de lenguaje C, en una máquina que ejecuta el sistema operativo UNIX, necesita saber que los archivos con definiciones de macros y tipos de datos están en el directorio `/usr/include`. Es necesario especificar el nombre absoluto, porque no es posible saber en qué directorio del árbol instalará cada usuario el compilador. Además, la mayoría de los sistemas operativos modernos permiten definir **nombres relativos**, es decir, *nombres de archivo que sólo especifica una porción del nombre absoluto a partir de un determinado subdirectorio del árbol de nombres*. Los nombres relativos no se pueden interpretar si no se conoce el directorio del árbol a partir del que empiezan; para ello existe un **directorio de trabajo**, o **actual**, a partir del cual se interpretan siempre los nombre relativos. Es responsabilidad del usuario colocarse en el directorio de trabajo adecuado antes de usar nombres relativos al mismo.

Muchos sistemas operativos con directorios jerárquicos tienen dos entradas especiales, . y .., en cada directorio para referirse a sí mismos y a su directorio *padre* en la jerarquía. Estas entradas especiales son muy útiles para especificar posiciones relativas al directorio actual y para viajar por el árbol.

5.9. SERVICIOS DE ARCHIVOS Y DIRECTORIOS

En este apartado se describen los servicios del sistema operativo necesarios para hacer las prácticas que se proponen en las secciones siguientes. En la mayoría de los sistemas operativos modernos, los directorios se implementan como archivos que almacenan una estructura de datos definida: entradas de directorios. Por ello, los servicios de archivos pueden usarse directamente sobre directorios. Sin embargo, la funcionalidad necesaria para los directorios es mucho más restringida que la de los archivos, por lo que los sistemas operativos suelen proporcionar servicios específicos para satisfacer dichas funciones de forma eficiente y sencilla.

5.9.1. Servicios POSIX para archivos y directorios

POSIX proporciona una visión lógica de archivo equivalente a una tira secuencial de bytes. Para acceder al archivo se mantiene un apuntador de posición, a partir del cual se ejecutan las operaciones de lectura y escritura sobre el archivo. Para identificar a un archivo, el usuario usa nombres al estilo de UNIX, como, por ejemplo, /users/miguel/datos. Cuando se abre un archivo, se devuelve un descriptor de archivo, que se usa a partir de ese momento para identificar al archivo en otras llamadas al sistema. Estos descriptores son números enteros de 0 a n y son específicos para cada proceso. Cuando se realiza una operación open, el sistema de archivos busca desde la posición 0 hasta que encuentra una posición libre, siendo ésa la ocupada. Cuando se cierra un archivo (close), se libera la entrada correspondiente. En los sistemas UNIX, cada proceso tiene tres descriptores de archivos abiertos por defecto. Estos descriptores ocupan las posiciones 0 a 2 y reciben los siguientes nombres:

- Entrada estándar, `fd = 0`.
- Salida estándar, `fd = 1`.
- Error estándar, `fd = 2`.

El objetivo de estos descriptores estándar es poder escribir programas que sean independientes de los archivos sobre los que han de trabajar.

Usando servicios de POSIX, se pueden consultar los atributos de un archivo. Estos atributos son una parte de la información existente en el descriptor interno del archivo (nodo-i). Entre ellos se encuentran el número de nodo-i, el sistema de archivos al que pertenece, su dispositivo, tiempos de creación y modificación, número de enlaces físicos, identificación de usuario y grupo, tamaño óptimo de acceso, modo de protección, etc. El modo de protección es especialmente importante porque permite controlar el acceso al archivo por parte de su dueño, su grupo y el resto del mundo. En POSIX, estos permisos de acceso se especifican usando máscaras de nueve bits con el siguiente formato:

dueño	grupo	mundo
<code>rwx</code>	<code>rwx</code>	<code>rwx</code>

Los prototipos de los **servicios POSIX para archivos** que se usan en las prácticas son:

```
int creat(const char *path, mode_t mode);
```

Su efecto es la creación de un archivo con nombre `path` y modo de protección `mode`. En caso de que el archivo no pueda ser creado, devuelve -1 y pone el código de error adecuado en la variable `errno`.

```
int unlink(const char *path);
```

Este servicio permite borrar un archivo indicando su nombre. El argumento path indica el nombre del archivo a borrar.

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

Este servicio permite abrir un archivo indicando su nombre en el argumento path. El argumento oflag permite especificar qué tipo de operación se quiere hacer con el archivo: lectura (O_RDONLY), escritura (O_WRONLY), lectura-escritura (O_RDWR), añadir información nueva (O_APPEND), creación (O_CREAT), truncado (O_TRUNC), escritura no bloqueante (O_NONBLOCK), etc. Si el archivo no existe, no se puede abrir con las características especificadas o no se puede crear, la llamada devuelve -1 y un código de error en la variable errno.

```
int close(int fildes);
```

Esta llamada libera el descriptor de archivo obtenido cuando se abrió el archivo, dejándolo disponible para su uso posterior por el proceso.

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

Estos servicios permiten a un proceso leer y escribir datos de un archivo, que debe abrirse previamente, y copiarlos a su espacio de memoria. El descriptor de archivo se indica en fildes; la posición de memoria donde copiar, o de donde escribir, los datos se especifican en el argumento buf, y el número de bytes a leer, o escribir, se especifica en nbyte. La lectura se lleva a cabo a partir de la posición actual del apuntador de posición del archivo. Si la llamada se ejecuta correctamente, devuelve el número de bytes leídos realmente, que pueden ser menos que los pedidos, y se incrementa el apuntador de posición del archivo con esa cantidad.

```
off_t lseek(int fildes, off_t offset, int whence);
```

Esta llamada permite cambiar el valor del apuntador de posición de un archivo abierto, de forma que posteriores operaciones de E/S se ejecuten a partir de esa posición. El descriptor de archivo se indica en fildes, el desplazamiento se indica en offset y el lugar de referencia para el desplazamiento se indica en whence.

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

POSIX especifica los servicios stat y fstat para consultar los distintos atributos de un archivo, si bien en el caso de fstat dicho archivo debe estar abierto. Ambas devuelven una estructura de tipo stat.

A continuación se describen los **servicios POSIX para directorios** más comunes.

```
int mkdir (const char *path, mode_t mode);
```

El servicio mkdir permite crear un nuevo directorio en POSIX. Esta llamada al sistema crea el directorio especificado en path con el modo de protección especificado en mode.

```
int rmdir (const char *path);
```

Permite borrar un directorio especificando su nombre. El directorio se borra únicamente cuando está vacío.

```
DIR *opendir(const char *dirname);
int closedir (DIR *dirp);
```

Abre el directorio de nombre especificado en la llamada y devuelve un identificador de directorio. El apuntador de posición indica a la primera entrada del directorio abierto. En caso de error devuelve NULL. Un directorio abierto, e identificado por dirp, puede ser cerrado ejecutando la llamada closedir.

```
struct dirent *readdir (DIR *dirp);
```

Este servicio permite leer de un directorio abierto, obteniendo como resultado la siguiente entrada del mismo.

5.9.2. Servicios Win32 para archivos y directorios

Windows NT proporciona una visión lógica de archivo equivalente a una tira secuencial de bytes. Para acceder al archivo, se mantiene un apuntador de posición, a partir del cual se ejecutan las operaciones de lectura y escritura sobre el archivo. Para identificar a un archivo, el usuario usa nombres jerárquicos, como, por ejemplo, C:\users\miguel\datos. Cada archivo se define como un objeto dentro del núcleo de Windows NT. Por ello, cuando se abre un archivo, se crea en memoria un objeto archivo y se le devuelve al usuario un manejador (HANDLE) para ese objeto.

A continuación se muestran los **servicios de Win32 para gestión de archivos**. Como se puede ver, son similares a los de POSIX, si bien los prototipos de las funciones que los proporcionan son bastante distintos.

```
HANDLE CreateFile(LPCSTR lpFileName, DWORD dwDesiredAccess,
                   DWORD dwShareMode, LPVOID lpSecurityAttributes,
                   DWORD CreationDisposition, DWORD dwFlagsAndAttributes,
                   HANDLE hTemplateFile);
```

Su efecto es la creación, o apertura, de un archivo con nombre lpFileName, modo de acceso dwDesiredAccess (GENERIC_READ, GENERIC_WRITE) y modo de compartición dwShareMode (NO_SHARE, SHARE_READ, SHARE_WRITE).

```
BOOL DeleteFile(LPCTSTR lpszFileName);
```

Este servicio permite borrar un archivo indicando su nombre. El nombre del archivo a borrar se indica en lpszFileName.

```
BOOL CloseHandle(HANDLE hObject);
```

La llamada CloseHandle libera el descriptor de archivo obtenido cuando se abrió el archivo, dejándolo disponible para su uso posterior.

```
BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer,
               DWORD nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead,
```

```

    LPVOID lpOverlapped);
BOOL WriteFile(HANDLE hFile, LPVOID lpBuffer,
    DWORD nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten,
    LPVOID lpOverlapped);

```

Estos servicios permiten a un proceso leer, o escribir, datos de un archivo abierto y copiarlos a, o desde, su espacio de memoria. El manejador del archivo se indica en `hFile`, la posición de memoria para los datos se especifica en el argumento `lpBuffer` y el número de bytes a leer, o escribir, se especifica en `lpNumberOfBytesToRead`, `lpNumberOfBytesToWrite`.

```

DWORD SetFilePointer(HANDLE hFile, LONG lDistanceToMove,
    LONG FAR *lpDistanceToMoveHigh, DWORD dwMoveMethod);

```

Esta llamada permite cambiar el valor del apuntador de posición de un archivo abierto previamente, de forma que operaciones posteriores de E/S se ejecuten a partir de esa posición. El manejador de archivo se indica en `hFile`, el desplazamiento (positivo o negativo) se indica en `lDistanceToMove` y el lugar de referencia para el desplazamiento se indica en `dwMoveMethod`.

A continuación se describen los **servicios de Win32 para gestión de directorios**.

```

BOOL CreateDirectory(LPCSTR lpPathName,
    LPVOID lpSecurityAttributes);

```

El servicio `CreateDirectory` permite crear en Win32 el directorio especificado en `lpPathName` con el modo de protección especificado en `lpSecurityAttributes`.

```

BOOL RemoveDirectory (LPCSTR lpszPath);

```

Permite borrar el directorio `lpszPath`, que se borra únicamente cuando está vacío.

En Win32 hay tres servicios que permiten leer un directorio: `FindFirstFile`, `FindNextFile` y `FindClose`. Su prototipos son:

```

HANDLE FindFirstFile(LPCSTR lpFileName,
    LPWin32_FIND_DATA lpFindFileData);
BOOL FindNextFile(HANDLE hFindFile,
    LPWin32_FIND_DATA lpFindFileData);
BOOL FindClose(HANDLE hFindFile);

```

`FindFirstFile` es equivalente al `opendir` de POSIX. Permite obtener un manejador para buscar en un directorio. Además, busca la primera ocurrencia del nombre de archivo especificado en `lpFileName`. `FindNextFile` es equivalente al `readdir` de POSIX, permite leer la siguiente entrada de archivo en un directorio que coincide con el especificado en `lpFileName`. `FindClose` es equivalente a `closedir` de POSIX y permite cerrar el manejador de búsqueda en el directorio.

5.10. PRÁCTICA: MONITORIZACIÓN DE ARCHIVOS: FILEMON

5.10.1. Objetivos de la práctica

El objetivo de esta práctica es que el alumno se familiarice con una herramienta de monitorización de archivos y directorios y sea capaz de observar los eventos del sistema significativos

para estos objetos. Para ello se usará la herramienta Filemon, un monitor muy sencillo, pero eficiente, que se puede conseguir de forma gratuita, junto con su material de ayuda, en la página web del libro.

El monitor está disponible para Windows y Linux, por lo que esta práctica se puede llevar a cabo en ambos sistemas operativos o en cualquiera de ellos.

NIVEL: Introducción.

HORAS ESTIMADAS: 6.

5.10.2. Descripción de la funcionalidad que debe desarrollar el alumno

Filemon es un monitor del sistema de archivos. Una vez ejecutado, registra todas las operaciones relacionadas con los archivos. Por ejemplo, cada vez que se hace doble clic en el ícono MiPC, entra en un directorio o abre un archivo desde cualquier aplicación, Filemon captura todas esas acciones. El resultado es una pantalla como la de la Figura 5.3, en la que, como se puede apreciar, hay varias columnas: la primera (#) se corresponde con el código de la operación (un número que se asigna a cada operación), la segunda columna (Time) es la hora y la tercera columna (Process) es el programa que accedió al archivo. En la Figura 5.3 puede ver que el programa Mspub (Microsoft Publisher) realizó varias operaciones en el disco (estaba generando el archivo HTML de una

#	Time	Process	Request	Path	Result	Other
35170	18:43:00.254	Kodakimg	Close	C:\WINDOWS\SYSTEM\VOIDIS400.DLL	SUCCESS	CLOSE_FINAL
35171	18:43:00.254	Kodakimg	Close	C:\WINDOWS\SYSTEM\VOIDADM400.DLL	SUCCESS	CLOSE_FINAL
35172	18:43:00.339	Kodakimg	Read	C:\WINDOWS\SYSTEM\LEP032.DLL	SUCCESS	Offset: 98304 Length: 4096
35173	18:43:00.359	Kodakimg	Read	C:\WINDOWS\SYSTEM\DF792B.TMP	SUCCESS	Offset: 12484608 Length: 40
35174	18:43:00.439	KERNEL32	Close	C:\WINDOWS\SYSTEM\LEV032.DLL	SUCCESS	CLOSE_FINAL
35175	18:43:00.439	KERNEL32	Close	C:\WINDOWS\SYSTEM\MSCMS.DLL	SUCCESS	CLOSE_FINAL
35176	18:43:00.439	KERNEL32	Close	C:\WINDOWS\SYSTEM\KODAKIMG.EXE	SUCCESS	CLOSE_FINAL
35177	18:43:00.439	KERNEL32	Close	C:\WINDOWS\SYSTEM\UMDCHN.DLL	SUCCESS	CLOSE_FINAL
35178	18:43:00.444	KERNEL32	Close	C:\WINDOWS\SYSTEM\H01GIF400.DLL	SUCCESS	CLOSE_FINAL
35179	18:43:00.444	KERNEL32	Close	C:\WINDOWS\SYSTEM\H01GIF400.DLL	SUCCESS	CLOSE_FINAL
35180	18:43:05.843	MSGSRV/32	Attributes	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	SeAttributes
35181	18:43:05.843	MSGSRV/32	Open	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	OPENFILEISINGWRITEON
35182	18:43:05.888	MSGSRV/32	Write	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Offset: 0 Length: 32
35183	18:43:05.888	MSGSRV/32	Commit	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	NOACCESSUPDATE
35184	18:43:09.688	MSGSRV/32	Seek	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Beginning Offset: 32 / New ...
35185	18:43:09.688	MSGSRV/32	Write	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Offset: 32 Length: 32
35186	18:43:09.688	MSGSRV/32	Seek	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Beginning Offset: 28704 / N ...
35187	18:43:09.688	MSGSRV/32	Write	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Offset: 28704 Length: 29672
35188	18:43:09.688	MSGSRV/32	Seek	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Beginning Offset: 57376 / N ...
35189	18:43:09.688	MSGSRV/32	Write	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Offset: 57376 Length: 20480
35190	18:43:09.688	MSGSRV/32	Seek	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Beginning Offset: 385056 /
35191	18:43:09.688	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 3751336 Length: 4096
35192	18:43:09.688	MSGSRV/32	Write	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Offset: 385056 Length: 51440
35193	18:43:09.928	MSGSRV/32	Seek	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Beginning Offset: 565200 /
35194	18:43:09.928	MSGSRV/32	Write	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Offset: 565200 Length: 45056
35195	18:43:09.928	MSGSRV/32	Seek	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Beginning Offset: 610336 /
35196	18:43:09.928	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11476932 Length: 40 ...
35197	18:43:09.928	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11476936 Length: 40 ...
35198	18:43:09.928	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11468900 Length: 40 ...
35199	18:43:09.928	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11464704 Length: 40 ...
35200	18:43:09.933	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11462608 Length: 40 ...
35201	18:43:09.933	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11459512 Length: 40 ...
35202	18:43:09.933	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11452416 Length: 40 ...
35203	18:43:09.933	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11448320 Length: 40 ...
35204	18:43:09.933	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11444224 Length: 40 ...
35205	18:43:09.933	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11440128 Length: 40 ...
35206	18:43:09.933	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11436032 Length: 40 ...
35207	18:43:09.933	MSGSRV/32	Read	C:\Windows\SYSTEM\DF792B.TMP	SUCCESS	Offset: 11431936 Length: 40 ...
35208	18:43:09.928	MSGSRV/32	Write	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Offset: 610336 Length: 51440
35209	18:43:09.933	MSGSRV/32	Seek	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Beginning Offset: 671776 /
35210	18:43:09.933	MSGSRV/32	Write	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Offset: 671776 Length: 20480
35211	18:43:09.933	MSGSRV/32	Commr	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	NOACCESSUPDATE
35212	18:43:10.028	MSGSRV/32	Seek	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Beginning Offset: 0 / New of ...
35213	18:43:10.028	MSGSRV/32	Write	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	Offset: 0 Length: 32
35214	18:43:10.028	MSGSRV/32	Close	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	CLOSE_FINAL
35215	18:43:10.028	MSGSRV/32	Attributes	C:\Windows\S\PROFILES\JCARRETE\USER.DAT	SUCCESS	SeAttributes

Figura 5.3. Pantalla de salida de Filemon.

página web). También puede ver el tipo de operación (Request), el archivo accedido (Path) y otras informaciones de carácter general (Result) y (Other).

La práctica tiene dos partes:

1. **Instalación del monitor Filemon** en la máquina de prácticas. El alumno deberá ir a la página web anteriormente citada y traer la versión del monitor que corresponda con su sistema operativo (Windows 9x, Windows NT/2000, Linux). Nota: Para poder instalar la utilidad en Windows 2000, y para poder ejecutarla, se necesitan permisos de administrador.
2. **Activación del monitor Filemon durante cinco minutos**, captura y elaboración de los datos de salida.

La primera parte de la práctica es muy sencilla, ya que la aplicación se descarga de la web en un archivo comprimido tipo ZIP. Basta con extraer los archivos y ya están listos para funcionar. El monitor es un programa de apenas 120 KB, pero con una gran utilidad para aquellos usuarios interesados en el funcionamiento interno del sistema operativo Windows o Linux. Haga doble clic en el archivo Filemon.exe para ejecutar el programa.

La segunda parte tiene tres pasos:

1. **Control de número de operaciones** de archivos que hacen determinadas aplicaciones al abrirse. Para ello abra un Explorer, haga doble clic sobre MiPC y abra el Internet Explorer. En cada caso anote el número de operaciones antes y después y calcule las operaciones realizadas sobre archivos.
2. **Filtrado de operaciones** para estudiar sólo determinados tipos. Para ello, el alumno deberá filtrar las operaciones para que aparezcan sólo las operaciones Open, Close, Read y Write. Para filtrar operaciones pulse Edit -> Filter/Highlight y seleccione Highlight de esas operaciones. Se verán rojas en la pantalla. El alumno deberá capturar una de estas pantallas con PrintScreen e incluirla en su memoria.
3. **Registro de la monitorización** en un archivo de texto filemon.log. Ábralo como una hoja de cálculo de Excel y aplique un autofiltro sobre la columna Request. Basándose en el filtrado, el alumno deberá responder a las siguientes preguntas:
 - ¿Cuántas operaciones se han hecho en total?
 - Incluya en su memoria la lista de las operaciones ejecutadas durante la monitorización.
 - Realice un gráfico como el que se muestra en la Figura 5.4. Dicho gráfico muestra todas las operaciones realizadas y el número de las mismas. ¿Qué conclusiones se pueden sacar de este estudio?



Figura 5.4. Número y tipo de operaciones sobre archivos.

5.10.3. Recomendaciones generales

Antes de empezar con la monitorización de todo el sistema de archivos, se recomienda monitorear un directorio pequeño. El del usuario, por ejemplo, `c:\jesus*.*`, puede ser un buen ejemplo. De esta forma será capaz de comprender mejor el funcionamiento de Filemon y de realizar un mejor análisis de los resultados finales. El directorio se elige mediante la opción `Edit->Filter->Include`.

Si quiere observar el efecto de determinadas operaciones, como, por ejemplo, la copia de un archivo de un directorio a otro, se podría ver muy bien probando con este directorio más restringido.

El monitor tiene una buena herramienta de ayuda en línea, úsela.

5.10.4. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- `Memoria.doc` : Memoria de la práctica.
- `Filemon.log`: Archivo que contiene el resultado de la sesión de monitorización.

5.10.5. Bibliografía

- D. Solomon y M. Russinovich. *Inside Windows 2000*. 3.^a edición, Microsoft Press, 2000.
- R. Nagar. *Windows NT File System Internals*. O'Reilly and Associates, 1997.

5.11. PRÁCTICA: GESTIÓN DE CUOTAS DE DISCO USANDO *SCRIPTS* DE UNIX

5.11.1. Objetivos de la práctica

El objetivo de esta práctica es que el alumno se familiarice con la línea de mandatos de UNIX y con la programación de *scripts* en UNIX como método de automatización para las tareas de administración. En este caso, se gestionan las cuotas de disco asociadas a los usuarios.

NIVEL: Introducción.

HORAS ESTIMADAS: 6.

5.11.2. Descripción de la funcionalidad que debe desarrollar el alumno

La práctica consiste en desarrollar un *script* para un intérprete de mandatos de UNIX, de nombre `gestion_cuotas`, que pida un nombre de usuario y le asigne una cuota de disco a elegir entre tres: mínima (20 M), media (40 M) o máxima (100 M). Si ya tiene cuota asignada, se indica y se pregunta si la quiere modificar. En caso positivo, se solicita la cuota a aplicar y se asigna.

La sintaxis del mandato debe ser la siguiente:

- `gestion_cuotas`

El funcionamiento detallado del *script* es el siguiente:

- Tras ejecutar el mandato, debe pedir un nombre de usuario. El *script* debe mirar si el usuario existe en el sistema. En caso negativo, da un error por pantalla.
- Si el usuario existe, debe mirar si ya tiene cuota de disco asignada (mediante el mandato *quota*). Si no la tiene, debe pedir el tipo de cuota y asignársela mediante el mandato *setquota*.
- Si ya tiene cuota, se pregunta si se quiere modificar. En caso positivo, debe pedir el tipo de cuota y asignársela mediante el mandato *setquota*.

El resultado de *gestion_cuotas* asigna una cuota del tipo deseado a un usuario del sistema. Este *script* podría ser parte de uno más grande que creara cuentas de usuario en un sistema UNIX.

5.11.3. Recomendaciones generales

Para realizar el programa se recomienda estudiar con detalle la página de manual referente al *bash*, donde se encuentra casi todo lo necesario para realizar la práctica. Para poder guardar dicha información en un archivo ASCII se debe ejecutar el siguiente mandato:

```
man bash | groff -t -e -mandoc -Tascii | col -bx > bash.txt
```

Especial atención merece la opción `#set -x`, que permite ver todo lo que hace el *script* y es muy útil para su depuración mientras se programa.

Además, conviene estudiar en detalle el comportamiento de los mandatos *quota* y *setquota*.

5.11.4. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- *memoria.txt*: Memoria de la práctica.
- *gestion_cuotas*: Archivo que contiene el *script*.

5.11.5. Bibliografía

- A. Afzal. *Introducción a UNIX*. Prentice-Hall, 1997.
- C. Newham y B. Rosenblatt. *Learning the bash shell*. Sebastopol: O'Reilly, 1995.
- S. R. Bourne. *The UNIX System*. Addison-Wesley, 1983.
- J. Carretero, F. García, A. Calderón y J. Fernández. *Diseño e implementación de programas en C*. Pearson Educación, 2002.

5.12. PRÁCTICA: IMPLEMENTACIÓN DEL MANDATO du

5.12.1. Objetivos de la práctica

Esta práctica permitirá al alumno familiarizarse con los servicios que ofrece el estándar POSIX para la gestión de archivos y directorios.

NIVEL: Introducción.

HORAS ESTIMADAS: 6.

5.12.2. Descripción de la funcionalidad que debe desarrollar el alumno

Se trata de desarrollar un programa similar al mandato du de UNIX, denominado midu. A continuación se describe este programa.

Programa midu

Este programa deberá calcular el tamaño ocupado por todos los archivos que se encuentran en un determinado árbol de directorios. La sintaxis de este programa será la siguiente:

- midu directorio

El programa deberá recorrer todos los archivos situados dentro del directorio recibido como argumento. Por cada archivo deberá imprimir una línea con el nombre del archivo y su tamaño expresado en bytes. La última línea a imprimir por el programa será el tamaño total ocupado por todos los archivos. Este último tamaño también se expresará en bytes. El directorio debe recorrerse en profundidad.

A modo de ejemplo, considérese el árbol de directorios situado debajo del directorio dir (véase la Figura 5.5). Las entradas de directorio con nombre fi representan nombres de archivo y las entradas con nombre di nombres de directorio. El número encerrado entre paréntesis indica el tamaño que ocupa el archivo.

La salida que debe imprimir el programa midu deberá ser la siguiente y en ese mismo orden:

```
f1 200
f2 300
d2/f3 100
d2/f4 150
d3/f5 600
'd3/d4/f6 200
d3/d4/f7 300
f8 100
```

Espacio total ocupado: 1900 bytes

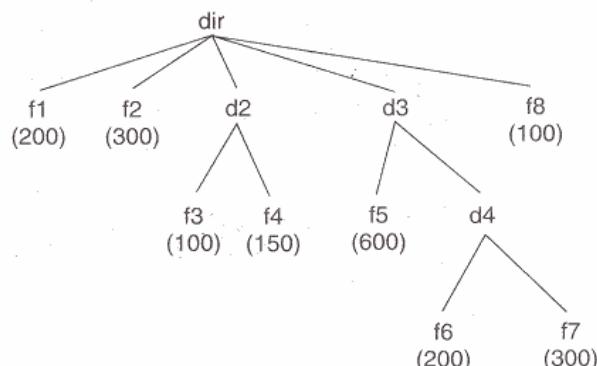


Figura 5.5. Árbol de directorios que recorre el programa du.

Para realizar el recorrido en profundidad del directorio puede emplearse el siguiente algoritmo:

```
recorrido(directorio)
{
    para cada entrada del directorio
    {
        if (entrada es un directorio)
            recorrido(entrada);
        else
            Indicar el espacio que ocupa el archivo
    }
}
```

El programa deberá realizar un correcto tratamiento de todos los errores.

5.12.3. Código fuente de apoyo

Para facilitar la realización de la práctica se recomienda proporcionar a los alumnos un archivo comprimido que contenga el código fuente de apoyo. Dentro de un archivo como el propuesto, existente en la página web del libro (archivo `practica-5.12.tar.gz`), se han incluido los siguientes archivos:

- **Makefile**: Archivo fuente para la herramienta `make`. Con él se consigue la recopilación automática de los archivos fuente cuando se modifiquen. Basta con ejecutar el mandato `make` para que el programa se compile de forma automática.
- **midu.c**: Archivo fuente de C donde se incluirá el programa `midu`.

5.12.4. Recomendaciones generales

Es importante estudiar previamente el funcionamiento del mandato `du` de UNIX (`man du`).

5.12.5. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- **memoria.txt**: Memoria de la práctica.
- **midu.c**: Código fuente del programa `midu`.

5.12.6. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- J. Carretero, F. García, A. Calderón y J. Fernández. *Diseño e implementación de programas en C*. Pearson Educación, 2002.
- M. J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, 1985.
- W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

5.13. PRÁCTICA: GESTIÓN DE ARCHIVOS USANDO *SCRIPTS* DE UNIX

5.13.1. Objetivos de la práctica

El objetivo de esta práctica es que el alumno se familiarice con la línea de mandatos de archivos disponibles en UNIX.

NIVEL: Introducción.

HORAS ESTIMADAS: 6.

5.13.2. Descripción de la funcionalidad que debe desarrollar el alumno

La práctica consiste en desarrollar un *script* para un intérprete de mandatos de UNIX, de nombre *purgar*, que recorra una lista de archivos y realice una copia, en un directorio auxiliar, de todos aquellos archivos cuyo tamaño sea igual a cero.

La sintaxis del mandato debe ser la siguiente:

```
purgar [-R] archivo[...]
```

El funcionamiento detallado del *script* es el siguiente:

- Si se ejecuta el mandato sin parámetros, debe imprimirse por pantalla un mensaje de ayuda y después el programa debe terminar. El mensaje de ayuda será el siguiente:

```
Use: purgar [-R] archivo[...]
```

- El programa deberá recorrer la lista de archivos proporcionada como parámetro en la línea de mandatos. Aquellos archivos que sean archivos regulares (eso excluye los directorios) y cuyo tamaño sea igual a cero, deberán ser copiados a un directorio nuevo, de nombre *.aux*, que deberá crear el *script* en el directorio de trabajo actual.
- Si se activa la opción *-R*, el *script*, además de procesar los archivos regulares, deberá recorrer de forma recursiva todos y cada uno de los directorios que se encuentran como parámetro en la línea de mandatos, copiando al directorio *.aux* todos los archivos regulares de tamaño cero que encuentre.

Por ejemplo, para el árbol de directorios siguiente:

```
pepe
.
.
.
f 1974 f1.o
d 1024 f2
    f 0 vacío
    f 23 lleno
f 0 f3.c
f 2345 f5.c
f 19288 f7.c
```

El resultado de *purgar pepe* copiaría en el directorio el archivo *f3.c*. El resultado de *purgar -R pepe* sería la copia de *f3.c* y de *vacío*.

5.13.3. Recomendaciones generales

Para realizar el programa se recomienda estudiar con detalle la página de manual referente al *bash*, donde se encuentra casi todo lo necesario para realizar la práctica. Para poder guardar dicha información en un archivo ASCII se debe ejecutar el siguiente mandato:

```
man bash | groff -t -e -mandoc -Tascii | col -bx > bash.txt
```

Especial atención merece el mandato `#set -x`, que permite ver todo lo que hace el *script* y es muy útil para su depuración mientras se programa.

5.13.4. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- *memoria.txt*: Memoria de la práctica.
- *purgar*: Archivo que contiene el *script*.

5.13.5. Bibliografía

- A. Afzal. *Introducción a UNIX*. Prentice-Hall, 1997.
- C. Newham y B. Rosenblatt. *Learning the bash shell*. O'Reilly, 1995.
- S. R. Bourne. *The UNIX System*. Addison-Wesley, 1983.

5.14. PRÁCTICA: INTÉRPRETE DE MANDATOS CON REDIRECCIÓN Y MANDATOS INTERNOS

El objetivo de esta práctica es completar la funcionalidad de las Prácticas 2.8 y 4.2, incluyendo la funcionalidad necesaria para la ejecución de mandatos y secuencias de mandatos con redirección de archivos y mandatos internos relacionados con archivos.

NIVEL: Intermedio.

HORAS ESTIMADAS: 4.

5.14.1. Descripción de la funcionalidad que debe desarrollar el alumno

La práctica a desarrollar tiene que completar la funcionalidad de las Prácticas 2.8 y 4.2. Para el desarrollo de esta práctica debe utilizarse el mismo material de apoyo y la misma función `obtain_order` descrita en las prácticas anteriores y que permite obtener el mandato o secuencia de mandatos que hay que ejecutar. A partir de la Práctica 4.2, desarrollada en el Capítulo 4, el alumno deberá incluir la siguiente funcionalidad:

1. *Ejecución de mandatos con redirecciones* (entrada, salida y de error).
2. *Ejecución de mandatos internos*. Un mandato interno es aquél que bien se corresponde directamente con una llamada al sistema o bien es un complemento que ofrece el propio

minishell. Para que su efecto sea permanente ha de ser implementado y ejecutado dentro del propio *minishell*. Será ejecutado en un *subshell* sólo si se invoca en *background* o aparece en una secuencia y no es el último. Todo mandato interno comprueba el número de argumentos con que se le invoca y si encuentra este o cualquier otro error lo notifica (por el error estándar) y termina devolviendo un **valor** distinto de cero.

Los mandatos internos a desarrollar en el *minishell* son:

- **cd [Directorio]**. Cambia el directorio por defecto. Si aparece Directorio, debe cambiar al mismo. Si no aparece, cambia al directorio especificado en la variable de entorno HOME. Presenta (por la salida estándar) como resultado el camino absoluto al directorio actual de trabajo con el formato "%s\n".
- **umask [Valor]**. Cambia la máscara de creación de archivos. Presenta (por la salida estándar) como resultado el valor de la actual máscara con el formato "%o\n". Además, si aparece Valor (dado en octal, man strtol), cambia la máscara a dicho valor.

5.14.2. Código fuente de apoyo

Para la realización de esta práctica, el alumno debe partir del código desarrollado en la Práctica 4.2 y que permitía la ejecución de mandatos sencillos y secuencias de mandatos conectados por tuberías. Sobre este código deberá incorporarse la funcionalidad descrita en el apartado anterior.

5.14.3. Entrega de documentación

El alumno deberá entregar los siguientes archivos:

- memoria.txt: Memoria de la práctica.
- main.c: Código fuente del *minishell*, implementando todas las funcionalidades que se requieren en las Prácticas 2.9, 4.2 y en esta misma.

5.14.4. Bibliografía

- F. García, J. Carretero, J. Fernández y A. Calderón. *Lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.
- J. Carretero, F. García, P. de Miguel y F. Costoya. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- K. A. Robbins y S. Robbins. *UNIX Programación Práctica*. Prentice-Hall, 1997.
- M. J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, 1985.

5.15. PRÁCTICA: LLAMADAS AL SISTEMA PARA LA GESTIÓN DE ARCHIVOS Y DIRECTORIOS EN POSIX: **mcp** Y **mils**

5.15.1. Objetivos de la práctica

Esta práctica permitirá al alumno familiarizarse con los servicios que ofrece el estándar POSIX para la gestión de archivos y directorios.

NIVEL: Intermedio.
HORAS ESTIMADAS: 6.

5.15.2. Descripción de la funcionalidad que debe desarrollar el alumno

Parte obligatoria de la práctica

Se trata de desarrollar dos programas similares a los mandatos `cp` y `ls` de UNIX. El primero se denominará `micp` y el segundo `mils`. A continuación se describen estos dos programas.

Programa micp

Este programa copiará un archivo a otro. Su sintaxis será la siguiente:

```
micp archivo_origen archivo_destino
```

El programa deberá copiar `archivo_origen` en `archivo_destino`. Si el archivo destino no existe, deberá crearse. En caso de existir, deberá truncarse su contenido. En este caso, se deberán utilizar las llamadas al sistema POSIX que sean necesarias.

La memoria deberá incluir la lista de llamadas al sistema que se han empleado para el desarrollo de este programa. Cada una de estas llamadas se describirá muy brevemente, indicando los argumentos que recibe y el resultado que devuelve.

Programa mils

Se trata de desarrollar un programa similar al mandato `ls` de UNIX, aunque con una funcionalidad reducida. El programa se denominará `mils` y recibirá el nombre de uno o varios directorios y mostrará el contenido de cada uno de ellos. Si no recibe ningún argumento, listará el contenido del directorio actual. En el caso de que alguno de los argumentos no sea un directorio, sino un archivo ordinario, el programa tratará directamente el archivo imprimiendo una línea que lo describa.

Cuando no recibe ninguna opción, muestra sólo el nombre de los archivos contenidos en el directorio restringiéndose a aquellos cuyo nombre no empieza por el carácter `.` (se podría decir que en UNIX los archivos cuyo nombre empieza por dicho carácter son *ocultos*). El programa puede además recibir dos opciones:

- Si recibe la opción `-a`, mostrará todos los archivos, es decir, los ocultos y no ocultos.
- Si se especifica la opción `-l`, además del nombre de cada archivo, se imprimirá en la misma línea algunos de sus atributos: su tipo (`R` para regular, `D` para directorio, `C` para un dispositivo de caracteres, `B` para un dispositivo de bloques y `F` para un FIFO), el número de enlaces, el tamaño y la fecha del último acceso. Para procesar este último valor se recomienda usar las funciones `gmtime` o `localtime` (se recomienda consultar el manual interactivo del sistema para aclarar su uso: `man gmtime` o `man localtime`).

Un ejemplo de ejecución del mandato `mils` -la sería el siguiente:

D	2	12456	27/1/98	.
D	6	8978	27/1/98	..
R	1	9878	27/1/98	f1.o

R	1 1024	27/1/98	f2
R	1 2039	27/1/98	f3.c
R	1 2345	27/1/98	f5.c
R	1 19288	27/1/98	f7.c
R	1 512	27/1/98	f9.c

La primera columna indica el tipo de cada archivo, la segunda el número de enlaces, la tercera el tamaño, la cuarta la fecha del último acceso. La última línea contiene el nombre del archivo.

Parte opcional de la práctica

Se plantea una parte opcional que consiste en modificar el programa `mils` anterior para que realice un listado recursivo (opción `-R`) de todo el árbol de archivos y directorios que hay por debajo de cada uno de los directorios que recibe como argumento.

Uno de los aspectos que hay que tener en cuenta a la hora de realizar el recorrido recursivo es que se debe evitar atravesar los directorios `. y ..` presentes en todo directorio, ya que de hacerlo se entraría en un bucle infinito.

Para emular dentro de lo posible el comportamiento de este mandato en UNIX, el recorrido del árbol se hará como se describe a continuación. No se realizarán las llamadas recursivas correspondientes a los subdirectorios encontrados en un determinado directorio hasta que no se haya tratado dicho directorio. Una posible forma de llevar a cabo este recorrido es utilizar una lista. Cada vez que el programa encuentre un directorio, insertará su nombre en la lista. Cuando se haya terminado de recorrer un determinado directorio, se comenzará con los directorios que se han almacenado en la lista, cada uno de los cuales se recorrerá a su vez de la forma descrita. A continuación se presenta un posible seudocódigo de dicho recorrido. Este seudocódigo asume la existencia de dos funciones, `insertar` y `extraer`. La primera introduce el nombre de un directorio en la lista y la segunda lo extrae.

```

recorrido(raíz) {
    para cada nodo que cuelgue directamente de la raíz {
        visitar el nodo;
        si (nodo es un directorio)
            insertar(nodo);
    }
    mientras (lista no sea vacía) {
        extraer(&siguiente);
        recorrido(siguiente);
    }
}

```

El alumno deberá modificar el programa `mils` siguiendo las siguientes normas:

- Se deberá recoger la opción `-R` dentro de la función `main`.
- Se deberá incorporar el recorrido descrito anteriormente, para lo cual deberá implementar la lista y las funciones `insertar` y `extraer` comentadas.

5.15.3. Código fuente de apoyo

Para facilitar la realización de la práctica se recomienda proporcionar a los alumnos un archivo comprimido que contenga el código fuente de apoyo. Dentro de un archivo como el propuesto,

existente en la página web del libro (archivo `practica-5.13.tar.gz`), se ha incluido el archivo:

- **Makefile**: archivo fuente para la herramienta **make**. Con él se consigue la recopilación automática de los archivos fuente cuando se modifique. Basta con ejecutar el mandato `make` para que el programa se compile de forma automática.

5.15.4. Recomendaciones generales

Es importante estudiar previamente el funcionamiento del mandato `cp` y `ls` de UNIX.

5.15.5. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- `memoria.txt`: Memoria de la práctica.
- `micp.c`: Código fuente del programa `micp`.
- `mils.c`: Código fuente del programa `mils`.

5.15.6. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- J. Carretero, F. García, A. Calderón y J. Fernández. *Diseño e implementación de programas en lenguaje C*. Pearson Educación, 2002.
- M. J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, 1985.

5.16. PRÁCTICA: LLAMADAS AL SISTEMA PARA LA GESTIÓN DE ARCHIVOS Y DIRECTORIOS EN WINDOWS: `micopy` Y `midir`

5.16.1. Objetivos de la práctica

Esta práctica permitirá al alumno familiarizarse con los servicios que ofrece la interfaz de Win32 para la gestión de archivos y directorios.

NIVEL: Intermedio.

HORAS ESTIMADAS: 8.

5.16.2. Descripción de la funcionalidad que debe desarrollar el alumno

Parte obligatoria de la práctica

Se trata de desarrollar dos programas similares a los mandatos `copy` y `dir` de Win32: El primero se denominará `micopy` y el segundo `midir`. A continuación se describen estos dos programas.

Programa mcopy

Este programa copiará un archivo a otro. Su sintaxis será la siguiente:

```
mcopy archivo_origen archivo_destino
```

El programa deberá copiar `archivo_origen` en `archivo_destino`. Si el archivo destino no existe, deberá crearse. En caso de existir, deberá truncarse su contenido.

La memoria deberá incluir la lista de llamadas al sistema que se han empleado para el desarrollo de este programa. Cada una de estas llamadas se describirá muy brevemente, indicando los argumentos que recibe y el resultado que devuelve.

Programa midir

Se trata de desarrollar un programa similar al mandato `dir` de Windows, aunque con una funcionalidad reducida. El programa se denominará `midir` y recibirá el nombre de uno o varios directorios y mostrará el contenido de cada uno de ellos. Si no recibe ningún argumento, listará el contenido del directorio actual. En el caso de que alguno de los argumentos no sea un directorio, sino un archivo ordinario, el programa tratará directamente el archivo imprimiendo una línea que lo describa.

Cuando no recibe ninguna opción muestra sólo el nombre de los archivos contenidos en el directorio restringiéndose a aquellos cuyo nombre no está oculto. El programa puede además recibir un parámetro opcional:

- Si recibe la opción `/w`, mostrará todos los archivos con formato de lista ancha, imprimiendo en la misma línea algunos de los atributos que se especifiquen: *D* para directorios, *A* para archivos modificados y *R* para archivos de sólo lectura.

Un ejemplo de ejecución del mandato `midir /w C:\` sería el siguiente:

ADOBEAPP	<DIR>	22/05/00	11:15p	ADOBEAPP
ARCHIV~1	<DIR>	13/07/98	1:59p	Archivos de programa
COREL	<DIR>	18/03/99	9:56p	Corel
EXCHANGE	<DIR>	02/04/99	2:37p	Exchange
GNAT	<DIR>	16/02/00	5:21p	GNAT
JDK1 3	<DIR>	22/09/00	1:26p	jdk1.3
WINDOWS	<DIR>	13/07/98	1:52p	WINDOWS
AECU	SYS	11,826	11/06/98	7:31p AECU.SYS
AUTOEXEC	BAT	902	26/09/00	2:38p AUTOEXEC.BAT
COMMAND	COM	96,312	15/05/98	8:01p COMMAND.COM
CONFIG	DOS	130	13/07/98	1:58p CONFIG.DOS
CONFIG	SYS	260	15/08/00	8:45p CONFIG.SYS
WINZIP	LOG	41,010	10/02/02	6:53p winzip.log

La primera columna indica el nombre de cada archivo, la segunda si es archivo o directorio, la tercera el tamaño, la cuarta la fecha del último acceso. La última línea contiene el nombre del archivo.

Parte opcional de la práctica

Se plantea una parte opcional que consiste en modificar el programa `midir` anterior para que realice un listado recursivo (opción `/s`) de todo el árbol de archivos y directorios que hay por debajo de cada uno de los directorios que recibe como argumento.

Uno de los aspectos que hay que tener en cuenta a la hora de realizar el recorrido recursivo es que se debe evitar atravesar los directorios . y .. presentes en todo directorio, ya que de hacerlo se entraría en un bucle infinito.

Para emular dentro de lo posible el comportamiento de este mandato en Windows, el recorrido del árbol se hará como se describe a continuación. No se realizarán las llamadas recursivas correspondientes a los subdirectorios encontrados en un determinado directorio hasta que no se haya tratado dicho directorio. Una posible forma de llevar a cabo este recorrido es utilizar una lista. Cada vez que el programa encuentre un directorio, insertará su nombre en la lista. Cuando se haya terminado de recorrer un determinado directorio, se comenzará con los directorios que se han almacenado en la lista, cada uno de los cuales se recorrerá a su vez de la forma descrita. A continuación se presenta un posible seudocódigo de dicho recorrido. Este seudocódigo asume la existencia de dos funciones, insertar y extraer. La primera introduce el nombre de un directorio en la lista y la segunda lo extrae.

```

recorrido(raíz) {
    para cada nodo que cuelgue directamente de raíz {
        visitar el nodo;
        if (nodo es un directorio)
            insertar(nodo);
    }
    mientras (lista no sea vacía) {
        extraer(&siguiente);
        recorrido(siguiente);
    }
}

```

El alumno deberá modificar el programa `midir` siguiendo las siguientes normas:

- Se deberá recoger la opción /S dentro de la función main.
- Se deberá incorporar el recorrido descrito anteriormente, para lo cual deberá implementar la lista y las funciones insertar y extraer comentadas.

5.16.3. Recomendaciones generales

Es importante estudiar previamente el funcionamiento del mandato `copy` y `dir` de Windows.

Se recomienda consultar el manual interactivo del sistema en una pantalla de MS-DOS para aclarar su uso: `copy /?` o `dir /?`.

5.16.4. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- `memoria.doc`: Memoria de la práctica.
- `micopy.c`: Código fuente del programa `micopy`.
- `midir.c`: Código fuente del programa `midir`.

5.16.5. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- J. Carretero, F. García, A. Calderón y J. Fernández. *Diseño e implementación de programas en C*. Pearson Educación, 2002.

5.17. CONCEPTOS DE DISEÑO DE SISTEMAS DE ARCHIVOS Y DIRECTORIOS

El sistema de archivos permite organizar la información dentro de los dispositivos de almacenamiento secundario en un formato inteligible para el sistema operativo. Habitualmente, cuando se instala el sistema operativo, los dispositivos de almacenamiento están vacíos. Por ello, previamente a la instalación del sistema de archivos, es necesario dividir físicamente, o lógicamente, los discos en **particiones** o **volúmenes**. Una **partición** es una *porción de un disco a la que se la dota de una identidad propia y que puede ser manipulada por el sistema operativo como una entidad lógica independiente*. Admite formato, instalación de sistemas de archivos, comprobaciones, etc. Este objeto no es utilizable directamente por la parte del sistema operativo que gestiona los archivos y directorios, que debe instalar un sistema de archivos dentro de dicha partición.

Una vez creadas las particiones, el sistema operativo debe crear las estructuras de los sistemas de archivos dentro de esas particiones. Para ello se proporcionan mandatos como `format` o `mkfs` al usuario. El siguiente mandato de UNIX crea un sistema de archivo dentro de una partición del disco duro a: `/dev/hda3`:

```
#mkfs -c /dev/hda3 -b 8192 123100
```

El tamaño del sistema de archivos, por ejemplo, 123100, se define en bloques. Un **bloque** se define como una *agrupación lógica de sectores de disco y es la unidad de transferencia mínima que usa el sistema de archivos*. Se usan para optimizar la eficiencia de la entrada/salida de los dispositivos secundarios de almacenamiento. Aunque todos los sistemas operativos proporcionan un tamaño de bloque por defecto, en UNIX los usuarios pueden definir el tamaño de bloque a usar dentro de un sistema de archivos mediante el mandato `mkfs`. Por ejemplo, `-b 8192` define un tamaño de bloque de 8 KB para `/dev/hda3`. El tamaño de bloque puede variar de un sistema de archivos a otro, pero no puede cambiar dentro del mismo sistema de archivos. En muchos casos, además de estos parámetros, se puede definir el tamaño de la **agrupación**, es decir, *el conjunto de bloques que se gestionan como una unidad lógica de gestión del almacenamiento*. El problema que introducen las agrupaciones, y los bloques grandes, es la existencia de fragmentación interna. Por ejemplo, si el tamaño medio de archivo es de 6 KB, un tamaño de bloque de 8 KB introduce una fragmentación interna media de un 25 por 100. Si el tamaño de bloque fuera de 32 KB, la fragmentación interna alcanaría casi el 80 por 100.

Todos los sistemas operativos de propósito general incluyen un componente, denominado servidor de archivos, que se encarga de gestionar todo lo referente a los sistemas de archivos.

5.17.1. Estructura del sistema de archivos

Cuando se crea un sistema de archivos en una partición de un disco, se crea una entidad lógica autocontenido con espacio para la información de carga del sistema de operativo, descripción de su estructura, descriptores de archivos, información del estado de ocupación de los bloques del

sistema de archivos y bloques de datos. La Figura 5.6 muestra las estructuras de un sistema de archivos para MS-DOS, UNIX y Windows NT, así como la disposición de sus distintos componentes, almacenado en una partición.

En UNIX, cada sistema de archivos tiene un **bloque de carga** que contiene el código que ejecuta el programa de arranque del programa almacenado en la ROM de la computadora. Cuando se arranca la máquina, el iniciador ROM lee el bloque de carga del dispositivo que almacena al sistema operativo, lo carga en memoria, salta a la primera posición del código y lo ejecuta. Este código es el que se encarga de instalar el sistema operativo en la computadora, leyéndolo desde el disco. No todos los sistemas de archivos necesitan un bloque de carga. En MS-DOS o UNIX, por ejemplo, sólo los dispositivos de *sistema*, es decir, aquellos que tienen el sistema operativo instalado, contienen un bloque de carga válido. Sin embargo, para mantener la estructura del sistema de archivos uniforme se suele incluir en todos ellos un bloque reservado para carga. El sistema operativo incluye un número, denominado **número mágico**, en dicho bloque para indicar que es un dispositivo de carga. Si se intenta cargar de un dispositivo sin número mágico, o con un valor erróneo del mismo, el monitor ROM lo detecta y genera un error.

A continuación del bloque de carga está la metainformación del sistema de archivos (superbloque, nodos-i...). La **metainformación** describe el sistema de archivos y la distribución de sus componentes. Suele estar agrupada al principio del disco y es necesaria para acceder al sistema de archivos. El primer componente de la metainformación en un sistema de archivos UNIX es el **superbloque** (el bloque de descripción del dispositivo en Windows NT), que contiene la información que describe toda la estructura del sistema de archivos. La información contenida en el superbloque indica al sistema operativo las características del sistema de archivos, dónde están los distintos elementos del mismo y cuánto ocupan. La Figura 5.7 muestra una parte del superbloque de un sistema de archivos de Linux. Como se puede ver, además de la información reseñada, se

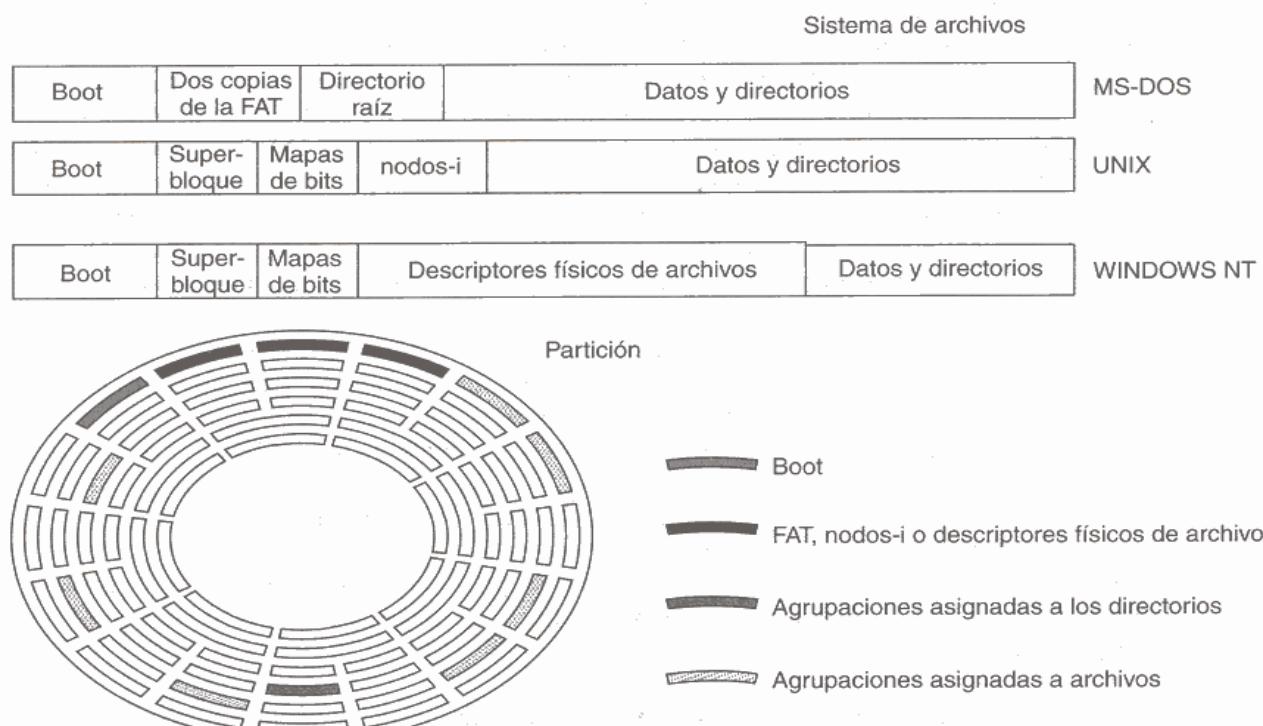


Figura 5.6. Estructura de distintos sistemas de archivos.

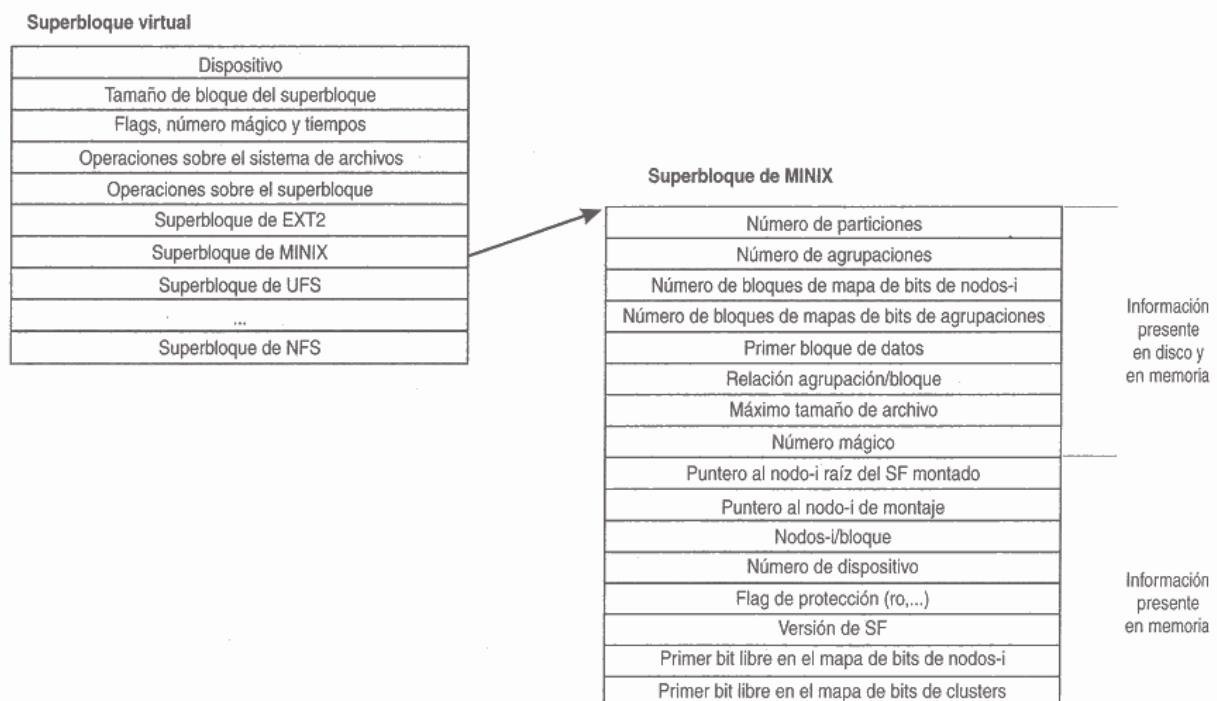


Figura 5.7. Superbloque de un sistema de archivos en MINIX.

incluye información común para todos los sistemas de archivos que proporciona el sistema operativo y una entrada para cada tipo de archivos en particular (MINIX, MS-DOS, ISO, NFS, SYSTEM V, UFS, etc.). En este caso, se incluye información de gestión tal como el tipo del dispositivo, el tamaño de bloque, número mágico, tipo de archivo, operaciones sobre el superbloque y de cuota de disco y apuntadores a los tipos de archivo que soporta. Para cada uno de ellos se incluye una estructura de datos para su superbloque particular, el máximo tamaño de archivo posible, la protección que se aplica al sistema de archivos, etc. Además, para optimizar aspectos como la búsqueda de espacio libre, se incluye información sobre la situación del primer bloque libre y del primer descriptor de archivos libre. Como ejemplo de superbloque particular, se muestra el de MINIX.

Cuando arranca la computadora y se carga el sistema operativo, el superbloque del dispositivo de carga, o sistema de archivos *raíz*, se carga en memoria en la **tabla de superbloques**. A medida que otros sistemas de archivos son incorporados a la jerarquía de directorios (en UNIX se dice que son *montados*), sus superbloques se cargan en la tabla de superbloques existente en memoria.

Tras el superbloque, el sistema de archivos incluye **información de gestión de espacio** en el disco. Esta información es necesaria por dos razones: para permitir al servidor de archivos implementar distintas políticas de asignación de espacio y para reutilizar los recursos liberados para nuevos archivos y directorios. Normalmente, los sistemas de archivos incluyen dos mapas de espacio libre:

- Información de **bloques de datos**, en la que se indica si un bloque de datos está libre o no. En caso de que el espacio de datos se administre con agrupaciones para optimizar la gestión de espacio libre, esta información se refiere a las agrupaciones.

- Información de la **descripción física de los archivos**, como nodos-i en UNIX o registros de Windows NT, en la que se indica si un descriptor de archivo está libre o no.

Después de los mapas de recursos del sistema de archivos se encuentran los **descriptores físicos de archivos**. Estos descriptores, sean nodos-i de UNIX o registros de Windows NT, tienen una estructura y tamaño variable dependiendo de cada sistema operativo. Por ejemplo, en Linux ocupa 128 bytes y en Windows NT el registro ocupa todo un bloque de 4 KB. El tamaño del área de descriptores de archivo es fácilmente calculable si se conoce el tamaño del nodo-i y el número de nodos-i disponibles en el sistema de archivos. Normalmente, cuando se crea un sistema de archivos, el sistema operativo habilita un número de descriptores de archivo proporcional al tamaño del dispositivo. Por ejemplo, en Linux se crea un nodo-i por cada dos bloques de datos. Este parámetro puede ser modificado por el usuario cuando crea un sistema de archivos, lo que en el caso de UNIX se hace con el mandato `mkfs`.

El último componente del sistema de archivos son los **bloques de datos**. Estos bloques, bien tratados de forma individual o bien en grupos, son asignados a los archivos por el servidor de archivos, que establece una correspondencia entre el bloque y el archivo a través del descriptor del archivo. Tanto si se usan bloques individuales como *agrupaciones*, el tamaño de la unidad de acceso que se usa en el sistema de archivos es uno de los factores más importantes en el rendimiento de la entrada/salida del sistema operativo. Puesto que el bloque es la mínima unidad de transferencia que maneja el sistema operativo, elegir un tamaño de bloque pequeño, por ejemplo, 512 bytes, permite aprovechar al máximo el tamaño del disco. Así, el archivo prueba de 1,2 KB ocuparía tres bloques y sólo desperdiciaría medio bloque o el 20 por 100 del espacio de disco si todos los archivos fuesen de ese tamaño. Si el bloque fuese de 32 KB, el archivo ocuparía un único bloque y desperdiciaría el 90 por 100 del bloque y del espacio de disco. Ahora bien, transferir el archivo prueba en el primer caso necesitaría la transferencia de tres bloques, lo que significa buscar cada bloque en el disco, esperar el tiempo de latencia y hacer la transferencia de datos. Con bloques de 32 KB sólo se necesitaría una operación.

5.18. PRÁCTICA: CREACIÓN DE UN SISTEMA DE ARCHIVOS: `mkfs`

5.18.1. Objetivos

Esta práctica tiene como objetivo principal que el alumno aprenda a relacionar los conceptos de sistemas de archivos vistos y las estructuras de datos asociadas a los mismos con el uso práctico de las mismas. Para ello, el alumno deberá construir un sistema de archivo muy sencillo similar al de UNIX System V o MINIX.

NIVEL: Diseño.

HORAS ESTIMADAS: 20.

5.18.2. Descripción de la funcionalidad que debe desarrollar el alumno

El alumno debe desarrollar una aplicación que permita crear sistemas de archivos en un dispositivo cualquiera (incluyendo un archivo normal).

El sistema de archivos que hay que construir tiene los siguientes elementos constructivos: superbloque, mapa de bits de nodos-i, mapa de bits de bloques, nodos-i y bloques (Figura 5.8). Se supone que el bloque 0 se deja reservado para un programa de carga.

Boott	Super-bloque	Mapa bits nodos-i	Mapa bit bloques	nodos-i	Bloques de datos
-------	--------------	-------------------	------------------	---------	------------------

Figura 5.8. Estructura del sistema de archivos.

El **superbloque** ocupa, como mínimo, un bloque y dentro de él hay una estructura de datos que contiene, al menos, lo siguiente:

- Tipo de sistema de archivos. En este caso, un entero de 4 bytes que contiene el valor 32.
- Nombre del dispositivo (máximo 32 bytes).
- Tamaño del dispositivo en bytes.
- Tamaño de bloque del sistema de archivos.
- Tamaño del dispositivo en bloques.
- Tamaño en bloques del bitmap de nodos-i.
- Tamaño en bloques del bitmap de bloques.
- Número del primer bloque de nodos-i.
- Número del primer bloque de datos.

El nodo-i es una estructura de datos que identifica a un archivo. Dentro de él hay lo siguiente:

- Número de nodo-i (entero de 4 bytes).
- Nombre del archivo (máximo 32 bytes).
- Longitud del archivo (entero de 4 bytes).
- Lista de diez bloques (enteros de 4 bytes) que puede tener asignado el archivo como máximo (tamaño máximo de archivo = 10 * tamaño de bloque).

La aplicación que genera el sistema de archivos se denominará **pr-mkfs** y su sinopsis es:

```
pr-mkfs [-i numero-nodosi] [tamaño-bloque]
           [tamaño-en-bloques] dispositivo
```

Las opciones del mandato significan lo siguiente:

- **-i numero-nodosi**: especifica el número de nodos-i del sistema de archivos. Si no se especifica nada, se crea un nodo-i por cada dos bloques.
- **tamaño-bloque**: tamaño del bloque del sistema de archivos en Kbytes. Si no se especifica nada se usan 2 KB.
- **tamaño-en-bloques**: tamaño del sistema de archivos en bloques. Si no se especifica nada, se obtiene del tamaño del archivo y se calcula en número de bloques que caben en él. El tamaño máximo debería ser de 1 MB.
- **dispositivo**: nombre del archivo sobre el que se crea el sistema de archivos. Puede ser un archivo normal.

La aplicación debe crear el archivo si no existe. Si ya existe, se indica un error.

Además, el alumno debe desarrollar una **biblioteca con cuatro funciones** para probar que el sistema de archivos es correcto:

- `int asignar-nodoi (char * nombre-archivo, pr-nodoi * nodo-i);` Busca un número de nodo-i libre en el mapa de bits de nodos-i y lo marca ocupado. Devuelve una estructura nodo-i iniciada con su número y el nombre de archivo. Devuelve -1 si hay error y 0 si fue bien.
- `int asignar-bloque (pr-nodo-i *nodo-i);` Busca un bloque libre en el mapa de bits de bloques. Cuando lo encuentra, lo marca como ocupado, devuelve su número y lo inserta dentro de la lista de bloques del nodo-i. Devuelve -1 si hay error y un número positivo, el de bloque, si fue bien.
- `int leer-bloque (int num-bloque, char *buffer, int tamanyo);` Lee del bloque especificado al *buffer* un tamaño máximo de datos. Devuelve -1 si hay error y el número de bytes leídos si fue bien.
- `int escribir-bloque (int num-bloque, char *buffer, int size);` Escribe al bloque especificado desde el *buffer* un tamaño máximo de datos. Devuelve -1 si hay error y el número de bytes escritos si fue bien.

Usando estas funciones, debe escribir un **programa de prueba** que:

- Asigne un nodo-i para un archivo con nombre `prueba-mkfs`.
- Escriba la frase: «Prueba del sistema de archivos pr-mkfs» en los bloques 50, 60, 70 y 80, poniéndolos previamente como ocupados y asignándolos al archivo.
- Lea los bloques y escriba su contenido por pantalla. En caso de que no coincida con el pedido, se indicará un error de coherencia en el sistema de archivos.

5.18.3. Código fuente de apoyo

Para facilitar la realización de la práctica se recomienda proporcionar a los alumnos un archivo comprimido que contenga el código fuente de apoyo. Dentro de un archivo como el propuesto, existente en la página web del libro (archivo `practica-5.18.tar.gz`), se han incluido los siguientes archivos:

- `Makefile`: Archivo fuente para la herramienta `make`. Con él se consigue la recopilación automática de los archivos fuente cuando se modifiquen. Basta con ejecutar el mandato `make` para que el programa se compile de forma automática.
- `pr-mkfs.c`: Archivo fuente de C, donde se incluirá el programa `pr-mkfs`.
- `pr-mkfs.h`: Archivo de cabecera con la definición parcial de las estructuras de datos.

5.18.4. Recomendaciones generales

Es importante estudiar previamente el funcionamiento del mandato `mkfs` de UNIX y MINIX (`man mkfs.minix` en Linux).

5.18.5. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- `memoria.txt`: Memoria de la práctica.
- `pr-mkfs.c`: Código fuente del programa de la utilidad `pr-mkfs`.

- pr-mkfs.h: Archivo de cabecera con la definición de las estructuras de datos.
- mkfs-lib.c: Código fuente con el código de las funciones de biblioteca.
- prueba-mkfs.c: Código fuente del programa de prueba descrito para probar mkfs.

5.18.6. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- J. Carretero, F. García, A. Calderón y J. Fernández. *Diseño e implementación de programas en C*. Pearson Educación, 2002.
- M. J. Rochkind. *Advanced UNIX Programming*. Prentice-Hall, 1985.
- W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

5.19. PRÁCTICA: DISEÑO E IMPLEMENTACIÓN DE ARCHIVOS CON BANDAS

5.19.1. Objetivos

Esta práctica tiene los siguientes objetivos principales:

- Permitir que el alumno entienda que se puede optimizar el almacenamiento en archivos haciendo uso de la concurrencia y del paralelismo.
- Enseñar al alumno a diseñar las estructuras que representan un archivo.
- Familiarizar al alumno con los mecanismos de programación de sistemas de archivos.
- NIVEL: Diseño
- HORAS ESTIMADAS: 20

5.19.2. Entorno de desarrollo de la práctica

La práctica consiste en el diseño e implementación de archivos con bandas sobre el sistema de archivos de UNIX siguiendo las directrices de este enunciado.

Para ello se proporciona al alumno una estructura de datos, denominada nodo_sfp, en la que se describe el archivo con bandas, y una descripción de cuál es la representación de un archivo con bandas.

5.19.3. Sistemas de archivos con bandas

Un sistema de archivos con bandas permite crear sistemas de archivos que ocupan varias particiones. Su estructura distribuye los bloques de datos de forma cíclica por los discos que conforman la partición lógica, repartiendo la carga de forma equitativa. Para optimizar la eficiencia del sistema de archivos, se puede definir una unidad de almacenamiento en cada banda con un tamaño mayor que el del bloque del sistema de archivos. Esta unidad, denominada unidad de distribución (*stripe unit*), es la unidad de información que se escribe de forma consecutiva en cada banda. Este valor cambia de un sistema, e incluso archivo, a otro, siendo 64 KB el valor por defecto en Windows NT.

La Figura 5.9 muestra la estructura de un sistema de archivos de bandas con cuatro particiones.

Además, este tipo de sistemas de archivos permite incrementar la fiabilidad del sistema de archivos insertando bloques de paridad con información redundante. De esa forma, si falla un dispositivo, se puede reconstruir la información mediante los bloques de los otros dispositivos y mediante la información de paridad. Además, se puede hacer que la partición sea más tolerante a fallos distribuyendo también la información de la partición del sistema.

Un archivo con bandas queda definido por dos parámetros:

- **Unidad de distribución.** Es la unidad de información que se escribe de forma consecutiva en cada banda.
- **Número de subarchivos.** Número de archivos UNIX que componen el archivo con bandas.

En el caso de la práctica, tanto la unidad de distribución como el número de subarchivos, son parámetros que se piden a la hora de crear el archivo con bandas.

Lógicamente, el archivo con bandas se ve como un único archivo por el usuario. Esto significa que el puntero de posición es único para el archivo con bandas, independientemente de que

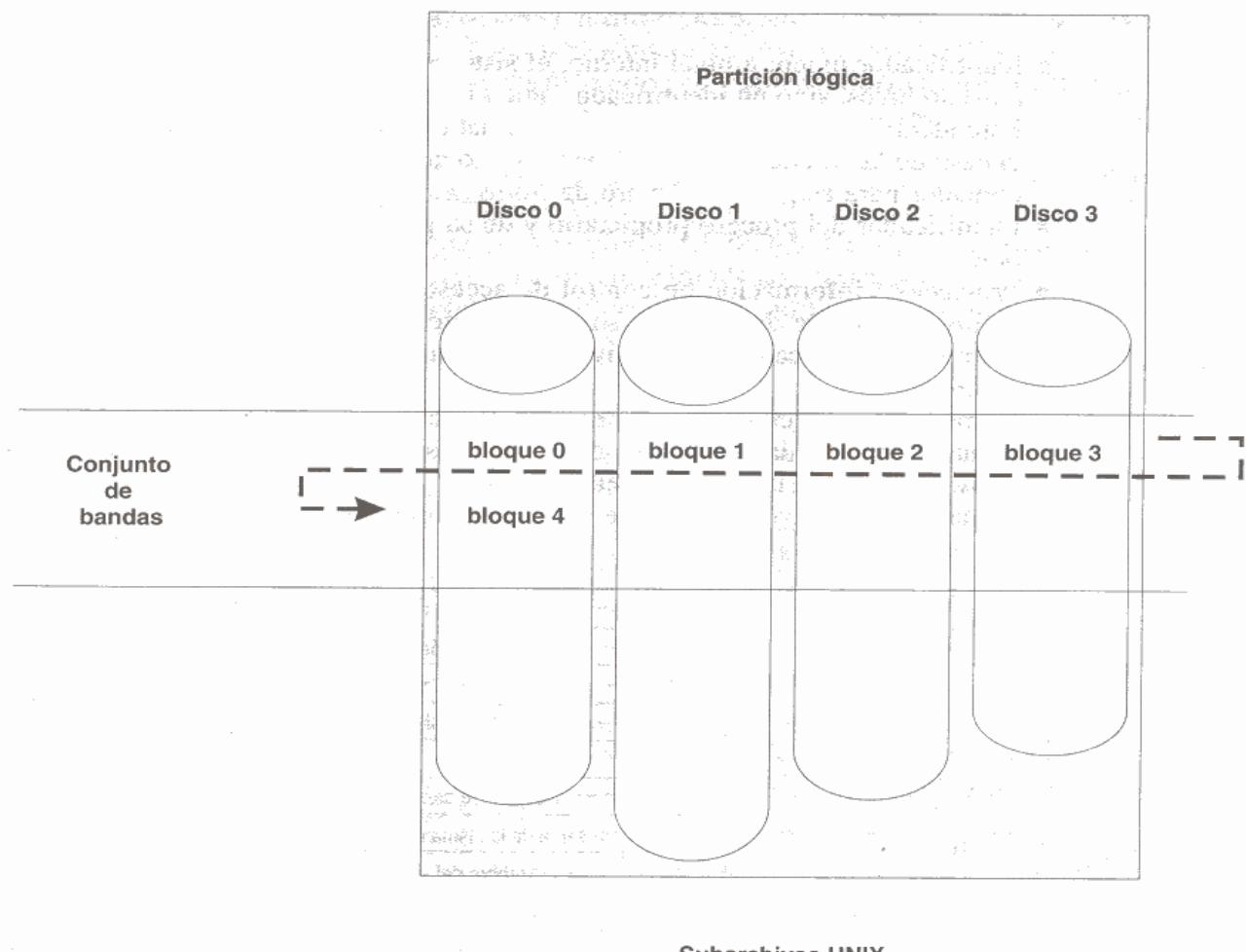


Figura 5.9. Formato de archivo con bandas.

internamente el sistema mantenga un puntero por cada subarchivo. Igualmente, cuando se pide información de estado del archivo, o se hace un `ls`, la información que aparece es global para el archivo.

La forma de operar sobre un archivo con bandas es la siguiente. Cuando se va a leer del archivo, se accede al subarchivo donde indica el puntero de posición del archivo con bandas. Sobre esa subunidad se lee lo que reste de la unidad de reparto y luego se pasa a los siguientes subarchivos. Evidentemente, si se leyera de varias subunidades se podría hacer en paralelo, pero teniendo en cuenta que la modificación del puntero de posición será la del puntero global.

nodo_sfp de un archivo con bandas

En un objeto archivo se puede almacenar información de tipo muy distinto: código fuente, programas objeto, bibliotecas, programas ejecutables, texto ASCII, agrupaciones de registros, imágenes, sonidos, etc. Desde el punto de vista del sistema operativo, un archivo se caracteriza por tener una serie de atributos. Dichos atributos varían de unos sistemas operativos a otros, pero todos ellos tienen una estructura interna que los describen. En nuestro caso, esa estructura se denominará `nodo_sfp` y se muestra en la Figura 5.10.

El `nodo_sfp` usado contiene la siguiente información:

- Identificador único: a nivel interno, el sistema operativo no usa el nombre para identificar a los archivos, sino un identificador único fijado con criterios internos al sistema operativo. Este identificador suele ser un número y habitualmente es desconocido por los usuarios. En el caso de la práctica será un número entero que nunca decrece. Por tanto, hay que tener un contador para asignar el número de `nodo_sfp`.
- Identificador del proceso propietario y de su grupo. Son el `pid` y el `gid` del proceso creador.
- Protección: información de control de acceso que define quién puede hacer qué sobre el archivo (el dueño del archivo, su creador, etc.).
- Nombre: identificador del archivo en formato comprensible para el usuario. Definido por su creador.
- Número de subarchivos que componen el archivo con bandas. Definido por su creador.
- Tamaño de la unidad de distribución, en bytes. Este dato indica la cantidad de datos consecutiva que se escribe en cada subarchivo.
- Tamaño del archivo: número de bytes en el archivo.

Número de nodo-i
Protección
Propietario
Grupo del propietario
Tamaño
Instante de creación
Instante de la última modificación
Nombre del archivo
Número de subarchivos
Veces abierto

Figura 5.10. Superbloque de un sistema de archivos con bandas.

- Tiempos de creación y de última actualización. Esta información es muy útil para gestionar, monitorizar y proteger los sistemas de archivos.
- Número de veces que el archivo ha sido abierto simultáneamente. Es el número de sesiones existentes sobre el archivo.

La biblioteca debe mantener en memoria una **tabla de nodos_sfp** abiertos, denominada `tnodos_sfp`. En esa tabla estarán los `nodo_sfp` con sesiones pendientes, pero cada `nodo_sfp` estará únicamente una vez, independientemente de las sesiones que tenga abiertas. Las entradas de la tabla tendrán un `nodo_sfp` con un número identificador positivo si están ocupadas y con identificador -1 si están libres.

Cuando se abre un archivo, hay que entrar en la tabla y buscar el primer hueco libre. Esa entrada es la ocupada y sobre ella se almacena el `nodo_sfp` leído del archivo de `nodos.sfp`, que incluye las estructuras `nodo_sfp` de los archivos con bandas existentes en el sistema. El número de `nodo_sfp` no coincide con su posición en las filas de la tabla, por lo que cuando se busque un `nodo_sfp` habrá que buscar desde el principio. El valor del `nodo_sfp` se extrae del archivo de `nodos.sfp`. Si el archivo pedido ya estaba abierto, tendrá una entrada en esta tabla. Por tanto, lo único que se debería hacer es incrementar en 1 el número de veces que está abierto.

Cuando se cierre un archivo, hay que decrementar el contador de sesiones abiertas y, si es cero, liberar la entrada de la tabla asociada al `nodo_sfp`. En el archivo `nodos.sfp` también hay que actualizar el valor del `nodo_sfp` con el número de sesiones pendientes a cero.

Importante: Todas las modificaciones del `nodo_sfp` deben reflejarse en el archivo `nodos.sfp`, es decir, se debe usar la política de escritura inmediata (*write-through*).

Tabla de descriptores de archivos (tdf_sfp)

Para desacoplar a los procesos que usen la biblioteca de los descriptores internos de los archivos, `nodo_sfp`, y para permitir que un archivo se pueda abrir varias veces sin que haya conflictos con los accesos, se usará una tabla de descriptores de archivos como la que se muestra en la Figura 5.11.

Tabla de descriptores
de archivos con bandas

fd	Nodo-i	Posición	libre
0			1
1			1
2			1
⋮			
6	98	456	1
7	98	2.348	1

Tabla de
nodos-i

Figura 5.11. Tabla de descriptores de archivos.

La tabla tiene cuatro entradas:

- Descriptor de archivo abierto. Es el que obtiene el proceso cuando ejecuta open. Este descriptor es un número entero entre 0 y MAX_FD. Teniendo en cuenta que 0, 1 y 2 están reservados para la STDIN, STDOUT y STDERR, como en UNIX.
- Número de nodo_sfp. Descriptor interno del archivo. Con este descriptor se pueden buscar los metadatos del archivo en la tabla tnodos_sfp.
- Puntero de posición del archivo. Para esa sesión del archivo abierto incluye la posición del archivo sobre la que se ejecutan las operaciones de E/S.
- Bandera de ocupado o libre. Está a 0 cuando está libre y a 1 cuando está ocupado.

Cuando se abre un archivo, hay que entrar en la tabla y buscar el primer descriptor libre. Esta entrada es la ocupada y su descriptor de archivo el devuelto. En la práctica, el número de fd coincidirá con su posición en las filas de la tabla. El valor del puntero de posición del archivo recién abierto es cero. El valor del nodo_sfp se extrae del archivo de nodos_sfp, nodos.sfp, y se incluye en la columna correspondiente. Puesto que, como se ve en la figura, varias entradas de esta tabla pueden apuntar al mismo nodo_sfp, se puede abrir un archivo repetidamente sin ningún problema.

Cuando se cierre un archivo, hay que liberar la entrada del descriptor asociado al mismo y se borran los contenidos de dicha entrada.

5.19.4. Organización de la información

Con lo descrito hasta ahora ya se puede explicar cómo y dónde debe quedar la información resultante de la ejecución de la práctica. Para simplificar la realización de la práctica, supondremos que todos los archivos con bandas se crean dentro de un subdirectorio del directorio donde se ejecuta la práctica, denominado sfp. Por tanto, todos los nombres de archivo con bandas deberían empezar por ./sfp/.

Dentro del directorio ./sfp estarán:

- El archivo nodos.sfp. Todos los accesos a este archivo se hacen usando el tipo nodo_sfp definido.
- Los subarchivos resultantes de los archivos con bandas creados. Por tanto, cuando se ejecute un mandato ls ./sfp de UNIX sobre ese directorio, se verán entradas del tipo mi_fbandas_1 a mi_fbandas_n. No existirá ninguna entrada de archivo con bandas, ya que esta entelequia existe únicamente a nivel de biblioteca y de metadatos.

Para saber si las operaciones de archivos con bandas funcionan bien, bastará en muchos casos con monitorizar el directorio ./sfp.

5.19.5. Descripción de la funcionalidad que debe desarrollar el alumno

Para construir un servidor de archivos similar a un servidor real se recomienda desarrollar esta práctica en varias etapas.

Primera etapa

En esta etapa, el alumno debe desarrollar una biblioteca de funciones para manipular los archivos con bandas. En este apartado se describe la funcionalidad de esta biblioteca, que debe incluir las siguientes funciones para manejar este tipo de archivos:

crear_sfp. Permite crear un archivo con bandas. Su prototipo es:

```
int crear_sfp (char * nombre, unsigned int permisos,  
                unsigned int num_bandas, unsigned int u_distribucion);
```

Esta función crea el archivo, pero no lo abre. Para abrirlo es necesario hacer la llamada que se describe a continuación. El efecto de `crear_sfp` es la aparición de un archivo vacío con el nombre, protecciones, número de bandas y unidad de distribución definida en la llamada. Si el archivo ya existe, se trunca, dejándolo vacío de contenido. Además, se crea el `nodo_sfp` y se incluye, relleno con los datos del archivo, en el archivo `nodos.sfp`. En caso de que ya exista el archivo, se modifica su longitud a cero y se liberan sus datos. Si se intenta crear un archivo que está abierto, la llamada debe devolver un error. El código de error es -1.

Cuando se abra un archivo con bandas aparecerán en el subdirectorio `./sfp` tantos subarchivos (archivos UNIX) como se indique en la creación. El nombre de dichos subarchivos se debe construir uniendo al nombre del archivo con bandas un entero correspondiente al número de subarchivo. Por ejemplo, para `mi_fbandas`, los subarchivos serían: `mi_fbandas_1` a `mi_fbandas_n`. Los permisos de acceso de los subarchivos deberán estar en consonancia con los solicitados para el archivo con bandas.

abrir_sfp. Permite abrir un archivo con bandas. Su prototipo es:

```
int abrir_sfp (char * nombre, unsigned int modo);
```

Esta llamada abre el archivo, y sus subarchivos asociados, y devuelve un identificador de archivo con el que trabajar. Además, lee el `nodo_sfp` en el archivo `nodos.sfp` y lo carga en la tabla de `nodos_sfp` en memoria dentro de la primera entrada de la tabla que esté libre. Los modos de acceso posibles son `FD_READ`, `FD_WRITE` y `FD_RDWR` para lectura, escritura y lectura-escritura, respectivamente.

Si el archivo ya estaba abierto, incrementa el contador de aperturas del `nodo_sfp`. Para terminar, actualiza la tabla de descriptores de archivos con una nueva entrada para este archivo. En caso de que el archivo no exista, el modo de acceso solicitado no sea adecuado o no haya espacio en alguna tabla, la llamada devuelve un -1.

cerrar_sfp. Permite cerrar un archivo con bandas. Su prototipo es:

```
int cerrar_sfp (int fd);
```

Esta llamada cierra el archivo con bandas y sus subarchivos asociados y decrementa el contador de aperturas del `nodo_sfp`. Además, se libera su entrada de la tabla de descriptores de archivos. Si el contador de aperturas del archivo es igual a cero, se elimina el `nodo_sfp` del archivo de la tabla de `nodo_sfp`. Si el identificador de archivo pedido no existe, debe devolver un error.

borrar_sfp. Borra un archivo con bandas. Su prototipo es:

```
int borrar_sfp (char * nombre);
```

Esta llamada comprueba si el archivo está abierto, devolviendo error en este caso. Si el archivo no está abierto por ningún usuario, se borra el archivo con bandas y los subarchivos que lo componen y se elimina su `nodo_sfp` del archivo `nodos.sfp`.

leer_sfp. Lee de un archivo con bandas a un *buffer* de memoria. El archivo debe estar abierto. Su prototipo es:

```
int leer_sfp (int fd, char * buffer, int tamanyo);
```

Lee del archivo *fd* a un *buffer* la cantidad de datos *tamanyo*. Además, adelanta el puntero de posición y lo modifica en la tabla de descriptores de archivo y en el *nodo_sfp*. Evidentemente, si el tamaño pedido es mayor que la unidad de distribución, hay que leer de los distintos subarchivos que contienen los datos. En caso de que el puntero de posición se encuentre en medio de una unidad de distribución, hay que leer en ese subarchivo la porción de unidad restante y luego seguir a los otros.

Esta función devuelve la longitud de los datos leídos realmente, que será siempre menor o igual al tamaño pedido. Si el archivo no está abierto debe dar un error. Si se intenta leer más allá del fin de archivo, se deben devolver los datos existentes. En caso de que no haya ninguno, se devuelve cero.

escribir_sfp. Escribe un *buffer* de memoria a un archivo con bandas. El archivo debe estar abierto. Su prototipo es:

```
int escribir_sfp (int fd, char * buffer, int tamanyo);
```

Escribe en el archivo *fd* desde un *buffer* la cantidad de datos *tamanyo*. Además, adelanta el puntero de posición, lo actualiza en la tabla de descriptores de archivos y modifica su hora de última actualización. Evidentemente, si el tamaño pedido es mayor que la unidad de distribución, hay que escribir en los distintos subarchivos que contienen los datos. En caso de que el puntero de posición se encuentre en medio de una unidad de distribución, hay que escribir en ese subarchivo la porción de unidad restante y luego seguir a los otros.

Esta función devuelve la longitud de los datos escritos realmente, que será siempre menor o igual al tamaño pedido. Si el archivo no está abierto, debe dar un error. Si se intenta escribir más allá del fin de archivo, no hay ningún problema, simplemente se añaden los datos al archivo y se modifica su longitud. En caso de que no haya espacio en el dispositivo, se devuelve error.

posicionar_sfp. Mueve el puntero de posición de un archivo con bandas. Su prototipo es:

```
int posicionar_sfp (int fd, off_t desplazamiento, int desde);
```

Esta operación se hace sobre la tabla de descriptores de archivo, donde está su puntero de posición. Con ella se puede cambiar dicho puntero mediante un *desplazamiento* con relación a *desde*. Los puntos que se pueden usar como referencia para *desde* son el principio del archivo (*SEEK_SET*), el final (*SEEK_END*) y la posición actual (*SEEK_CUR*). Si se mueve más allá del fin de archivo, la llamada debe devolver un error. El archivo debe estar abierto.

Segunda etapa

Además de la biblioteca anterior, el alumno puede hacer los mandatos *sfp_ls* y *sfp_cat* para poder consultar la información de dichos archivos.

sfp_ls. Permite ver los metadatos de los archivos con bandas que se han creado hasta el momento, es decir, todos los existentes en el directorio *./sfp/*. La información debe representar al archivo con bandas y no a sus subarchivos. La información a mostrar por cada archivo es: permisos, gid, uid, longitud, fecha de última escritura y nombre. Similar a un *ls -l* de UNIX:

```
-rw-r--r-- 1 jcarrete 937 Sep 23 16:05 Makefile.
```

El mandato admite dos formatos:

- **sfp_ls** ./.sfp/nombre-archivo, para ver información de un archivo específico, y
- **sfp_ls** ./.sfp, para ver todos los archivos del directorio.

sfp_cat. Permite ver los datos de un archivo con bandas existente en el directorio ./.sfp por la salida estándar. El formato del mandato es **sfp_cat** ./.sfp/nombre-archivo. Es similar al mandato **cat** de UNIX.

Tercera etapa

Completar las llamadas **crear_sfp** y **open_sfp** para incluir control de acceso mediante bits de protección al estilo de UNIX, además de un *uid* y un *gid* como identificador de usuario y de grupo.

Al crear el archivo se introducirá en el *nodo_sfp* el *uid* y el *gid* del usuario (serán los de UNIX, que se pueden obtener del archivo /etc/passwd).

Al abrir el archivo habrá que comprobar la identidad del usuario y de su grupo para ver si tiene los permisos adecuados. Básicamente, son: O_RDONLY para lectura, O_WRONLY para escritura y O_RDWR para lectura-escritura. Es necesario definir estas constantes.

Además, se pide implementar una llamada nueva: **estado_sfp**. Permite consultar la información de estado de un archivo. Su prototipo es:

```
int estado_sfp (char *nombre, struct nodo_sfp *estado);
```

Se entiende por información de estado la contenida en el *nodo_sfp* del archivo. El archivo no tiene por qué estar abierto. Esta operación se hace sobre el *nodo_sfp* del archivo. Devuelve un -1 en caso de error.

5.19.6. Recomendaciones generales

Se recomienda:

- Programar funciones de utilidad para ver los contenidos de las tablas de *nodo_sfp*, de descriptores de archivos y del archivo de nodos.sfp.
- Existen varias pruebas que el alumno puede realizar para ver si la funcionalidad de la práctica es correcta. A continuación se sugieren algunas:
 - Después de crear un archivo con bandas, se puede verificar que las entradas de los subarchivos están en ./.sfp con longitud cero. Después de borrarlo, dichas entradas deberán haber desaparecido. Igualmente, cuando se cree un archivo, debe aparecer un *nodo_sfp* nuevo en el archivo ./.sfp/nodos.sfp. En caso de borrado, la entrada deberá desaparecer. Si se crea un archivo que ya existe, compruebe que su longitud se trunca a cero.
 - Después de abrir un archivo, la entrada de su *nodo_sfp* deberá estar en la tabla de *nodos_sfp* de memoria. Si ya lo estaba, su contador de aperturas se habrá incrementado en 1. Después de cerrarlo, la entrada deberá desaparecer. Si el contador era mayor que 1, se decrementará en 1.
 - Se recomienda al alumno implementar dos rutinas, **read-verify** y **write-verify**, para comprobar que los datos se leen y escriben bien en los archivos. Como indica su

nombre, leen y escriben datos y después verifican que los datos manipulados están correctos en el archivo con bandas.

- Escribir en medio de la unidad de distribución y ver si los datos se distribuyen bien. No se limite a escribir siempre en bloques que coinciden con la unidad de distribución.
- Comprobar que tras las operaciones de creación y de escritura de un archivo, las fechas correspondientes del nodo-i se habrán modificado.
- Escribir un archivo con longitud y datos conocidos. Hacer una operación de posicionar y leer para ver si se está en el lugar adecuado. A continuación, intentar ir más allá del fin de archivo para ver si se obtiene error.
- Escribir un archivo con longitud y datos conocidos y ejecutar un mandato `sfp_cat` para ver si se obtienen los datos escritos previamente.

5.19.7. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- `memoria.txt`: Memoria de la práctica.
- `Makefile`: Archivo de compilación.
- `nodo-sfp.c`: Gestión de nodo-sfp.
- `tdf-sfp.c`: Gestión de tabla de archivos.
- `sfp.c`: Biblioteca de funciones para archivos paralelos.
- `sfp-ls-c`, `sfp-cat.c`: Funciones de utilidad para mandatos.

5.19.8. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*, McGraw-Hill, 2001.
- J. Carretero, F. García, A. Calderón y J. Fernández. *Diseño e implementación de programas en C*. Pearson Educación, 2002.
- A. Silberchatz y Galvin. *Operating Systems Concepts*. 5.^a edición. Addison-Wesley, 1999.
- A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1991.
- W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.
- P. S. Wang. *An Introduction to Berkeley Unix*. Wadsworth. International Student Edition, 1988.
- M. Solomon. *Inside Windows NT*. Microsoft Press, 1998.

6

Introducción a los sistemas distribuidos

En este capítulo se presentan prácticas relacionadas con sistemas distribuidos. Este capítulo tiene dos objetivos básicos: en primer lugar, se muestran al lector los servicios y mecanismos básicos que permiten construir aplicaciones distribuidas; en segundo lugar, las prácticas propuestas intentan que el lector entienda cómo se construyen algunos de los servicios que ofrece un sistema operativo distribuido, como son el sistema de archivos distribuido y los servicios de sincronización para aplicaciones distribuidas.

6.1. CONCEPTOS BÁSICOS

Un sistema distribuido se puede definir según dos puntos de vista:

- Desde un punto de vista **físico**, un sistema distribuido es un conjunto de procesadores, posiblemente heterogéneos, sin memoria ni reloj común que se encuentran conectados a través de una red de interconexión. Esta estructura se presenta en la Figura 6.1.
- Desde un punto de vista **lógico**, un sistema distribuido es un conjunto de procesos que ejecutan en una o más computadoras y que colaboran y se comunican entre ellos mediante el intercambio de mensajes.

Este tipo de sistemas ha tenido un enorme desarrollo durante la década de los noventa debido sobre todo a la gran difusión de Internet. Las principales **características** que presentan los sistemas distribuidos se enumeran a continuación:

- **Posibilidad de compartir recursos.** Un sistema distribuido permite compartir diferentes recursos *hardware*, *software* o datos. Un ejemplo típico de recurso compartido en un entorno distribuido es una impresora conectada a una red, que puede ser utilizada por los usuarios de una organización.
- **Capacidad de crecimiento.** Es relativamente fácil que un sistema distribuido crezca para adaptarse a las nuevas demandas de servicio, basta con conectar más computadoras a la red.
- **Alto rendimiento.** El empleo de múltiples procesadores hace posible la construcción de un sistema de altas prestaciones que permite ofrecer servicio concurrente a múltiples usuarios y del que pueden beneficiarse las aplicaciones paralelas.
- **Fiabilidad y disponibilidad.** El hecho de disponer de múltiples procesadores conectados a una red ofrece la posibilidad de tener una alta disponibilidad. Así, por ejemplo, considere un servidor de archivos. En un sistema distribuido es posible replicar la información de ese servidor en otro, de tal forma que se puede tener un servidor de archivos replicado en dos nodos. En caso de que uno de los nodos falle, el otro podrá seguir ofreciendo la funcionalidad de servidor de archivos. Esto es algo imposible de conseguir en un entorno centralizado.

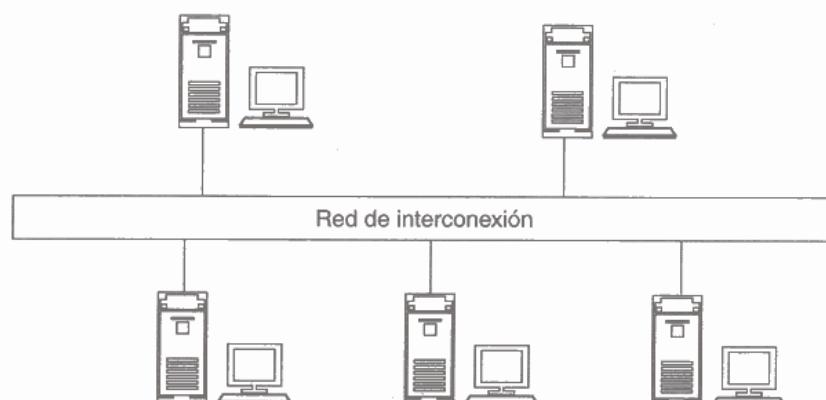


Figura 6.1. Arquitectura de un sistema distribuido típico.

6.2. PROTOCOLOS DE COMUNICACIÓN

Uno de los aspectos clave de todo sistema distribuido es la comunicación, y para que ésta tenga lugar se necesita la existencia de protocolos de comunicación. Un **protocolo de comunicación** define un conjunto de reglas e instrucciones que gobiernan el intercambio de mensajes entre procesos. La definición de un protocolo incluye la especificación de la secuencia de mensaje que deben intercambiarse los procesos y la especificación del formato de los mensajes. Los protocolos de comunicación suelen estructurarse como una **pila de protocolos**, es decir, un conjunto de protocolos apilados, cada uno de los cuales realiza tareas bien diferenciadas, entre las que se encuentran el encapsulado, el control de flujo, de errores, etc. El protocolo de comunicación más utilizado en los sistemas distribuidos actuales es el protocolo TCP/IP. El protocolo TCP/IP, cuya arquitectura se muestra en la Figura 6.2, está formado por cuatro niveles:

- **Nivel de interfaz de red.** Se encarga de manejar el acceso al *hardware* de red y es específico de cada red en concreto. También ofrece funciones de nivel de enlace de datos, encargándose de la transferencia de tramas entre computadoras directamente conectadas entre sí.
- **Nivel de Internet.** Este nivel, denominado también nivel IP, se encarga de transferir los paquetes de este nivel, denominados datagramas, entre diferentes computadoras. Este nivel es similar al nivel de red del modelo OSI y ofrece funciones de encaminamiento.
- **Nivel de transporte.** Este nivel se encarga de la transmisión de mensajes entre los procesos. Dentro de este nivel, los usuarios pueden utilizar dos tipos de protocolos:
 - Protocolo UDP. Protocolo basado en datagramas de tamaño máximo de 64 KB. Es un protocolo no orientado a conexión que no ofrece fiabilidad ni ordenación en la entrega de los diferentes datagramas.
 - Protocolo TCP. Protocolo orientado a conexión que garantiza que los datos se entregan en el orden en el que se envían. Una conexión TCP constituye un flujo de bytes y asegura la entrega ordenada y fiable de los datos enviados.

Las conexiones dentro del nivel de transporte se definen mediante el tipo de protocolo (TCP o UDP), la dirección IP de la computadora y un número de puerto que identifica a un proceso concreto.

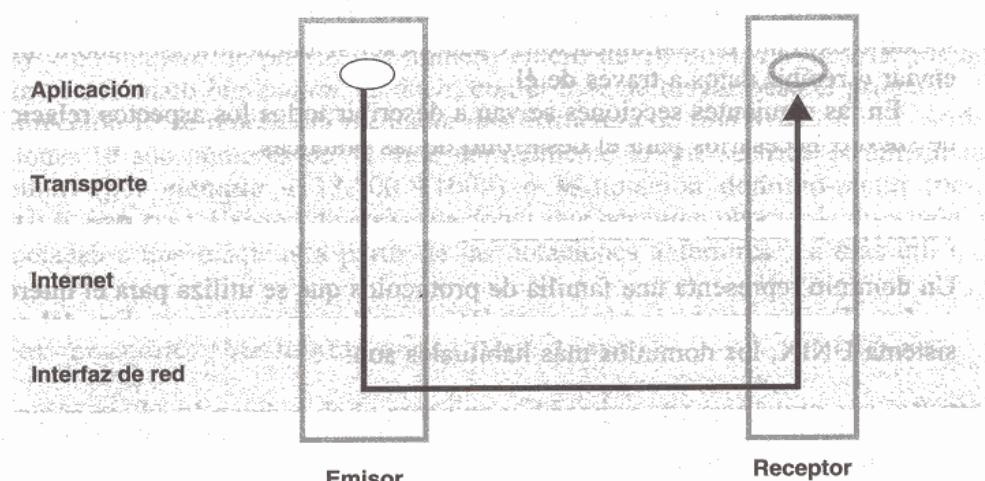


Figura 6.2. Niveles del protocolo TCP/IP.

- **Nivel de aplicación.** Dentro de este protocolo se encuentra cualquier aplicación que utilice el nivel de transporte, como, por ejemplo, la transferencia de archivos (*ftp*) o el protocolo *http* utilizado en los servidores web.

6.3. COMUNICACIÓN DE PROCESOS EN SISTEMAS DISTRIBUIDOS

La comunicación de procesos es una parte fundamental y básica en cualquier sistema distribuido. Existen dos enfoques de comunicación, ambos basados en el paso de mensajes:

- **Mecanismos de bajo nivel.** En este tipo de esquemas se ofrecen servicios en los que el programador debe encargarse de establecer los protocolos de comunicación, la forma de representación de los datos, etc. Ejemplos de este tipo de mecanismos son las colas de mensajes de POSIX, los Mailslots de Win32 y los *sockets*.
- **Mecanismos de alto nivel.** En este tipo de enfoques se ofrecen abstracciones que facilitan la programación y en las que el programador no debe preocuparse de los aspectos de bajo nivel relacionados con el paso de mensajes. En este tipo de esquemas se incluyen las llamadas a procedimientos remotos y la invocación de métodos de remotos como la que ofrece CORBA o RMI de Java.

En este capítulo se describirán los modelos utilizados para la realización de las prácticas propuestas en este capítulo. Estos modelos, muy utilizados para el desarrollo de aplicaciones distribuidas, son los *sockets* y las llamadas a procedimientos remotos (RPC).

6.3.1. *Sockets*

Un *socket* es una abstracción que representa un extremo en la comunicación bidireccional entre dos procesos. Ofrece una interfaz para acceder a los servicios de red en el nivel de transporte de los protocolos TCP/IP. Actualmente, la interfaz de *sockets* está siendo estandarizada dentro de POSIX y está disponible en prácticamente todos los sistemas operativos.

Utilizando esta interfaz, dos procesos que desean comunicarse crean cada uno de ellos un *socket* o extremo de comunicación. Cada *socket* se encuentra asociado a una dirección y permite enviar o recibir datos a través de él.

En las siguientes secciones se van a describir todos los aspectos relacionados con la interfaz de *sockets* necesarios para el desarrollo de las prácticas.

6.3.1.1. Dominios de comunicación

Un dominio representa una familia de protocolos que se utiliza para el intercambio de datos entre *sockets*. Es importante destacar que sólo se pueden comunicar *sockets* del mismo dominio. En un sistema UNIX, los dominios más habituales son:

- Dominio UNIX (PF_UNIX), que se utiliza para la comunicación de procesos dentro de la misma computadora.
- Dominio Internet (PF_INET), que se emplea para la comunicación de procesos que ejecutan en computadoras conectadas por medio de los protocolos TCP/IP.

6.3.1.2. Tipos de *sockets*

Existen dos tipos básicos de *sockets* que determinan el estilo de comunicación empleado. Estos dos tipos son:

- *Sockets Stream* (SOCK_STREAM). Con este tipo de *sockets* la comunicación es orientada a conexión. El intercambio de datos utilizando *sockets* de este tipo es fiable y además se asegura el orden en la entrega de los mensajes. El canal de comunicaciones puede verse como un flujo de bytes en el que no existe separación entre los distintos mensajes. Cuando se emplea el dominio Internet, este tipo de *sockets* se corresponde con el protocolo de transporte orientado a conexión, TCP.
- *Sockets* de tipo *datagrama* (SOCK_DGRAM). Este tipo de *sockets* se corresponde con una comunicación no orientada a conexión. Los mensajes en este caso se denominan datagramas y tienen un tamaño máximo de 64 KB. No se asegura fiabilidad, los datagramas se pueden perder y tampoco se asegura la entrega ordenada de los mismos. En este caso, sí existe separación entre cada uno de los distintos datagramas. Cuando se emplea el dominio Internet, los *sockets* de este tipo permiten acceder a los servicios del protocolo de transporte UDP.

6.3.1.3. Direcciones de *sockets*

Cada *socket* debe tener asignada una dirección única. Se usan para asignar una dirección a un *socket* local y para especificar la dirección de un *socket* remoto con el que se desea comunicar. Cada dominio utiliza una dirección específica. Existen, por tanto, dos familias de direcciones:

- AF_UNIX para *sockets* del dominio UNIX. Utilizan como dirección el nombre de un archivo local, lo que representa una dirección única.
- AF_INET para *sockets* del dominio Internet. Utilizan la dirección IP de una máquina y un número de puerto dentro de la máquina. El par (dirección IP, puerto) representa también una dirección única en Internet.

La dirección genérica de un *socket* se describe utilizando una estructura de tipo `struct sockaddr`. Cuando se trata de *sockets* del dominio Internet, se usa la estructura `struct sockaddr_in`. Una estructura de este tipo incluye, entre otros, la dirección IP (un número entero de 32 bits) y un número de puerto (un número entero de 16 bits). En TCP/IP, los números se representan con formato *big-endian*, es decir, con el byte de mayor peso el primero.

Una dirección IP se representa mediante una estructura de tipo `struct in_addr`. Aunque las direcciones IP son números de 32 bits, normalmente lo que se hace es utilizar la notación decimal-punto (por ejemplo «138.100.8.100») o la notación dominio-punto (por ejemplo, «laurel.datsi.fi.upm.es»). Existen diversas funciones que permiten obtener la dirección o direcciones IP asociadas a una máquina a partir de las notaciones anteriores. La más útil es, quizás, la siguiente:

```
struct hostent *gethostbyname(char *name);
```

La función recibe el nombre de la máquina y devuelve una estructura con información relativa a dicha máquina. Esta estructura es la siguiente:

```
struct hostent {
    char h_name;           /* nombre de la máquina */
```

```

    char **h_aliases;           /* lista de alias */
    int h_addrtype;            /* tipo de dirección */
    int h_length;               /* longitud de las direcciones */
    char **h_addr_list;         /* lista de todas las direcciones */
};

}

```

Como se dijo anteriormente, en TCP/IP los números se emplean con formato *big-endian*. Sin embargo, este formato no se emplea en todas las computadoras. En este caso es necesario utilizar una serie de rutinas que permiten traducir números entre el formato que utiliza TCP/IP y el empleado por la propia computadora. Estas rutinas son:

```

u_long htonl(u_long hostlong);
u_short htons(u_short hostshort);
u_long ntohl(u_long netlong);
u_short ntohs(u_long netshort);

```

La primera traduce un entero de 32 bits representado en el formato de la propia computadora al formato de red (el empleado por TCP/IP). La segunda traduce un número de 16 bits representado en el formato de la computadora al formato de red. Las dos últimas realizan el trabajo inverso.

6.3.1.4. Operaciones con *sockets*

A continuación se describen otras operaciones típicas relacionadas con los *sockets*.

```
int socket(int dominio, int tipo, int protocolo);
```

Permite crear un *socket*. El primer argumento especifica el dominio (PF_UNIX o PF_INET). El segundo indica el tipo de *socket* creado: *socket stream* (SOCK_STREAM) o *socket* de tipo datagrama (SOCK_DGRAM). El tercer argumento permite especificar el protocolo a emplear. Por lo general, se utiliza para este argumento el valor 0, aunque en el archivo /etc/protocols de las máquinas UNIX se pueden encontrar diversos protocolos. La llamada *socket* devuelve un descriptor de archivos que se emplea en el resto de operaciones (especialmente en las de envío y recepción de datos) de forma similar a lo que ocurre con los archivos en POSIX. En caso de error se devuelve -1. Es importante recordar que el *socket* creado con esta llamada no tiene asociada ninguna dirección, por lo que no se puede utilizar tal cual para la transferencia de datos.

```
int bind(int socket, struct sockaddr *dir, int long);
```

Permite asignar una dirección a un *socket*. El primer argumento es el descriptor de *socket* devuelto en la llamada *socket*. El segundo argumento especifica la dirección que se va asignar al *socket*. El tercer argumento especifica la longitud en bytes que ocupa la dirección.

La dirección que se utiliza en el segundo argumento depende del dominio del *socket* creado. Si el dominio del *socket* es PF_UNIX, la estructura que se emplea es del tipo struct *sockaddr_un*. Si el dominio es PF_INET, la estructura es de tipo struct *sockaddr_in*, que incluirá, entre otras cosas, una dirección IP y un número de puerto. La llamada devuelve 0 en caso de éxito o -1 si hubo algún error.

```
int connect(int socket, struct sockaddr *dir, int long);
```

Llamada de solicitud de conexión. La invoca el cliente para establecer una conexión con un proceso servidor cuando se utilizan *sockets* de tipo *stream*. El primer argumento representa el

descriptor de *socket* devuelto en la llamada *socket*. El segundo representa la dirección del *socket* del proceso servidor y que deberá haber sido rellenada con la dirección IP y el número de puerto correspondiente. El tercer argumento especifica en bytes la longitud de la dirección utilizada en el segundo argumento. La llamada devuelve 0 si se ejecutó con éxito o -1 en caso de error.

```
int listen (int socket, int backlog);
```

Esta llamada se utiliza para aceptar una nueva conexión. Se utiliza en el servidor cuando se utilizan *sockets* de tipo *stream*. El primer argumento es el descriptor de *socket* y el segundo representa el número máximo de peticiones pendientes de aceptar que se encolarán. Su valor típico y máximo en algunos casos es 5.

```
int accept(int socket, struct sockaddr *dir, int *long);
```

Una vez que el *socket* está preparado para aceptar conexión, esta llamada permite al servidor aceptar peticiones por parte de los clientes (los clientes utilizarán *connect*). El primer argumento representa el descriptor de *socket*. En el segundo se almacena la dirección del *socket* del proceso cliente que realiza la conexión. En el tercer argumento se almacena la longitud en bytes de la dirección anterior.

Esta llamada bloquea al proceso servidor hasta que un cliente realice una conexión. Cuando se produce la conexión, el servidor obtiene en el segundo argumento la dirección del *socket* del cliente. Además, la llamada devuelve un nuevo descriptor de *socket* que será el que utilice el servidor para recibir o enviar datos. Nótese que después de la conexión permanecen activos dos descriptores de *socket* en el servidor: el original, que se utilizará para aceptar nuevas conexiones, y el devuelto por la llamada, que se usará para la transferencia de datos con el proceso que ha establecido la conexión.

```
int write(int socket, char *mensaje, int longitud);
int send(int socket, char *mensaje, int longitud, int flags);
```

Permiten enviar datos a través de un *socket*. El primer argumento representa el descriptor de *socket*; el segundo, el mensaje que se quiere enviar, y el tercero, la longitud de ese mensaje en bytes. El argumento *flags* en la llamada *send* se utiliza en aspectos avanzados, que no serán cubiertos en este libro. Su valor típico es 0. Las llamadas anteriores devuelven el número de bytes realmente transferidos.

```
int read(int socket, char *mensaje, int longitud);
int recv(int socket, char *mensaje, int longitud, int flags);
```

Estas llamadas bloquean al proceso esperando la recepción de un mensaje. La llamada devuelve el número de bytes que se han transferido o -1 en caso de error.

En las llamadas de envío y recepción es importante comprobar siempre el valor que devuelven las llamadas de envío y recepción de datos, ya que el valor devuelto puede no coincidir con el campo *longitud*, que indica la cantidad de datos que se quieren transferir. En caso de que ambos valores no coincidan, debería volverse a enviar o recibir los datos que aún no se han transferido. A continuación se presenta una función que se puede utilizar para enviar un bloque de datos haciendo los reintentos que sean necesarios. La función devuelve 0 en caso de éxito o -1 si hubo algún error y no se pudieron enviar todos los datos.

```
int enviar(int socket, char *mensaje, int longitud)
{
```

```

int r;
int long = longitud;

do {
    r = write(socket, mensaje, long);
    long = long - r;
    mensaje = mensaje + r;
} while ((long > 0) && (r >= 0));
if (r < 0)
    return (-1); /* la última llamada falló */
else
    return (0);
}

int sendto(int socket, char *mensaje, int long, int flags, struct
sockaddr *dir, int long);

```

Se utiliza para enviar un datagrama cuando se emplean *sockets* de tipo datagrama. El argumento *dir* representa la dirección del *socket* remoto al que se quieren enviar los datos y *long* la longitud en bytes que ocupa.

```

int recvfrom(int socket, char *mensaje, int long, int flags, struct
sockaddr *dir, int *long);

```

Se emplea para recibir datos a través de un *socket* de tipo datagrama. En este caso, en *dir* se almacena la dirección del *socket* del que se han recibido los datos y en *long* la longitud que ocupa en bytes.

```

int close(int socket);

```

La llamada *close* se utiliza para cerrar un *socket*. En el caso de que el *socket* sea de tipo *stream*, la llamada cierra, además, la conexión en ambos sentidos.

6.3.1.5. Utilización de *sockets* de tipo *stream*

Cuando se utilizan *sockets* de tipo *stream* es necesario llevar a cabo todos los pasos que se pueden ver en la Figura 6.3.

El proceso servidor de la aplicación (véase la Figura 6.3) realiza los siguientes pasos:

1. Crea un *socket* de tipo *stream*.
2. Le asigna una dirección utilizando *bind*. Cuando se utiliza un *socket* de dominio *PF_INET*, la estructura que define la dirección tiene tres campos básicos que hay que rellenar:
 - Familia de dirección (*sin_family*). En este caso será *AF_INET*.
 - Número de puerto asignado (*sin_port*). El rango de puertos disponibles es 0,65535, estando reservados los 1.024 primeros para el sistema.
 - Estructura que contiene, entre otras cosas, la dirección IP de una máquina (*struct in_addr sin_addr*).
3. Se prepara para recibir conexiones utilizando *listen*.
4. Se bloquea esperando la recepción de conexiones por parte de los clientes utilizando la llamada *accept*. Cuando se desbloquea, la llamada anterior devuelve un nuevo descriptor de *socket* que será el que se utilice en la transferencia

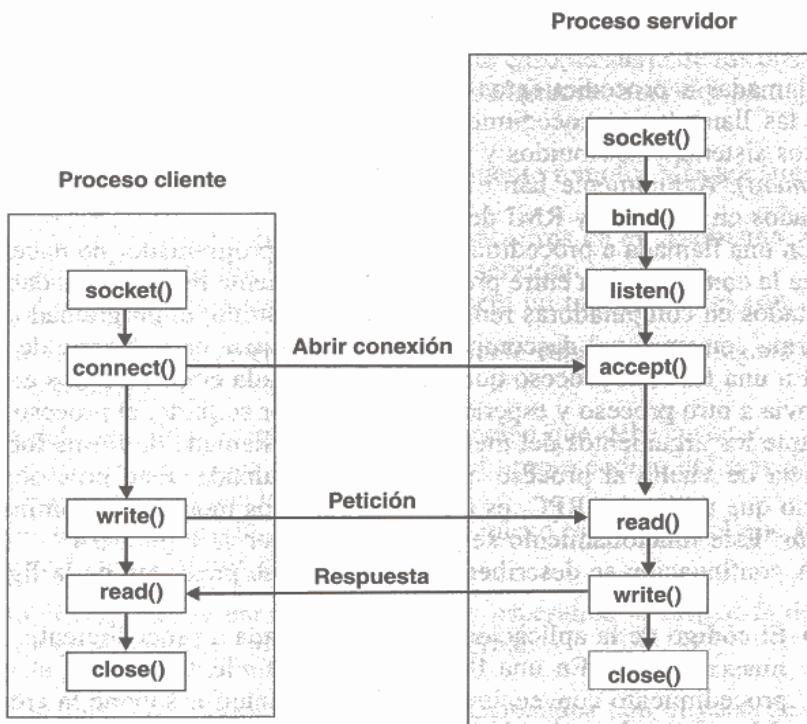


Figura 6.3. Escenario típico con «sockets» de tipo «stream».

5. El proceso recibe un mensaje del proceso cliente. Para ello se utiliza la llamada `read`.
6. El proceso servidor ejecuta el trabajo necesario, en este caso realiza la suma.
7. Devuelve el resultado al cliente utilizando `write`.
8. Cierra el descriptor de `socket` devuelto en la llamada `accept`, ya que éste no volverá a utilizarse. Al mismo tiempo se cierra la conexión con el cliente.
9. Vuelve al paso 4 para esperar nuevas peticiones.

Los pasos que debe realizar el cliente (véase la Figura 6.3) son más sencillos. El cliente debe:

1. Crear un `socket` de tipo `stream`.
2. Establecer una conexión con el proceso servidor utilizando `connect`.
3. Enviar los números que se quieren sumar utilizando la llamada `write`.
4. Esperar el resultado utilizando `read`.
5. Cerrar la conexión con `close`.

El cliente obtiene la dirección IP del servidor utilizando la función `gethostbyname`. Esta función devuelve información sobre el servidor. Para llenar la dirección del servidor basta con acceder a los campos correspondientes de la estructura devuelta por la función anterior.

```
hp = gethostbyname ("laurel.datsi.fi.upm.es");
memcpy (&(server_addr.sin_addr), hp->h_addr, hp->h_length)
```

6.3.2. Llamadas a procedimientos remotos

Las llamadas a procedimientos remotos (RPC, *Remote Procedure Call*) representan un híbrido entre las llamadas a procedimientos y el paso de mensajes. Las RPC constituyen el núcleo de muchos sistemas distribuidos y llegaron a su culminación con DCE (*Distributed Computing Environment*). Actualmente han evolucionado hacia la invocación de métodos remotos como los utilizados en CORBA y RMI de Java.

En una llamada a procedimiento remoto, el programador no necesita preocuparse de cómo se realiza la comunicación entre procesos. Simplemente realiza llamadas a procedimientos que serán ejecutados en computadoras remotas. En este sentido, el programador desarrolla sus aplicaciones de forma convencional descomponiendo su *software* en una serie de procedimientos bien definidos. En una RPC, el proceso que realiza la llamada empaqueta los argumentos en un mensaje, se los envía a otro proceso y espera el resultado. Por su parte, el proceso que ejecuta el procedimiento extrae los argumentos del mensaje, realiza la llamada de forma local, obtiene el resultado y se lo envía de vuelta al proceso que realizó la llamada. Este proceso, totalmente transparente al usuario que utiliza las RPC, es realizado por unos módulos denominados **resguardos**, **suplentes** o **stubs**. Este funcionamiento se puede apreciar en la Figura 6.4.

A continuación se describen los pasos que se presentan en la figura anterior.

- El código de la aplicación hace una llamada a procedimiento. Por ejemplo, puede invocar sumar (5,2). En una llamada a procedimiento remoto, al igual que en una llamada a procedimiento convencional, la llamada anterior supone la creación en la pila del proceso del registro de activación correspondiente. Este registro incluye, entre otras cosas, los parámetros del procedimiento (5 y 2 en este caso) y la dirección de retorno.
- El procedimiento invocado anteriormente, que forma parte del código del propio proceso, se encuentra en un módulo del programa cliente que se denomina resguardo y que como se

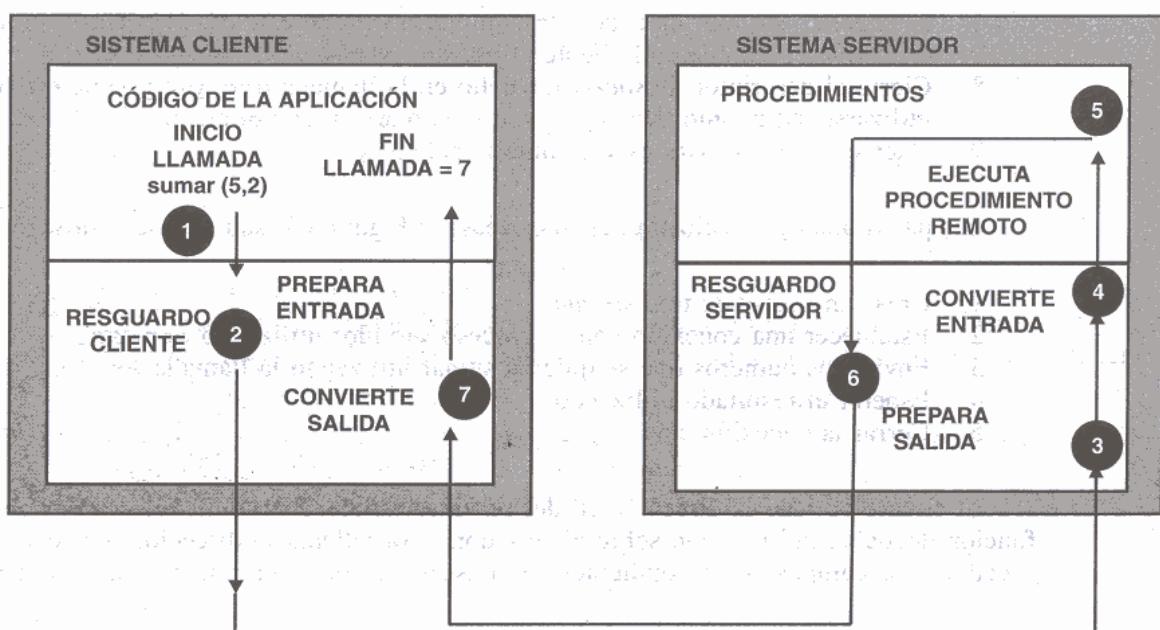


Figura 6.4. Llamadas y mensajes en una RPC.

verá más adelante se obtiene de forma automática. Este procedimiento extrae de la pila los argumentos, pero no realiza la operación; en este caso, la suma de los dos números pasados en la pila. Lo que hace es enviar los argumentos al proceso servidor para que sea éste el que realiza la llamada. Para ello:

- Localiza al servidor que ejecuta el procedimiento remoto. Más adelante se verá cómo se realiza este proceso.
- Construye un mensaje y empaqueta los parámetros en él. Asimismo, indica en el mensaje el procedimiento que debe realizar el servidor.
- Envía el mensaje al proceso servidor y espera un mensaje de él con la respuesta.
- El proceso servidor, que se encuentra en un bucle esperando la llegada de peticiones por parte de los clientes:
 - Extrae del mensaje los argumentos y la función que debe invocar.
 - Una vez extraídos, invoca a una función del propio código del servidor para realizar la llamada y obtiene el resultado.
 - Una vez tomado el resultado, construye un nuevo mensaje en el que introduce el resultado y se lo envía al proceso cliente.
- El proceso cliente que se encuentra bloqueado esperando la respuesta del servidor extrae del mensaje el resultado y simplemente lo retorna.

Con los pasos anteriores, todos los aspectos relacionados con el paso de mensajes quedan ocultos al programador, siendo el *software* del paquete de RPC utilizado el encargado de realizarlo.

Un aspecto muy importante de las llamadas a procedimientos remotos es el **lenguaje de definición de interfaces (IDL)**, que permite especificar las interfaces de los procedimientos a ejecutar en el servidor. A partir de un programa escrito utilizando un lenguaje de definición de interfaces, el **compilador de interfaces** se encarga de obtener los suplementos del cliente y del servidor. En la siguiente sección se muestra un ejemplo de uso de un paquete de RPC que será utilizado en las prácticas que se describen al final del capítulo.

6.3.3. Introducción a las RPC de SUN

La utilización de RPC permite a los programadores el desarrollo de aplicaciones distribuidas que utilizan el modelo cliente-servidor. El modelo de RPC desarrollado por *Sun Microsystems* es uno de los más extendidos y se diseñó inicialmente para el desarrollo del sistema de archivos en red NFS (*Network File System*).

Las RPC de Sun constan de unos formatos de datos descritos utilizando el formato de representación de datos XDR (*eXternal Data Representation*) desarrollado también por Sun. Este formato describe un lenguaje de descripción de datos y estandariza una sintaxis de transferencia. Dentro del lenguaje de descripción de datos, XDR dispone de una serie de tipos elementales (`int`, `bool`, `string`, etc.) y una serie de constructores para declarar tipos de datos más complejos (vectores, estructuras, uniones). Casi todos estos tipos y constructores tienen su equivalente en los lenguajes de programación más habituales.

En estas RPC se utiliza el término *programa* para hacer referencia a un servicio. Un programa consta de una serie de procedimientos que pueden ser utilizados por los clientes. Cada programa puede tener varias versiones. Los programas, versiones y procedimientos se identifican mediante números enteros. De acuerdo a esto, cada procedimiento queda perfectamente identificado por la terna `<programa, versión, procedimiento>`.

La especificación de una aplicación consiste en la definición de un conjunto de procedimientos dentro de un programa. De manera más formal, la definición de un protocolo de aplicación debe seguir la siguiente sintaxis:

```

programa-def:
    "program" identifier "{"
        version-def
        version-def *
    }" "=" contant ";"

version-def:
    "version" identifier "{"
        procedure-def
        procedure-def *
    }" "=" constant ";"

```

6.3.3.1. Principales tipos de datos en XDR

En esta sección se van a describir algunos de los tipos de datos definidos en XDR.

Enteros con signo:

Declaración: int a;
Equivalente en C: int a;

Enteros sin signo:

Declaración: unsigned a;
Equivalente en C: unsigned a;

Valores lógicos:

Declaración: bool a;
Equivalente en C:
`enum bool_t {TRUE = 1, FALSE=0};
typedef enum bool_t bool_t;
bool_t a;`

Números en coma flotante:

Declaración: float a;
Equivalente en C: float c;

Cadenas de bytes de longitud fija:

Declaración: opaque a[20];
Equivalente en C: char a[20];

Cadenas de bytes de longitud variable:

Declaración: opaque a<37>;
opaque b<>;
Equivalente en C:
`struct {
 int a_len;
 char *a_val;
} a;`

Cadenas de caracteres:

Declaración: string a<37>;
 string b<>;
 Equivalente en C: char *a;
 char *b;

Vectores de tamaño fijo:

Declaración: int a[12];
 Equivalente en C: int a[12];

Vectores de tamaño variable:

Declaración: int a<12>;
 float b<>;
 Equivalente en C: struct {
 int a_len;
 int *a_val;
 } a;
 struct {
 int b_len;
 float *b_val;
 } b;

Estructuras:

Declaración: struct t {
 int c1;
 string c2<20>;
 };
 t a;
 struct t {
 int c1;
 string *c2;
 };
 typedef struct t t;
 t a;

Constantes:

Declaración: const MAX = 12;
 Equivalente en C: #define MAX 12

6.3.3.2. Programa rpcgen

El programa rpcgen genera a partir de un archivo con la especificación de una aplicación (por defecto, el archivo tiene extensión .x) los siguientes archivos:

1. Archivo de cabecera con estructuras de datos en C equivalentes a las definidas en el archivo de especificación. Este archivo deberá incluirse en el código del cliente y el código del servidor.
2. Archivo con funciones de transformación de datos XDR, una por cada tipo de datos definido.

3. Archivo con el suplemento del cliente.
4. Archivo con el suplemento del servidor.

Tradicionalmente, una llamada a procedimiento remoto acepta un único parámetro y devuelve un único resultado. Para poder pasar diferentes valores a una llamada a procedimiento remoto es necesario agrupar éstos en una estructura y pasar esta estructura como parámetro. Si se quieren recoger varios valores de vuelta, también es necesario agrupar éstos en una estructura y hacer que el procedimiento devuelva una estructura. También es posible pasar múltiples parámetros en las RPC, pero para ello debe utilizarse la opción `-N` con el programa `rpcgen`.

6.3.3.3. Servidores

Los servidores deben implementar cada una de las funciones especificadas en el archivo de definición de interfaces. Estas funciones, en el caso de utilizar múltiples parámetros, tienen la siguiente forma:

```
tipo_resultado *procedimiento_V_svc(tipo_argumento2 arg1,
                                      tipo_argumento2 arg2,
                                      struct svc_req *rqstp);
```

Al nombre del procedimiento se le añade el número de versión y el sufijo «svc».

6.3.3.4. Clientes

El cliente, para poder ejecutar un procedimiento remoto, debe en primer lugar establecer una conexión con un servidor mediante el siguiente servicio:

```
CLIENT *clnt_create(char *host, u_long progrnum,
                     u_long versnum, char *nettype);
```

El primer argumento (`host`) especifica el nombre de la máquina donde ejecuta el servidor. Los dos siguientes argumentos especifican el número de programa y número de versión del procedimiento remoto a ejecutar. El último argumento especifica el protocolo (`udp` o `tcp`).

El prototipo que debe emplear el cliente para las llamadas a procedimientos remotos es:

```
tipo_resultado *procedimiento_V(tipo_argumento1 arg1,
                                  tipo_argumento2 arg2, ..., CLIENT *cl);
```

donde el sufijo `V` representa el número de versión.

6.3.3.5. Ejemplo

Si se desea realizar un servidor que exporte dos procedimientos remotos, sumar y restar, la interfaz que debe incluirse en un archivo `.x`, por ejemplo, `calcular.x`, debe ser:

```
program calcular {
    version UNO {
        int sumar(int a, int b) = 1;
        int restar(int a, int b) = 2;
    } = 1;
} = 9999999;
```

Una vez definida la interfaz, ésta se procesa de la siguiente forma:

```
rpcgen -N calcular.x
```

El programa rpcgen generará de forma automática los siguientes archivos:

- **calcular.h**, archivo de cabecera a incluir en el cliente y en el servidor..
- **calcular_svc.c**, resguardo o suplente del servidor.
- **calcular_clnt.c**, resguardo o suplente del cliente.
- **calcular_xdr.c**, programa con funciones para la transformación de tipos.

El programador, a continuación, tiene que codificar el programa cliente y el programa servidor. El programa servidor debe incluir la implementación de las funciones sumar y restar.

```
#include <calcular.h>
int * sumar_1_svc(int a, int b, struct svc_req *rqstp)
{
    static int r;

    r = a + b;

    return(&r);
}

int * restar_1_svc(int a, int b, struct svc_req *rqstp)
{
    static int r;

    r = a - b;

    return(&r);
}
```

Debido a que las funciones anteriores devuelven un puntero (una dirección de memoria), las variables que almacenan el resultado deben ser de tipo **static** (**static int r**).

El código del cliente tendrá, entre otras cosas, las siguientes:

```
#include <calcular.h>
void main (int argc, char **argv)
{
    char *host;
    CLIENT *sv1;
    int *res;

    host = argv[1];

    sv = clnt_create(host, CALCULAR, UNO, "tcp");
    if (sv == NULL) {
        clnt_pcreateerror(host);
        exit(0);
    }

    res = sumar_1(5, 2, sv);
```

```

        if (res == NULL) {
            clnt_perror(svl, "call failed");
            exit(0);
        }
        printf("5 + 2 = %d\n", *res);

        clnt_destroy(sv);
        exit(0);
    }
}

```

El programa cliente recibe el nombre de la máquina donde ejecuta el servidor en la línea de argumentos y la almacena en la variable host.

Para obtener los archivos ejecutables cliente y servidor respectivamente, debe compilarse de la siguiente forma:

```

gcc -c calcular_svc.c
gcc -c calcular_xdr.c
gcc -c cliente.c
gcc -c servidor.c
gcc -o calcular_cnlt.o calcular_xdr.o cliente.o -o cliente
gcc -o calcular_svc.o calcular_xdr.o servidor.o -o servidor

```

Los dos últimos mandatos permiten obtener los ejecutables del cliente (denominado cliente) y del servidor (denominado servidor).

6.4. PRÁCTICA: PARÁMETROS DE CONFIGURACIÓN Y MONITORIZACIÓN DE CONEXIONES DE RED EN WINDOWS 2000

6.4.1. Objetivos de la práctica

El objetivo de la práctica es familiarizar al alumno con algunas herramientas, disponibles en computadoras con Windows 2000, que permiten obtener los parámetros de configuración de los protocolos TCP/IP, así como la posibilidad de monitorizar ciertos aspectos relacionados con el protocolo TCP/IP.

NIVEL: Introducción.

HORAS ESTIMADAS: 4.

6.4.1.1. Uso del mandato IPCONFIG

El mandato IPCONFIG se puede utilizar desde la línea de mandatos para mostrar la configuración de los protocolos TCP/IP de una computadora con sistema operativo Windows 2000. Esta información también se puede obtener utilizando la opción *Conexiones de red y acceso telefónico*; sin embargo, este mandato es más rápido y sencillo. Una forma de obtener toda la información de configuración de su computadora es ejecutar el siguiente mandato:

```
ipconfig /all
```

Utilice este mandato para obtener la siguiente información de su computadora:

1. Nombre de su computadora (*host*).
2. Dirección IP.
3. Dirección física de su interfaz de red.
4. Máscara de la subred.
5. Servidores de DNS (*Domain Name Service*) utilizados para traducir los nombres lógicos a direcciones IP.

6.4.1.2. Uso del monitor de rendimiento de Windows

En este apartado de la práctica se propone el empleo del monitor de rendimiento para comprobar ciertos parámetros del uso de los protocolos TCP/IP de su sistema. Para ejecutar el monitor de rendimiento siga los siguientes pasos:

1. Seleccione el botón **Inicio | Programas | Herramientas administrativas**.
2. Pulse dentro del menú anterior la opción **Monitor de sistema**.

A continuación aparecerá una ventana como la que se muestra en la Figura 6.5.

El monitor de rendimiento permite monitorizar ciertos parámetros relacionados con el rendimiento del sistema; en concreto, se pueden monitorizar parámetros relacionados con los protocolos IP, TCP y UDP. Para monitorizar uno o varios parámetros basta con pulsar el ícono **Agregar contador** (*add counter*) de la pantalla inicial que se muestra en la Figura 6.6 y elegir los parámetros que se quieren visualizar. Por ejemplo, se pueden monitorizar los siguientes parámetros:

- Datagramas IP enviados por segundo.
- Datagramas IP recibidos por segundo
- Datagramas UDP recibidos por segundo.
- Número de conexiones TCP activas.

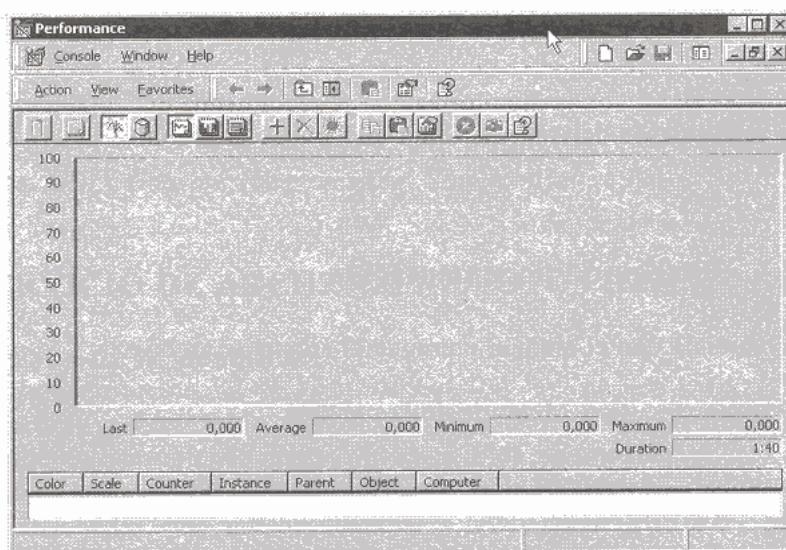


Figura 6.5. Ventana inicial del monitor de rendimiento de Windows.

Utilice este mandato para obtener la siguiente información de su computadora:

1. Nombre de su computadora (*host*).
2. Dirección IP.
3. Dirección física de su interfaz de red.
4. Máscara de la subred.
5. Servidores de DNS (*Domain Name Service*) utilizados para traducir los nombres lógicos a direcciones IP.

6.4.1.2. Uso del monitor de rendimiento de Windows

En este apartado de la práctica se propone el empleo del monitor de rendimiento para comprobar ciertos parámetros del uso de los protocolos TCP/IP de su sistema. Para ejecutar el monitor de rendimiento siga los siguientes pasos:

1. Seleccione el botón **Inicio | Programas | Herramientas administrativas**.
2. Pulse dentro del menú anterior la opción **Monitor de sistema**.

A continuación aparecerá una ventana como la que se muestra en la Figura 6.5.

El monitor de rendimiento permite monitorizar ciertos parámetros relacionados con el rendimiento del sistema; en concreto, se pueden monitorizar parámetros relacionados con los protocolos IP, TCP y UDP. Para monitorizar uno o varios parámetros basta con pulsar el icono **Agregar contador** (*add counter*) de la pantalla inicial que se muestra en la Figura 6.6 y elegir los parámetros que se quieren visualizar. Por ejemplo, se pueden monitorizar los siguientes parámetros:

- Datagramas IP enviados por segundo.
- Datagramas IP recibidos por segundo
- Datagramas UDP recibidos por segundo.
- Número de conexiones TCP activas.

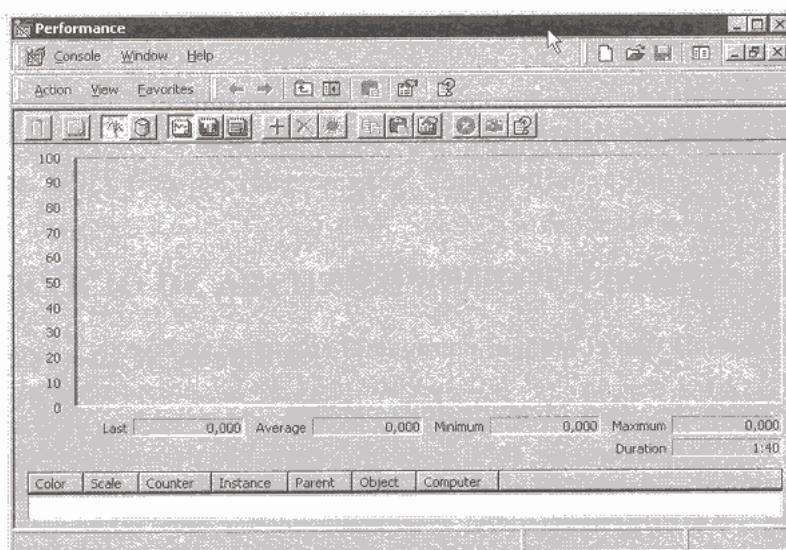


Figura 6.5. Ventana inicial del monitor de rendimiento de Windows.

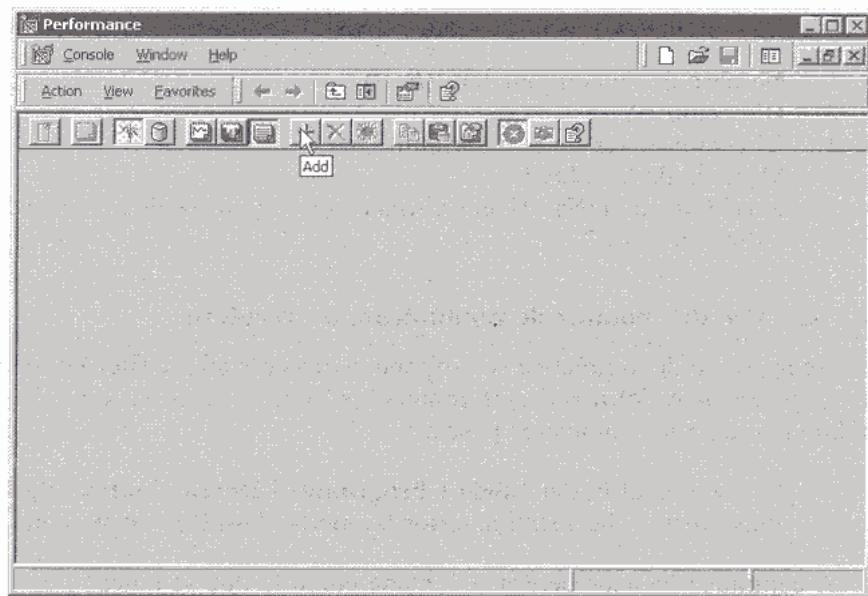


Figura 6.6. Agregar un nuevo contador.

El trabajo a desarrollar por el alumno consistirá en:

1. Describir todos los parámetros relacionados con los protocolos IP, TCP y UDP que se pueden monitorizar en su sistema.
2. Para los parámetros anteriores se debe indicar el número medio, máximo y mínimo que se obtiene durante un periodo de monitorización de diez minutos.

6.4.2. Recomendaciones generales

Antes de empezar con la monitorización se recomienda que el alumno se familiarice con el monitor de rendimiento del sistema. Para ello puede consultarse la ayuda en línea que ofrece dicho monitor.

6.4.3. Entrega de documentación

Se recomienda que el alumno entregue el siguiente archivo:

- Memoria.doc: Memoria de la práctica.

6.4.4. Bibliografía

- K. Siyan. *Microsoft Windows 2000 TCP/IP*. Prentice-Hall, mayo 2000.

6.5. PRÁCTICA: USO DEL MANDATO `rpcinfo` EN LINUX

6.5.1. Objetivos de la práctica

El objetivo de la práctica es que el alumno conozca el uso del mandato `rpcinfo` de UNIX/Linux. Este mandato permite conocer los servidores que emplean las RPC de Sun descritas al comienzo del capítulo que se encuentran arrancados en una determinada computadora.

NIVEL: Introducción.

HORAS ESTIMADAS: 2.

6.5.2. Descripción de la funcionalidad que debe desarrollar el alumno

Para conocer los servicios arrancados en una determinada computadora debe ejecutarse el siguiente mandato:

```
rpcinfo -p nombre_computadora
```

Este mandato muestra por pantalla una serie de servicios. Para cada servicio se muestra el número de programa, la versión, el protocolo utilizado (TCP o UDP), el número de puerto del servidor y el nombre del servicio.

El trabajo a desarrollar por el alumno consistirá en la ejecución de este mandato para conocer los servicios arrancados en determinadas máquinas de su laboratorio.

6.5.3. Entrega de documentación

Se recomienda que el alumno entregue el siguiente archivo:

- Memoria.txt: Memoria de la práctica.

6.5.4. Bibliografía

- R. Stevens. *UNIX Network Programming*. Prentice-Hall, 1996.

6.6. PRÁCTICA: SERVICIO DE ARCHIVOS REMOTOS UTILIZANDO SOCKETS

6.6.1. Objetivos de la práctica

Un **sistema de archivos distribuido** se encarga de la integración transparente de los archivos de un sistema distribuido, permitiendo compartir datos a los usuarios del mismo. En un sistema de archivos distribuido, cada archivo se almacena en un único servidor. El objetivo de esta práctica es doble; por una parte, se pretende que los alumnos se familiaricen con los servicios disponibles en sistemas UNIX/Linux para trabajar con *sockets*. Estos servicios permiten construir aplicaciones distribuidas utilizando el nivel de transporte de los protocolos TCP/IP. El segundo gran objetivo

de la práctica es que los alumnos entiendan y analicen todos los aspectos relacionados con el diseño e implementación de un pequeño servicio de archivos distribuido.

NIVEL: Avanzado.

HORAS ESTIMADAS: 16.

6.6.2. Descripción de la funcionalidad que debe desarrollar el alumno

Un servicio de archivos se encarga de proporcionar a los clientes acceso a los datos de los archivos almacenados en un servidor. Para el diseño e implementación del servicio de archivos se utilizará un modelo de acceso basado en servicios remotos. En un **modelo de acceso basado en servicios remotos**, el servidor ofrece todos los servicios relacionados con el acceso a los archivos. Todas las operaciones de acceso a los archivos se resuelven mediante peticiones a los servidores siguiendo un modelo cliente-servidor.

Este servicio a desarrollar estará compuesto de:

1. Un servidor que implementa el servicio.
2. Una interfaz de bajo nivel que utilizarán los clientes para acceder al servicio.

La **interfaz** que debe implementar el alumno es la siguiente:

```
FD *FD_creat(char *maquina, int puerto, char *nombre, int modo);
```

Esta función crea un archivo con nombre *nombre* en la máquina basada en el primer argumento. El segundo argumento (*puerto*) indica el puerto en el que se encuentra el servidor que implementa el servicio. La llamada devuelve un puntero a un descriptor de archivo de tipo FD. Este descriptor se describirá más adelante. El argumento *modo* tiene el mismo significado que en la llamada *creat* de POSIX. En caso de error, la llamada devuelve NULL. El archivo que se crea queda abierto para escritura (igual que la llamada *creat* en POSIX).

```
FD *FD_open(char *maquina, int puerto, char *nombre, int flags);
```

La función abre un archivo cuyo nombre se pasa como parámetro. El archivo reside en la máquina con nombre *maquina*. El segundo argumento (*puerto*) indica el puerto en el que se encuentra el servidor que implementa el servicio. La llamada devuelve un puntero a un descriptor de archivo de tipo FD. Este descriptor se describirá más adelante. El argumento *flags* tiene el mismo significado que en la llamada *open* de POSIX e indica si el archivo se abre para lectura, escritura o lectura-escritura. En caso de error, la llamada devuelve NULL.

```
int FD_close(FD *fd);
```

La función cierra un archivo previamente abierto, cuyo descriptor es *fd*. La llamada devuelve 0 en caso de éxito y -1 en caso de error.

```
int FD_read(FD *fd, char *buf, int long);
```

Esta función es similar a la llamada *read* de POSIX y sus argumentos tienen el mismo significado. En este caso, se leen datos del archivo previamente abierto cuyo descriptor es *fd* de tipo FD.

```
int FD_write(FD *fd, char *buf, int long);
```

Esta función es similar a la llamada `write` de POSIX y sus argumentos tienen el mismo significado. En este caso, se escriben datos en el archivo previamente abierto cuyo descriptor es `fd` de tipo `FD`.

```
int FD_get_size(FD *fd);
```

Esta función devuelve el tamaño del archivo correspondiente al descriptor `fd`. La llamada devuelve 0 en caso de éxito y -1 en caso de error.

```
FD *FD_unlink(char *maquina, int puerto, char *nombre);
```

La función borra un archivo de una máquina. Los argumentos son similares a los de las llamadas `FD_creat` y `FD_open`.

El tipo `FD` se utiliza como descriptor de archivos en este servicio y permite identificar a un archivo en nuestro sistema distribuido. Existen varias posibilidades para definir este tipo, según el esquema de comunicación que se utilice. El alumno puede elegir el que más le convenga. Dos posibles enfoques son:

1. Mantener la conexión abierta durante toda la sesión (desde que se abre o se crea el archivo hasta que se cierra). En este caso, si el servidor no es concurrente, no se podrá atender a varios clientes a no ser que se utilice la llamada `select`. En este caso, el descriptor de archivo debe contener el *socket* de la conexión. Un posible ejemplo sería el siguiente:

```
typedef struct descriptor{
    int socket;
    int fd; /* descriptor de archivo remoto */
} FD;
```

2. Crear una nueva conexión por cada operación que se realice. En este caso, cada operación necesita conocer la máquina y el puerto del servidor que atiende la petición. Estos datos deberán ir incluidos en el descriptor de archivo. La ventaja de esta solución frente a la anterior es que el servidor puede atender a varios clientes sin necesidad de utilizar la llamada `select`. Su desventaja es la sobrecarga de cada operación por la necesidad de establecer una conexión con el servidor en cada operación. Un posible ejemplo sería el siguiente:

```
typedef struct descriptor{
    struct sockaddr_in sock;
    int fd; /* descriptor de archivo remoto */
} FD;
```

Se valorará positivamente la implementación de un servidor *multithread*.

6.6.3. Ejemplo de aplicación cliente

El alumno deberá realizar, utilizando el servicio de archivos de bajo nivel diseñado e implementado anteriormente, una aplicación que permita transferir archivos entre un proceso cliente y otro servidor. El proceso cliente tendrá un bucle que aceptará las siguientes órdenes:

- **get host archivo_remoto archivo_local.** Permite transferir desde un *host* donde ejecuta el servidor un archivo remoto a un archivo local en la máquina cliente.
- **put archivo_local host archivo_remoto.** Transfiere un archivo a la máquina remota.

Para facilitar la implementación, los nombres de archivo se indicarán con camino absoluto (por ejemplo: /export/home/pepe/archivo.txt). El cliente finalizará su ejecución al teclear Control-D (fin de archivo en UNIX).

6.6.4. Recomendaciones generales

Es importante estudiar el funcionamiento de los servicios que ofrece el estándar POSIX para trabajar con archivos (`open`, `creat`, `close`, `read` y `write`). Para ello puede consultarse el Apéndice A.

6.6.5. Entrega de documentación

La documentación a entregar será la siguiente:

- `memoria.txt`: Memoria de la práctica, que incluirá el diseño del servidor de archivos, el diseño de la aplicación de transferencia de archivos y todos aquellos aspectos que se consideren de interés.
- `servicio.h` y `servicio.c`: Código fuente con la implementación del servicio de archivos que puede utilizar un cliente.
- `cliente.c`: Código fuente en C con el programa cliente que realiza la transferencia de archivos. Este módulo utilizará las funciones implementadas en el módulo anterior.
- `servidor.c`: Código fuente en C con el programa servidor que implementa el servicio de archivos

6.6.6. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- W. Richard Stevens. *Unix Network Programming*, vol. 2: *Interprocess Communications*. 2.^a edición. Prentice-Hall, 1999.
- K. A. Robbins y S. Robbins. *UNIX Programación Práctica*. Prentice-Hall, 1997.
- F. García, J. Carretero, J. Fernández y A. Calderón. *El lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.

6.7. PRÁCTICA. SISTEMA DE ARCHIVOS REMOTO UTILIZANDO RPC

6.7.1. Objetivos

El objetivo de esta práctica es que el alumno aprecie la facilidad que supone el uso de RPC para el desarrollo de aplicaciones y servicios distribuidos. Para ello se propone la realización de la

práctica anterior, pero utilizando en este caso llamadas a procedimientos remotos, en concreto las RPC de Sun disponibles en cualquier sistema UNIX.

NIVEL: Avanzado.

HORAS ESTIMADAS: 8.

6.7.2. Descripción de la funcionalidad que debe desarrollar el alumno

El alumno debe desarrollar la interfaz del servicio de archivos propuesto en la práctica descrita en la Sección 6.6. A continuación se recuerda esta interfaz:

```
FD *FD_creat(char *maquina, int puerto, char *nombre, int modo)
FD *FD_open(char *maquina, int puerto, char *nombre, int flags)
int FD_close(FD *fd)
int FD_read(FD *fd, char *buf, int long)
int FD_write(FD *fd, char *buf, int long)
int FD_get_size(FD *fd)
FD *FD_unlink(char *maquina, int puerto, char *nombre)
```

6.7.3. Ejemplo de aplicación cliente

El alumno deberá realizar, utilizando el servicio de archivos diseñado e implementado anteriormente, una aplicación que permita transferir archivos entre un proceso cliente y otro servidor. El proceso cliente tendrá un bucle que aceptará las siguientes órdenes:

- **get host archivo_remoto archivo_local.** Permite transferir desde un *host* donde ejecuta el servidor un archivo remoto a un archivo local en la máquina cliente.
- **put archivo_local host archivo_remoto.** Transfiere un archivo a la máquina remoto.

Para facilitar la implementación, los nombres de archivo se indicarán con camino absoluto (por ejemplo: /export/home/pepe/archivo.txt). El cliente finalizará su ejecución al teclear Control-D (fin de archivo en UNIX).

6.7.4. Recomendaciones generales

Es importante estudiar el ejemplo de RPC descrito al comienzo del capítulo. Para que no haya problemas con versiones de diferentes grupos de prácticas se utilizará como número de programa el número de matrícula que identifica a uno de los alumnos.

6.7.5. Entrega de documentación

Se recomienda que el alumno entregue los siguientes archivos:

- memoria.txt: Memoria de la práctica (véase Normas de presentación generales).
- sf.x: Archivo con la especificación de la interfaz.

- `sf.c`: Código fuente en C con la implementación del servicio.
- `cliente.c`: Código fuente en C con el programa cliente, que se encarga de la transferencia.

6.7.6. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill 2001.
- W. Richard Stevens. *Unix Network Programming*, vol. 2: *Interprocess Communications*. 2.^a edición. Prentice-Hall, 1999.
- K. A. Robbins y S. Robbins. *UNIX programación práctica*. Prentice-Hall, 1997.
- F. García, J. Carretero, J. Fernández y A. Calderón. *El lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.

6.8. PRÁCTICA: SERVICIO DE BLOQUES PARALELO CON TOLERANCIA A FALLOS RAID5

6.8.1. Objetivos de la práctica

El sistema de archivos es uno de los componentes fundamentales de cualquier sistema distribuido y es uno de los que mayor fiabilidad requiere. Habitualmente, la fiabilidad en los sistemas de archivos distribuidos se consigue mediante la **replicación** de archivos, es decir, la copia de los archivos en diferentes servidores, de forma que si uno de los servidores es inaccesible, siempre quedarán otros que sigan ofreciendo el servicio y, por tanto, el acceso a los datos. En esta práctica se propone un enfoque completamente distinto para conseguir fiabilidad y tolerancia a fallos. Para ello se propone construir un servicio de bloques paralelo soportado por diferentes servidores. La fiabilidad se conseguirá aplicando el concepto de disco RAID (*Redundant Array of Inexpensive Disks*) a estos servidores. El desarrollo de esta práctica permitirá al alumno:

1. Comprender el funcionamiento de un servicio de bloques paralelo y cómo se pueden agregar varios servidores de bloques para construir un servicio de bloques de mayor capacidad.
2. Comprender el funcionamiento de un dispositivo RAID5 y cómo este tipo de concepto se puede aplicar también a los servidores.

NIVEL: Diseño.

HORAS ESTIMADAS: 16.

6.8.2. Dispositivos RAID

La técnica habitual para ofrecer fiabilidad y tolerancia a fallos en los dispositivos de almacenamiento consiste en utilizar dispositivos RAID (*Redundant Array of Inexpensive Disks*). Estos dispositivos están formados por un conjunto de discos que almacenan la información y otro conjunto que almacena información de paridad del conjunto anterior. Desde el punto de vista físico, se ve como un único dispositivo, ya que existe un único controlador para todos los discos. Este controlador se encarga de reconfigurar y distribuir los datos entre los discos de forma transparente.

al sistema de E/S. Se han descrito diferentes niveles de discos RAID. Uno de los más utilizados habitualmente es el denominado RAID5. Un dispositivo RAID5 (véase la Figura 6.7) reparte los bloques y la paridad por todos los discos de forma cíclica. La paridad se calcula por el *hardware* en el controlador haciendo el XOR de los bloques de datos de cada franja. Así, por ejemplo, la paridad correspondiente a los bloques 0, 1 y 2 del RAID se almacena en el bloque físico 0 del disco 3.

A continuación se describen diferentes escenarios de funcionamiento en el acceso a un bloque del RAID5.

- *Lectura de un bloque de un disco sin fallo.* Si se desea leer, por ejemplo, el bloque 2 basta con calcular el disco y el bloque correspondiente a ese disco. El bloque 2 en el RAID5 de la Figura 6.7 se corresponde con el bloque 0 del disco 2. Si este disco está funcionando correctamente, se lee el bloque y se devuelve.
- *Lectura de un bloque de un disco con fallo.* Si se desea leer un bloque, por ejemplo, el bloque 2, de un disco con fallo, el controlador necesita reconstruir la información de ese bloque a partir de los otros bloques de su franja. Para ello deberá leer el bloque 0, el 1 y el bloque de paridad. El bloque 2 se obtiene calculando la paridad de estos tres bloques: Bloque 2 = bloque 0 \oplus bloque 1 \oplus bloque de paridad (donde \oplus define la operación XOR).
- *Escritura de un bloque en un RAID5 sin fallo.* Para escribir un bloque, por ejemplo, el 2, es necesario actualizar el bloque de paridad. Para ello se lee el bloque 2, el bloque de paridad y se calcula la paridad como: nueva paridad = (bloque 2 antiguo \oplus paridad antigua) \oplus bloque 2 nuevo. Una vez calculada la paridad, se escribe el bloque 2 y el nuevo bloque de paridad.
- *Escritura de un bloque en un RAID5 con fallo en el bloque de paridad.* En este caso basta con escribir el bloque en el disco correspondiente. Puesto que el disco de paridad no funciona, no se puede escribir en él.
- *Escritura de un bloque en un RAID5 con fallo en el disco en el que se almacena el bloque.* En este caso no se puede escribir en dicho disco; sin embargo, se necesita actualizar el

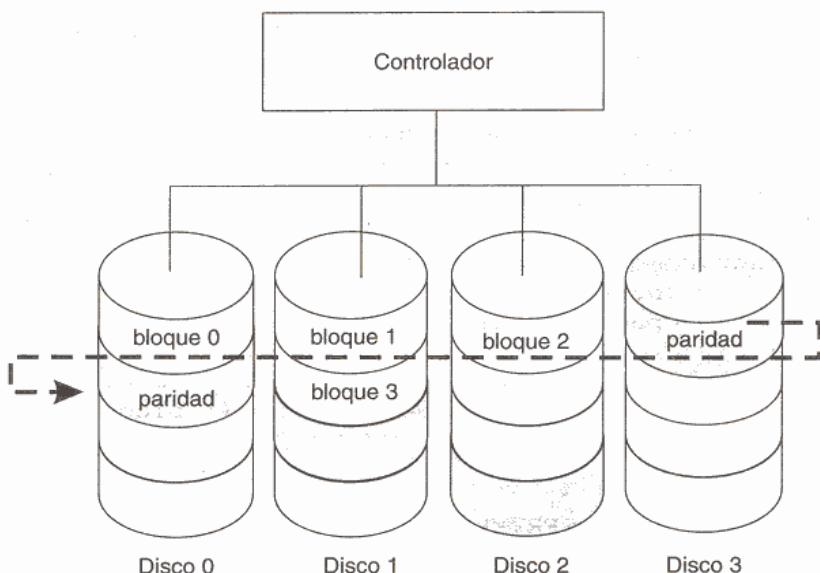


Figura 6.7. Distribución de bloques en un dispositivo RAID5.

bloque de paridad para que la información se pueda recuperar posteriormente. Para ello se lee el bloque 0 y el 1 y se calcula la nueva paridad como: nueva paridad = bloque 0 \oplus bloque 1 \oplus nuevo bloque 2. A continuación se escribe el bloque de paridad.

Observe que con el funcionamiento descrito anteriormente siempre se podrá acceder, tanto para lectura como para escritura, a los datos siempre y cuando no fallen más de un disco.

Una cuestión a tener en cuenta cuando se realizan escrituras concurrentes sobre un RAID 4 o un RAID5 es que las mismas tienen que hacerse en exclusión mutua, es decir, dos escrituras no pueden realizarse de forma concurrente sobre la misma franja de paridad debido a que podrían dejar el RAID en un estado inconsistente. Considere el siguiente escenario: dos procesos que intentan escribir en la misma franja de paridad (véase la Figura 6.7), el proceso A quiere escribir en el bloque 1 y el proceso B en el bloque 2, y acceden de la siguiente forma:

Proceso A	Proceso B
1 read(bloque1)	2 read(bloque2)
3 read(paridad)	4 read(paridad)
5 calculo nueva paridad	5 calculo nueva paridad
7 write(nuevo bloque1)	6 write(nuevo bloque2)
8 write(nueva paridad)	9 write(nueva paridad)

En el ejemplo anterior, el bloque de paridad que queda escrito en el RAID es el bloque de paridad que calcula el proceso B. Este nuevo bloque de paridad no tiene en cuenta al bloque 1 escrito por el proceso A; por tanto, la franja de paridad correspondiente a los bloques 1, 2 y 3 queda en un estado inconsistente. Para solucionar este problema es necesario disponer de un mecanismo de cerrojos que permita bloquear las franjas cuando se va a realizar una operación de escritura. El objetivo es impedir que dos procesos escriban de forma concurrente sobre una misma franja. Aunque este problema habría que resolverlo en un sistema real, en la práctica a desarrollar no se contemplará el uso de este servicio de cerrojos.

6.8.3. Descripción de la funcionalidad que debe desarrollar el alumno

El alumno parte de un servidor de bloques ya implementado (se proporciona el ejecutable al alumno) con la siguiente interfaz (utilizando las RPC de Sun), disponible en el archivo `block_server.x`:

```

struct block_request {
    int block_no;                      /* número de bloque */
    opaque data[1024];                  /* bloque de datos */
} ;
struct block_result {
    opaque data[1024];                  /* bloque de datos */
    int result;                         /* resultado de la operación */
} ;

program BLOCK_SERVER
{
    version UNO
    {

```

```

block_result  read_block ( int block_no ) = 1;
int          write_block ( block_request arg ) = 2;
int          total_blocks( ) = 3;
} = 1 ;
} = 9999999 ;
}

```

La función `read_block` permite leer el bloque con número `block_no` del servidor. El tamaño del bloque de cada servidor es de 1 KB. La función devuelve en el campo `result` de `block_result` un 0 en caso de éxito y -1 en caso de error. La función `write_block` permite escribir un bloque en el servidor. La estructura `block_request` incluye el número de bloque y el contenido del bloque a leer. La función devuelve 0 en caso de éxito y valor -1 en caso de error. La función `total_blocks` devuelve el número total de bloques que gestiona el servidor de bloques.

El alumno debe utilizar **cuatro** servidores para construir un sistema de almacenamiento que define una partición distribuida entre los cuatro servidores de la forma que se muestra en la Tabla 6.1.

Esta partición distribuida tiene un comportamiento similar a un RAID5 sobre discos. Es muy importante entender cuál es la distribución de los bloques lógicos en los distintos nodos. Para empezar, los bloques que sirven los servidores de bloques los denominamos *bloques físicos*. Los bloques que sirve la biblioteca RAID5 a través de su interfaz se denominan *bloques lógicos*. Hay una correspondencia entre un bloque lógico y su bloque físico correspondiente. Como se puede ver en la figura, cada elemento de la tabla representa el número de bloque lógico de la partición distribuida. Así, el bloque lógico 7 se corresponde con el bloque físico 0 del servidor 2. Por su parte, la paridad correspondiente a los bloques lógicos 3, 4 y 5 se encuentra en el bloque físico 1 del servidor 2.

El alumno debe utilizar cuatro servidores con la interfaz descrita anteriormente para construir sobre estos cuatro servidores un RAID5 con la distribución de bloques lógicos descrita anteriormente. Para ello debe implementar las dos siguientes funciones (cuya interfaz se encuentra en el archivo `raid5.h`, que se proporciona al alumno):

```

int raid5_read_block ( int block_no, char *data );
int raid5_write_block( int block_no, char *data );

```

La función `raid5_read_block` lee el bloque lógico con número `block_no` y lo almacena en el *buffer* `data`. La función `raid5_write_block` escribe un bloque lógico. Las funciones deben devolver 0 en caso de éxito y -1 en caso de error. Estas dos funciones deben gestionar el RAID5 y hacer frente de forma transparente a la caída de uno de los servidores.

Se proporciona al alumno un esqueleto de programa (`test_raid5.c`) que sirve para acceder al RAID5. Este programa permite leer y escribir bloques del RAID5 utilizando las funciones

Tabla 6.1. Distribución de bloques en la partición RAID5

	servidor 0	servidor 1	servidor 2	servidor 3
b. físico 0	0	1	2	P
b. físico 1	3	4	P	5
b. físico 2	6	P	7	8
b. físico 3	P	9	10	11
b. físico 4	12	13	14	P
...

anteriores. El alumno debe modificar este programa para inicializar el RAID. Se recomienda que el alumno implemente en el archivo `raid5.c` una función, denominada `raid5_init`, que se encargue de inicializar el RAID.

El alumno ha de programar la biblioteca `raid5`, así como el programa de prueba comentado anteriormente. El alumno ha de implementar, además, la funcionalidad de reconstrucción de la información de un servidor, supuesto que éste deje de funcionar y más tarde se rearranque.

La práctica ha de probarse suficientemente eligiendo diferentes servidores de bloques a la hora de simular la caída matando el proceso.

6.8.4. Código fuente de apoyo

Para facilitar la realización de la práctica, se dispone en la página web del libro del archivo `practica-6.8.tar.gz`, que contiene código fuente de apoyo. Al extraer su contenido, se crea el directorio `practica-6.8`, donde se debe desarrollar la práctica. Dentro de este directorio, se encuentran los siguientes archivos:

- `Makefile`: archivo fuente para la herramienta `make`. NO debe ser modificado. Con él se consigue la recopilación automática de los archivos fuente cuando se modifique. Basta con ejecutar el mandato `make` para que el programa se compile de forma automática.
 - `block_server`: ejecutable de un servidor.
 - `block_server.x`: archivo con la interfaz de un servidor.
 - `raid5.h`: archivo de cabecera con la interfaz de la biblioteca `raid5` descrita anteriormente. Este archivo puede modificarse si se considera necesario.
 - `raid5.c`: archivo fuente donde se implementará la biblioteca `raid5` descrita en el enunciado.
- Este archivo SÍ debe ser modificado y será parte de la documentación a entregar.
- `test_raid5.c`: archivo que permite al alumno probar las funciones que acceden al RAID5. Este archivo debe modificarse para iniciar el RAID5.
 - `rebuild.c`: archivo que contiene la funcionalidad para reconstruir la información de un servidor que supuestamente había fallado y después se recupera.

6.8.5. Recomendaciones generales

Es importante entender las bases del sistema: RPC y RAID5. Para el cálculo de la paridad es necesario emplear el operador binario de C, que permite calcular el XOR (^) entre dos valores. Conviene que el alumno implemente una función con prototipo similar al siguiente:

```
void calcularParidad(char *datos1, char *datos2, char *paridad, int long);
```

que se encargará de calcular la paridad de dos bloques de datos (`datos1` y `datos2`) y dejar el resultado en el bloque `paridad`. La longitud de los distintos bloques vendrá dada por el último parámetro (`long`).

Es importante que el alumno tenga en cuenta qué ocurre cuando un servidor falla y se pierde la conexión con él. En este caso, se genera una señal que deberá ser tratada.

6.8.6. Entrega de documentación

El alumno debe entregar los siguientes archivos:

- memoria.txt: Memoria de la práctica.
- raid5.c y raid5.h: Código fuente en C con la biblioteca que implementa la funcionalidad pedida.
- test_raid5.c: Código fuente en C con el programa de prueba.

6.8.7. Bibliografía

- J. L. Antonakos y K. C. Mansfiled, Jr. *Programación estructurada en C*. Prentice-Hall, 1997.
- F. García, J. Carretero, J. Fernández y A. Calderón. *El lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.
- P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice-Hall, 1994.

6.9. PRÁCTICA: EXCLUSIÓN MUTUA UTILIZANDO UN ANILLO CON PASO DE TESTIGO

6.9.1. Objetivos de la práctica

El objetivo de esta práctica es mostrar al alumno la forma de ofrecer un servicio de sincronización que permite a los procesos de una aplicación distribuida acceder en exclusión mutua a un determinado recurso compartido. El método elegido es la configuración de los procesos de la aplicación como un anillo y la utilización de un mecanismo basado en el paso de un testigo.

NIVEL: Diseño.

HORAS ESTIMADAS: 16.

6.9.2. Exclusión mutua basada en paso de testigo

Una forma de asegurar la exclusión mutua en una aplicación distribuida consiste en organizar los procesos de la aplicación como un anillo lógico, tal y como se muestra en la Figura 6.8. Por el anillo va circulando un testigo. Cuando un proceso desea entrar en la sección crítica, debe esperar a estar en posesión del testigo. Mientras el testigo no llegue a ese proceso, éste no podrá pasar a ejecutar dentro de la sección crítica. Cada vez que el testigo llega a un proceso, se comprueba si el proceso desea entrar en la sección crítica. En caso de no querer entrar, envía el testigo al siguiente proceso del anillo. En caso contrario, no reenvía el testigo y pasa a ejecutar dentro de la sección crítica. El testigo permanece en el proceso hasta que abandone la sección crítica. Una vez abandonada la sección crítica, se envía el testigo al siguiente proceso, dándole la oportunidad de entrar en la sección crítica.

El proceso descrito asegura la exclusión mutua en el acceso a un determinado recurso compartido, puesto que sólo aquel proceso que esté en posesión del testigo podrá acceder al recurso.

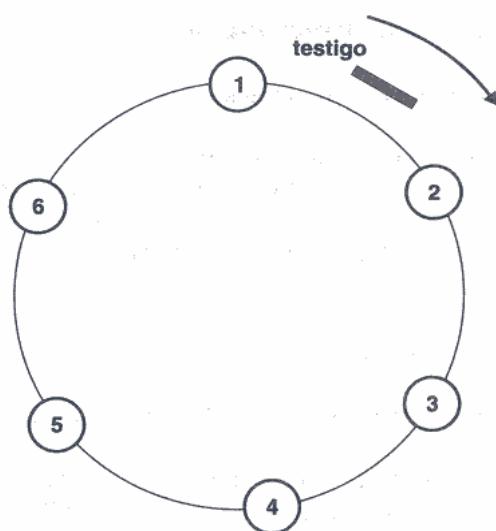


Figura 6.8. Sincronización en el acceso a un recurso compartido utilizando un anillo con paso de testigo.

6.9.3. Descripción de la funcionalidad que debe desarrollar el alumno

Se desea ofrecer un mecanismo de exclusión mutua que organice los procesos de una aplicación distribuida en forma de anillo. La descripción de la funcionalidad a desarrollar se explicará utilizando como ejemplo la aplicación distribuida de la Figura 6.9. En esta figura se muestran cuatro procesos (P_0 , P_1 , P_2 y P_3) que se van a configurar como un anillo. Para ejecutar estos cuatro procesos (bien en la misma máquina o bien en máquinas distintas) se deben ejecutar los siguientes mandatos:

```
anillo 0 m0 p0 m1 p1 m2 p2 m3 p3
anillo 1 m0 p0 m1 p1 m2 p2 m3 p3
anillo 2 m0 p0 m1 p1 m2 p2 m3 p3
anillo 3 m0 p0 m1 p1 m2 p2 m3 p3
```

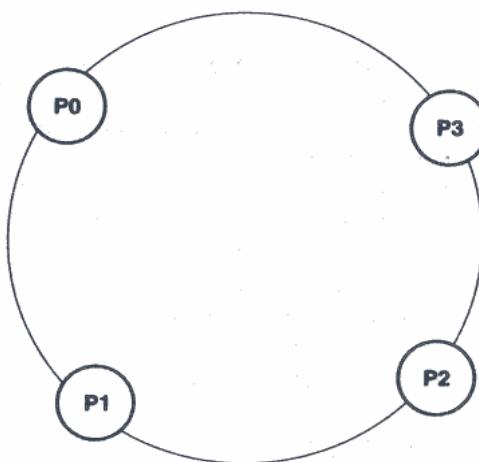


Figura 6.9. Ejemplo de aplicación configurada en un anillo.

Cada proceso (el nombre del ejecutable es anillo) recibe como segundo argumento en la línea de mandatos un número entero que lo identifica dentro del anillo. A continuación se pasan los nombres de las máquinas y los puertos donde van a ejecutar cada uno de estos procesos. Así, por ejemplo, el proceso P0 tiene como identificador 0, ejecutará en la máquina m1 y utilizará como número de puerto p1; el proceso P2 ejecutará en la máquina m2 y utilizará el puerto p2. Hay que asegurarse que cada proceso se ejecuta en la máquina correspondiente. Todos pueden ejecutarse en la misma máquina siempre que se utilicen números de puerto distintos.

Cada uno de estos procesos debe ser capaz de utilizar las siguientes primitivas a desarrollar por el alumno:

```
mutex_init(int argc, char **argv);
mutex_lock(void);
mutex_unlock(void);
```

La función `mutex_init` es la encargada de inicializar el anillo tal como se muestra en la Figura 6.10. Para conectar los procesos se utilizarán *sockets stream*. Observe que la función anterior crea un proceso ligero en cada proceso que será el encargado de hacer circular el testigo a través del anillo. Se puede elegir al proceso con identificador 0 como el proceso que inserta el testigo inicialmente en el anillo. El testigo puede ser un simple byte. Para que el anillo quede correctamente configurado, todos los procesos de la aplicación deben ejecutar la función

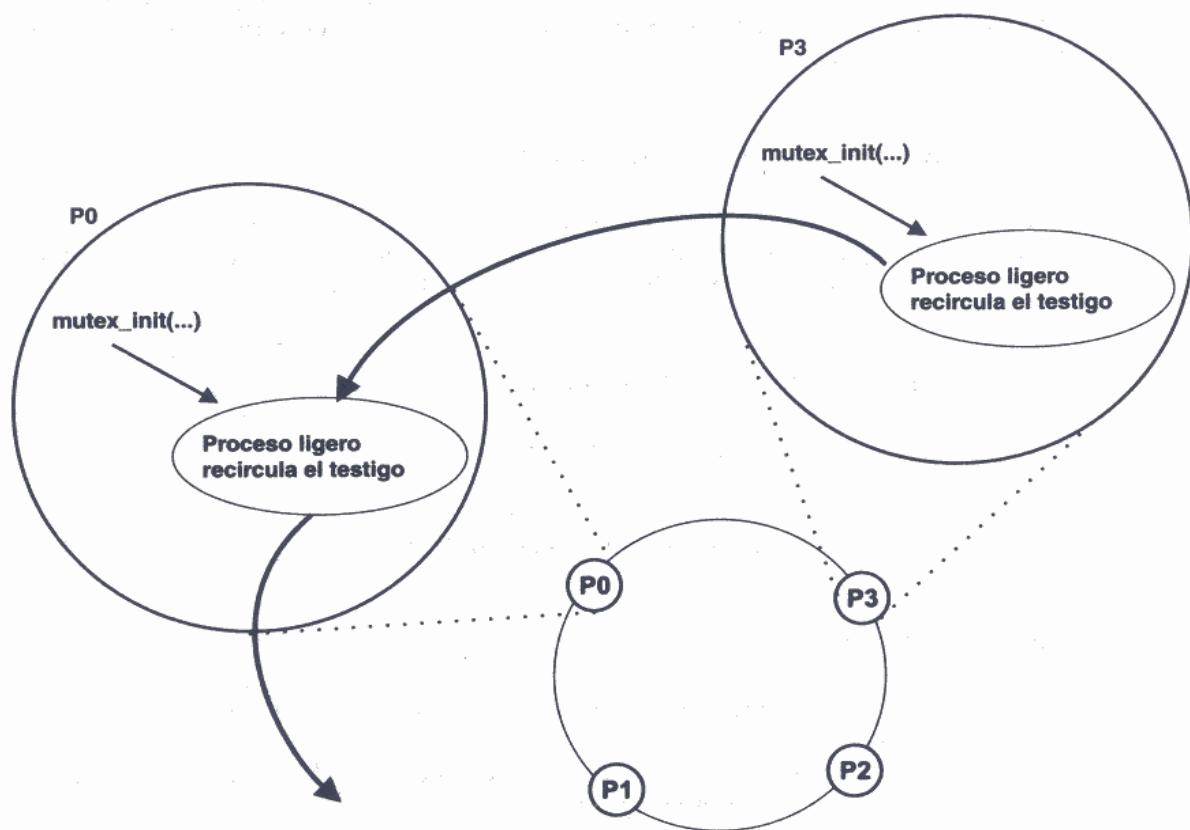


Figura 6.10. Primitiva `mutex_init`.

`mutex_init`. El objetivo del proceso ligero creado en esta función es recircular el testigo a través del anillo, es decir, debe estar en un bucle infinito similar al que se muestra a continuación:

```
while(1)
{
    leer el testigo del proceso anterior;
    enviar el testigo al proceso siguiente;
}
```

Una vez creado el proceso ligero, se deberán utilizar los servicios `accept` y `connect` para conectar de forma lógica a todos los procesos del anillo. La formación del anillo se puede realizar a partir de los parámetros `argc` y `argv` que acepta la función `main` y que se deben pasar a la función `mutex_init`.

La función `mutex_lock` se utiliza para acceder en exclusión mutua a un determinado recurso. El código de esta función se debe sincronizar con el proceso ligero que se encarga de recircular el testigo a través del anillo. La función `mutex_lock` bloqueará al proceso que ejecuta la llamada hasta que el testigo llegue al proceso ligero que se encarga de recircular el testigo.

Cuando el proceso ligero recibe el testigo, comprueba si se ha ejecutado dentro del proceso la función `mutex_lock`. Si es así, no envía el testigo el siguiente proceso del anillo y despierta al proceso suspendido en la llamada `mutex_lock` permitiendo que continúe con su ejecución. El proceso ligero se bloqueará a continuación hasta que se ejecute la función `mutex_unlock`. La función `mutex_unlock` se utiliza para liberar el acceso al recurso compartido, el código de esta función tiene que ser capaz de despertar al proceso ligero de forma que éste vuelva a insertar el testigo en el anillo.

6.9.4. Recomendaciones generales

Para sincronizar el código de las funciones `mutex_lock` y `mutex_unlock` con el código del proceso ligero que se encarga de recircular el testigo a través del anillo se pueden utilizar `mutex` y variables condicionales como las descritas en el Capítulo 4.

6.9.5. Entrega de documentación

El alumno debe entregar los siguientes archivos:

- `memoria.txt`: Memoria de la práctica.
- `anillo.c`: Código fuente con el código del programa.

6.9.6. Bibliografía

- J. Carretero, F. García, P. de Miguel y F. Pérez. *Sistemas operativos: una visión aplicada*. McGraw-Hill, 2001.
- F. García, J. Carretero, J. Fernández y A. Calderón. *El lenguaje de programación C: diseño e implementación de programas*. Prentice-Hall, 2002.

A

Guía del lenguaje de programación en C

En este apéndice se realiza una pequeña introducción al lenguaje de programación C. Hay que indicar que este apéndice no pretende ser un manual de dicho lenguaje de programación, sino una pequeña guía que ayude en la realización de las prácticas propuestas en el libro.

A.1. CARACTERÍSTICAS DE C

Las principales características del lenguaje C pueden resumirse en los siguientes puntos:

- C es un lenguaje de propósito general ampliamente utilizado. Presenta características de **bajo nivel**: C trabaja con la misma clase de objetos que la mayoría de las computadoras (caracteres, números y direcciones). Esto permite la creación de programas eficientes.
- Está estrechamente asociado con el sistema operativo UNIX. UNIX y su *software* fueron escritos en C.
- Es un lenguaje adecuado para *programación de sistemas* por su utilidad en la escritura de sistemas operativos.
- Es adecuado también para cualquier otro tipo de aplicación.
- Es un lenguaje relativamente pequeño: sólo ofrece sentencias de control sencillas y funciones.
- No ofrece mecanismos de E/S (entrada/salida). Todos los mecanismos de alto nivel se encuentran fuera del lenguaje y se ofrecen como funciones de biblioteca.
- Posibilita la creación de programas portables, es decir, programas que pueden ejecutarse sin cambios en multitud de computadoras. Permite la programación estructurada y el diseño modular.

Sin embargo, aunque C ofrece innumerables ventajas, también presenta una serie de inconvenientes:

- No es un lenguaje *fueramente tipado*.
- Es bastante permisivo con la conversión de datos.
- Sin una programación metódica puede ser propenso a errores difíciles de encontrar.
- La versatilidad de C permite crear programas difíciles de leer.

A.2. PRIMER PROGRAMA EN C

En esta sección se presenta un primer programa en C que imprime por pantalla el mensaje:

```
"Primer programa en C".

#include <stdio.h>
main()
{
    /* Primer programa en C */
    printf("Primer programa en C. \n");
    exit(0);
}
```

Para crear el programa anterior es necesario *editar*lo con un editor de texto y grabarlo en un archivo, como, por ejemplo, *ejemplo.c*. El programa anterior comienza con la siguiente línea:

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

Esta línea indica al preprocesador (véase la siguiente sección) que incluya el archivo de cabecera *stdio.h*, que contiene la declaración de funciones y constantes relacionadas con la

entrada/salida. A continuación se define la función `main`, el punto de inicio de todo programa escrito en C.

A.2.1. ¿Cómo compilar y ejecutar el programa?

Un **compilador de C** permite analizar y detectar errores en el código fuente y convertir un programa escrito en C en código ejecutable por la computadora. El mandato de compilación básico es el siguiente:

```
gcc ejemplo.c
```

Este mandato permite:

- Generar el código objeto: `ejemplo.o`.
- Generar el ejecutable: `a.out`.

El programa se ejecuta tecleando `a.out`.

El mandato `gcc -c ejemplo.c` genera el archivo objeto `ejemplo.o`. El mandato `gcc ejemplo.o -o ejemplo` genera el ejecutable `ejemplo`. En este caso, el programa se ejecuta tecleando `ejemplo`.

El modelo de compilación de C se puede ver en la Figura A.1

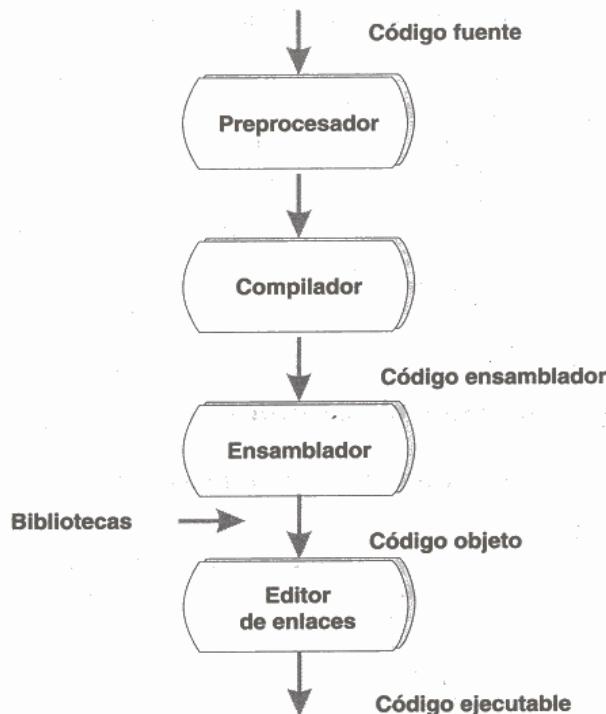


Figura A.1. *Modelo de compilación de C.*

A.3. CARACTERÍSTICAS BÁSICAS DE C

Esta sección presenta las principales características del lenguaje C.

A.3.1. Comentarios

Los comentarios comienzan con /* y finalizan con */. Algo importante a tener en cuenta es que no se pueden anidar, es decir, dentro de un comentario no puede aparecer el símbolo /*. El siguiente programa ilustra la utilización de los comentarios en C:

```
#include <stdio.h>
main()
{
    /*****
     * Esto es un comentario */
    printf("Este programa contiene\n");
    printf("un comentario. \n");
    exit(0);
}
```

A.3.2. Palabras reservadas

Las palabras reservadas que incluye el lenguaje C son las siguientes:

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
inline	int	long	register
return	short	signed	sizeof
static	struct	switch	typedef
union	unsigned	void	volatile
while			

A.3.3. Tipos de datos elementales

Los principales tipos de datos de C son los siguientes:

Tipo	Significado	Tamaños típicos en bytes
char	carácter	1
int	entero	2 – 4
short	entero corto	2
long	entero largo	4
long long	entero más largo	8
unsigned char	carácter sin signo	1

Tipo	Significado	Tamaños típicos en bytes
unsigned	entero sin signo	2 – 4
unsigned int	entero corto sin signo	2
unsigned long	entero largo sin signo	4
unsigned long long	entero más largo sin signo	8
float	coma flotante, real	4
double	coma flotante largo	8

A.3.4. Constantes

Los distintos tipos de constantes que se pueden emplear en C son:

- Caracteres: 'a', 'b'
- Valores enteros, que pueden expresarse de distintas formas:
 - Notación decimal: 987
 - Notación hexadecimal: **0x25** o **0X25**
 - Notación octal: **034**
 - Enteros sin signo: **485U**
 - Enteros de tipo long: **485L**
 - Entero de tipo long long: **756LL**
 - Enteros sin signo de tipo long: **485UL**
 - Valores negativos (signo menos): **-987**
- Valores reales (coma flotante):
 - Ejemplos: **12**, **14**, **8.**, **.34**
 - Notación exponencial: **.2e+9**, **1.04E-12**
 - Valores negativos (signo menos): **-12** **-2e+9**

A.3.5. La función printf

Esta función permite imprimir información por la salida estándar. El formato de dicha función es:

```
printf(formato, argumentos);
```

donde **formato** especifica, entre otras cosas, los tipos de datos que se van a imprimir y **argumentos** los datos o valores a imprimir.

A continuación se presenta un ejemplo de utilización:

```
printf("Hola mundo\n");
printf("El numero 28 es %d\n", 28);
printf("Imprimir %c %d %f\n", 'a', 28, 3.0e+8);
```

Los especificadores de formato permiten dar un determinado formato a la salida que se imprime con **printf**. Los principales especificadores de formato son:

Carácter	Argumentos	Resultado
d, i	entero	entero decimal con signo
u	entero	entero decimal sin signo
o	entero	entero octal sin signo
x, X	entero	entero hexadecimal sin signo
f	real	real con punto y con signo
e, E	real	notación exponencial con signo
g, G		
c	carácter	un carácter
s	cadena de caracteres	cadena de caracteres
%		imprime%
p	void	Dependiente implementación
ld, lu, lx, lo	entero	entero largo

A.3.5.1. Secuencias de escape

La secuencias de escape permiten imprimir determinados valores especiales, como, por ejemplo:

Secuencia	Significado
\n	nueva línea
\t	tabulador
\b	<i>backspace</i>
\r	retorno de carro
\"	comillas
\'	apóstrofo
\\\	<i>backslash</i>
\?	signo de interrogación

A.3.5.2. Especificadores de ancho de campo

Estos especificadores permiten indicar el ancho que se dejará en la salida para imprimir una determinada información. Así, por ejemplo:

```
printf("Número entero = %5d \n", 28);
```

produce la salida:

```
Número entero = 28
```

y también

```
printf("Número real = %5.4f \n", 28.2);
```

produce la salida:

```
Número entero = 28.2000
```

A.3.6. Variables

Una variable es un identificador utilizado para representar un cierto tipo de información. Cada variable es de un tipo de datos determinado. Una variable puede almacenar diferentes valores en

distintas partes del programa. Toda variable debe comenzar con una letra o el carácter `_`. El resto sólo puede contener letras, números o `_`.

Ejemplos de variables válidas son:

```
numero
_color
identificador_1
```

Algo importante a tener en cuenta es que C es **sensible a mayúsculas y minúsculas**, lo que quiere decir que las siguientes variables, por tanto, son todas distintas:

```
pi PI Pi pi
```

A.3.6.1. Definición de variables

Una definición asocia un tipo de datos determinado a una o más variables. El formato de una definición es:

```
tipo_de_datos var1, var2, ..., varN;
```

donde `var1, ..., varN` representan identificadores de variables válidos y `tipo_de_datos` el tipo de datos de las variables que se definen. A continuación se presentan algunos ejemplos:

```
int a, b, c;
float numero_1, numero_2;
char letra;
unsigned long entero;
```

Es importante tener en cuenta que todas las variables deben definirse antes de su uso y que deben asignarse a las variables nombres significativos. Así, en el siguiente ejemplo:

```
int temperatura;
int k;
```

`temperatura` es una variable bastante descriptiva, mientras que `k` no.

A.3.7. Expresiones y sentencias

Una **expresión** representa una unidad de datos simple, tal como un número o carácter. También puede estar formada por identificadores y operadores. A continuación se presentan dos ejemplos de expresiones válidas en C:

```
a + b
num1 * num2
```

Una **sentencia** controla el flujo de ejecución de un programa. Existen dos tipos de sentencias:

- *Sentencia simple:* formada por una única sentencia.

```
temperatura = 4 + 5;
```

- *Sentencia compuesta:* formada por varias sentencias, se encierran entre llaves:

```
{
    temperatura_1 = 4 + 5;
    temperatura_2 = 8 + 9;
}
```

A.3.8. Operador de asignación

El *operador de asignación* (=) asigna un valor a una variable. Puede asignarse valor inicial a una variable en su declaración. Considere el siguiente fragmento de programa:

```
#include <stdio.h>
main()
{
    int      a = 1;
    float   b = 4.0;
    int      c, d;
    char    letra;

    c = 10;
    letra = 'a';
    d = a + c;
    printf("a = %d \n", a);
    printf("b = %f \n", b);
    printf("c = %d \n", c);
    printf("d = %d \n", d);
    printf("La letra es %c \n", letra);
}
```

El programa anterior declara las variables a y b y les asigna valores iniciales.

A.3.9. Función **scanf()**

Esta función permite leer datos del usuario. La función devuelve el número de datos que se han leído correctamente. El formato de dicha función es el siguiente:

```
scanf(formato, argumentos);
```

Al igual que la función `printf()`, en esta función pueden especificarse determinados especificadores de formato. Por ejemplo, %f para números reales, %c para caracteres y %d para números enteros.

```
scanf("%f", &numero);
scanf("%c", &letra);
scanf("%f %d %c", &real, &entero, &letra);
scanf("%ld", &entero_largo);
```

La primera sentencia lee un número en coma flotante y lo almacena en la variable numero. La segunda lee un carácter y lo almacena en la variable letra de tipo char. La ultima sentencia lee

tres datos y los almacena en tres variables. (Es importante el uso del operador de dirección &, que se describirá más adelante.)

El siguiente programa lee un número entero y lo eleva al cuadrado.

```
#include <stdio.h>
main()
{
    int numero;
    int cuadrado;

    printf("Introduzca un numero:");
    scanf("%d", &numero);
    cuadrado = numero * numero;
    printf("El cuadrado de %d es %d\n", numero, cuadrado);
}
```

A.3.10. Introducción a la directiva #define

Esta directiva permite definir constantes simbólicas en el programa. Su formato es el siguiente:

```
#define nombre texto
```

donde **nombre** representa un nombre simbólico que suele escribirse en mayúsculas y **texto** no acaba en ;.

El **nombre** es sustituido por **texto** en cualquier lugar del programa. Considere los siguientes ejemplos:

```
#define PI      3.141593
#define CIERTO   1
#define FALSO    0
#define AMIGA "Marta"
```

El siguiente programa lee el radio de un círculo y calcula su área:

```
#include <stdio.h>
#define PI 3.141593
main()
{
    float radio;
    float area;

    printf("Introduzca el radio: ");
    scanf("%f", &radio);
    area = PI * radio * radio;
    printf("El area del circulo es %.4f \n", area);
    exit(0);
}
```

A.4. OPERADORES Y EXPRESIONES

Esta sección se dedica a presentar los principales operadores del lenguaje C.

A.4.1. Operadores aritméticos

Los operadores aritméticos de C son los siguientes:

Operador	Función
+	suma
-	resta
*	producto
/	división
%	operador módulo, resto de la división entera

Cuando los dos operandos de la división son enteros, el resultado se considera también entero. El operador % requiere que los dos operandos sean enteros. La mayoría de las versiones de C asignan al resto el mismo signo del primer operando.

A.4.1.1. Conversión de tipos

En C, un operador se puede aplicar a dos variables o expresiones distintas. Los operandos que difieren en tipo pueden sufrir una conversión de tipo. Como **norma general**, el operando de menor precisión toma el tipo del operando de mayor precisión. Las principales **reglas de conversión de tipos** son las siguientes:

1. Si un operando es long double, el otro se convierte a long double.
2. En otro caso, si es double, el otro se convierte a double.
3. En otro caso, si es float, el otro se convierte a float.
4. En otro caso, si es unsigned long int, el otro se convierte a unsigned long int.
5. Si un operando es long int y el otro es unsigned int, entonces:
 - a) Si el unsigned int puede convertirse a long int, el operando unsigned int se convertirá en long int.
 - b) En otro caso, ambos operandos se convertirán a unsigned long int.
 - c) En otro caso, si un operando es long int, el otro se convertirá a long int.
6. En otro caso, si un operando es unsigned int, el otro se convertirá a unsigned int.
7. En otro caso, ambos operandos serán convertidos a tipo int si es necesario.

A.4.1.2. Conversión de tipos o cast

Se puede convertir una expresión a otro tipo. Para ello se utiliza la siguiente expresión:

(tipo_datos) expresion

En este caso, expresion se convierte al tipo tipo_datos.

Así, por ejemplo, en la siguiente expresión, 5.5 se convierte a entero (obteniéndose 5).

(int) 5.5 % 4

A.4.1.3. Prioridad de los operadores aritméticos

La prioridad indica el orden en el que se realizan las operaciones aritméticas. Las operaciones con mayor precedencia se realizan antes. La prioridad de los operadores aritméticos se presenta a continuación:

Prioridad	Operación
Primero	()
Segundo	Negación (signo menos)
Tercero	* , / , %
Cuarto	+ , -

Dentro de cada grupo las operaciones se realizan de izquierda a derecha. La expresión:
 $a - b / c * d$ es equivalente a

$$a - ((b/c) * d)$$

A.4.1.4. Operadores de incremento y decremento

En C existen dos tipos de operadores de incremento y decremento:

- Operador de **incremento** (++) , incrementa en uno el operando.
- Operador de **decremento** (--), disminuye en uno el operando.

Estos presentan, a su vez, las siguientes variantes:

- Postincremento, `i++`
- Preincremento, `++i`
- Postdecremento, `i--`
- Predecremento, `--i`

Hay que tener en cuenta que:

- Si el operador *precede* al operando, el valor del operando se modificará **antes** de su utilización.
- Si el operador *sigue* al operando, el valor del operando se modificará **después** de su utilización.

Así, por ejemplo, si $a = 1$

```
printf("a = %d \n", a);
printf("a = %d \n", ++a);
printf("a = %d \n", a++);
printf("a = %d \n", a);
```

entonces se imprime:

```
a = 1
a = 2
a = 2
a = 3
```

Estos operadores tienen mayor prioridad que los operadores aritméticos.

A.4.2. Operadores relacionales y lógicos

Los operadores relaciones en C son:

Operador	función
<	menor que
>	mayor que
<=	menor o igual que
>=	mayor o igual que
==	igual que
!=	distinto que

Estos operadores se utilizan para formar expresiones lógicas. El resultado de una expresión lógica es un valor entero que puede ser: *cierto*, se representa con un 1; *falso*, se representa con un 0. Si $a = 1$ y $b = 2$, entonces:

Expresión	Valor	Interpretación
$a < b$	1	cierto
$a > b$	0	falso
$(a + b) != 3$	0	falso
$a == b$	0	falso
$a == 1$	1	cierto

A.4.2.1. Prioridad de los operadores relacionales

La prioridad de este tipo de operadores se presenta a continuación:

Prioridad	Operación
Primero	()
Segundo	> < >= <=
Tercero	== !=

Los operadores aritméticos tienen mayor prioridad que los operadores relacionales. Dentro de cada grupo, las operaciones se realizan de izquierda a derecha.

A.4.3. Operadores lógicos

Los operadores lógicos en C son:

Operador	Función
&&	Y lógica (AND)
	O lógica (OR)
!	NO lógico (NOT)

Estos operadores actúan sobre operandos que son a su vez expresiones lógicas que se interpretan como: *cierto*, cualquier valor distinto de 0; *falso*, el valor 0. Se utilizan para formar expresiones lógicas y su resultado es un valor entero que puede ser: *cierto*, se representa con un 1; *falso*, se representa con un 0. Las **tablas de verdad** indican el resultado de los operadores lógicos.

Tabla de verdad del Y lógico:

Expresión	Valor	Interpretación
1 && 1	1	cierto
1 && 0	0	falso
0 && 1	0	falso
0 && 0	0	falso

Tabla de verdad del O lógico:

Expresión	Valor	Interpretación
1 1	1	cierto
1 0	1	cierto
0 1	1	cierto
0 0	0	falso

Tabla de verdad de la negación lógica:

Expresión	Valor	Interpretación
! 1	0	falso
! 0	1	cierto

A.4.3.1. Prioridad de los operadores lógicos

La prioridad de los distintos operadores lógicos se presenta a continuación:

Prioridad	Operación
Primero	()
Segundo	!
Tercero	&&
Cuarto	

&& y || se evalúan de izquierda a derecha y ! se evalúa de derecha a izquierda. Las expresiones lógicas con && o || se evalúan de izquierda a derecha, **sólo** hasta que se ha establecido su valor cierto/falso. Así, por ejemplo, si $a = 8$ y $b = 3$, en la sentencia:

$a > 10 \ \&\& \ b < 4$

no se evaluará $b < 4$, puesto que $a > 10$ es falso, y, por tanto, el resultado final será falso.

Si $a = 8$ y $b = 3$, entonces en la sentencia:

$a < 10 \ || \ b < 4$

no se evaluará $b < 4$, puesto que $a < 10$ es verdadero, y, por tanto, el resultado final será verdadero.

A.4.4. Resumen de prioridades

A continuación se presenta la relación de las prioridades de los distintos operadores (vistos hasta ahora) de mayor a menor prioridad.

Operador	Evaluación
<code>++ -- (post)</code>	De izquierda a derecha
<code>++ -- (pre)</code>	De derecha a izquierda
<code>! - (signo menos)</code>	De derecha a izquierda
<code>(tipo) (cast)</code>	De derecha a izquierda
<code>* / %</code>	De izquierda a derecha
<code>+ -</code>	De izquierda a derecha
<code>< > <= >=</code>	De izquierda a derecha
<code>== !=</code>	De izquierda a derecha
<code>&&</code>	De izquierda a derecha
<code> </code>	De izquierda a derecha

A.4.5. Operadores de asignación

La forma general del operador de asignación es la siguiente:

```
identificador = expresion
```

El operador de asignación = y el de igualdad == son **distintos**. Éste es un error común que se comete cuando se empieza a programar en C. C también permite la realización de asignaciones múltiples como la siguiente:

```
id_1 = id_2 = ... = expresion
```

Las asignaciones se efectúan de derecha a izquierda. Así, en `i = j = 5`

1. A `j` se le asigna 5.
2. A `i` se le asigna el valor de `j`.

A.4.5.1. Reglas de asignación

La reglas de asignación a tener en cuenta son las siguientes:

- Si los dos operandos en una sentencia de asignación son de tipos distintos, entonces el valor del operando de la derecha será automáticamente convertido al tipo del operando de la izquierda. Además:
 - Un valor en coma flotante se puede truncar si se asigna a una variable de tipo entero.
 - Un valor de doble precisión puede redondearse si se asigna a una variable de coma flotante de simple precisión.

- Una cantidad entera puede alterarse si se asigna a una variable de tipo entero más corto o una variable de tipo carácter.

Es **importante** en C utilizar de forma correcta la conversión de tipos.

A.4.5.2. Operadores de asignación compuestos

C permite la utilización de los siguientes operadores de asignación compuestos, cuyo significado se muestra a continuación:

Expresión	Expresión equivalente
j += 5	j = j + 5
j -= 5	j = j - 5
j *= 5	j = j * 5
j /= 5	j = j / 5
j %= 5	j = j % 5

Los operadores de asignación tienen menor prioridad que el resto.

A.4.6. Operador condicional

Su forma general es:

```
expresion_1 ? expresion_2 : expresion_3
```

- Si expresion_1 es *verdadero*, devuelve expresion_2
- Si expresion_1 es *falso*, devuelve expresion_3

Su prioridad es justamente superior a los operadores de asignación. Se evalúa de derecha a izquierda. Por ejemplo, si *a* = 1, en la sentencia:

```
k = (a < 0) ? 0 : 100;
```

Primero se evalúa (*a* < 0). Como es *falso*, el operador condicional devuelve 100 y este valor se asigna a *k*. Es decir, *k* toma el valor 100.

A.5. SENTENCIAS DE CONTROL

Esta sección describe las sentencias de control de C.

A.5.1. Sentencia if

Su forma general es:

```
if (expresión)
    sentencia;
```

Si expresión es verdadera (valor distinto de 0), se ejecuta sentencia. La expresión debe estar entre paréntesis. En el caso de que sentencia sea compuesta, la forma general debería ser la siguiente:

```
if (expresión)
{
    sentencia 1
    sentencia 2
    .
    .
    .
    sentencia N
}
```

El diagrama de flujo de esta sentencia es el que se muestra en la Figura A.2.

A.5.2. Sentencia if-else

Su forma general es:

```
if (expresion)
    sentencia 1;
else
    sentencia 2;
```

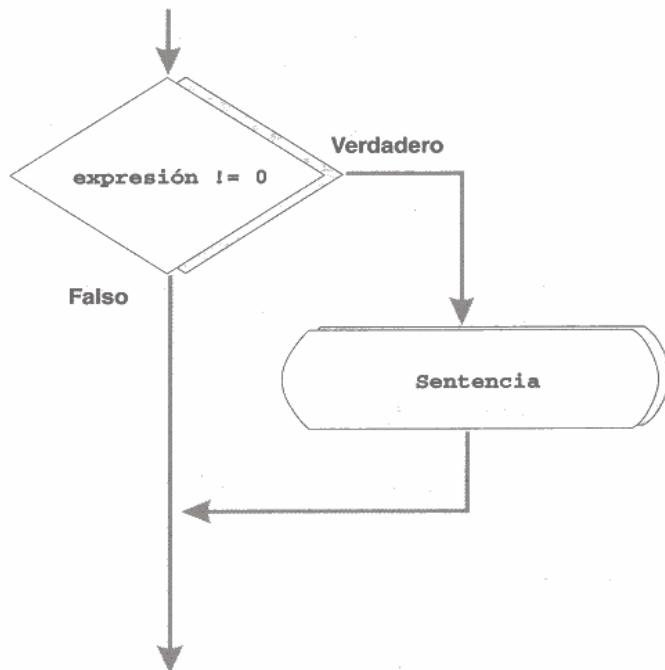


Figura A.2. Diagrama de flujo de la sentencia if.

Si expresión es verdadera (valor distinto de 0), se ejecuta sentencia 1. En caso de ser falsa (valor igual a 0), se ejecuta sentencia 2. Si las sentencias son compuestas, se encierran entre { }. El diagrama de flujo de esta sentencia es el que se muestra en la Figura A.3.

A.5.3. Sentencia `for`

Su forma general es:

```
for (expresión 1; expresión 2; expresión 3)
    sentencia;
```

Inicialmente, se ejecuta expresión 1. Esta expresión se inicializa con algún parámetro que controla la repetición del bucle. La expresión expresión 2 es una condición que debe ser cierta para que se ejecute sentencia. La expresión expresión 3 se utiliza para modificar el valor del parámetro. El bucle se repite mientras expresión 2 no sea 0 (falso). Si sentencia es compuesta, se encierra entre { }. Las expresiones expresión 1 y expresión 3 se pueden omitir. Si se omite expresión 2, se asumirá el valor permanente de 1 (cierto) y el bucle se ejecutará de forma indefinida. El diagrama de flujo, esta sentencia es el que se muestra en la Figura A.4.

El siguiente programa imprime los cien primeros números naturales.

```
#include <stdio.h>
main()
{
    int numero;
    for (numero=0; numero <100; numero++)
        printf("%d\n", numero);
}
```

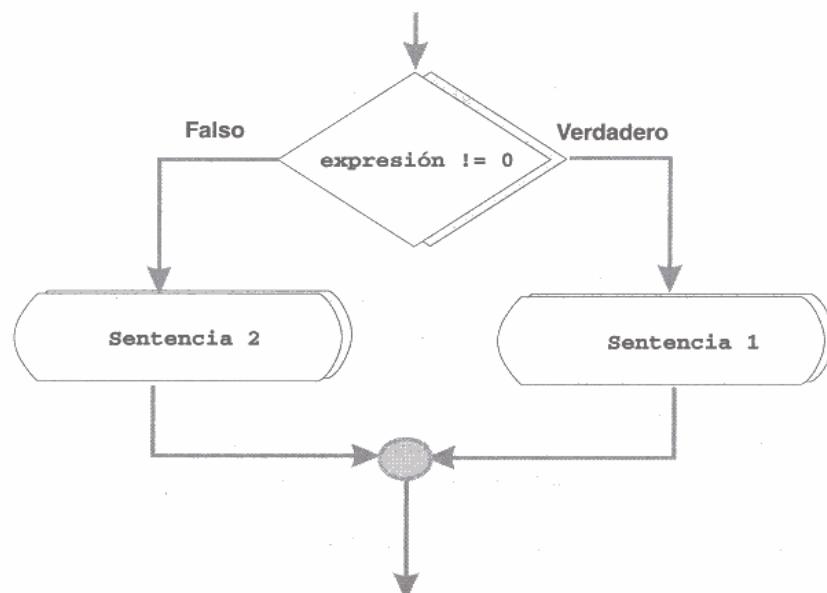


Figura A.3. Diagrama de flujo de la sentencia `if-else`.

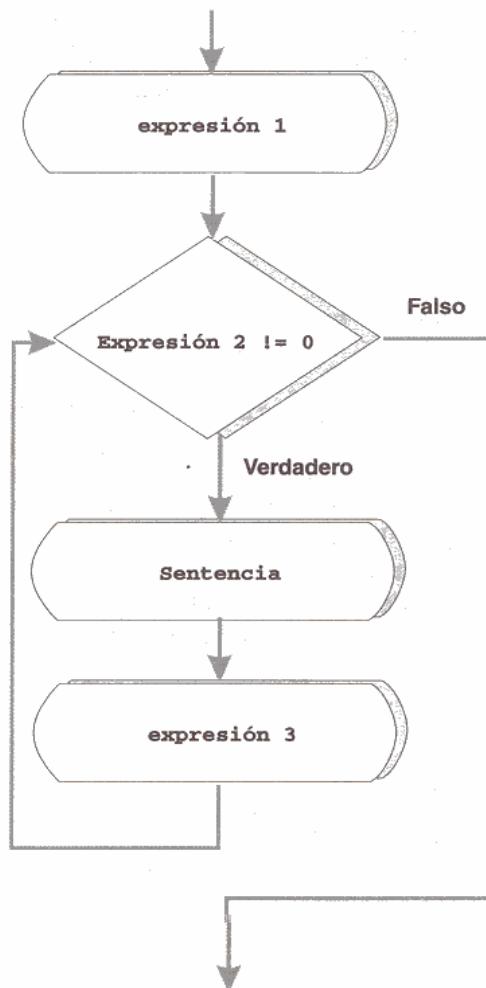


Figura A.4. Diagrama de flujo de la sentencia `for`.

A.5.4. Sentencia `while`

Su sintaxis general es la siguiente:

```
while (expresión)
    sentencia;
```

La sentencia `sentencia` se ejecutará mientras el valor de `expresión` sea verdadero (distinto de 0). En la sentencia anterior, primero se evalúa `expresión`. Lo normal es que `sentencia` incluya algún elemento que altere el valor de `expresión`, proporcionando así la condición de salida del bucle. Si la sentencia es compuesta, se encierra entre {} de la siguiente forma:

```
while (expresión)
{
```

```

sentencia 1
sentencia 2
.
.
.
sentencia N
}

```

El diagrama de flujo de la sentencia while es el que se muestra en la Figura A.5.
El siguiente programa lee un número N y calcula $1 + 2 + 3 + \dots + N$

```

#include <stdio.h>
main()
{
    int N;
    int suma = 0;

    /* leer el numero N */
    printf("N: ");
    scanf("%d", &N);
    while (N > 0)
    {
        suma = suma + N;
        N = N - 1; /* equivalente a N-- */
    }
    printf("1 + 2 +...+ N = %d\n", suma);
}

```

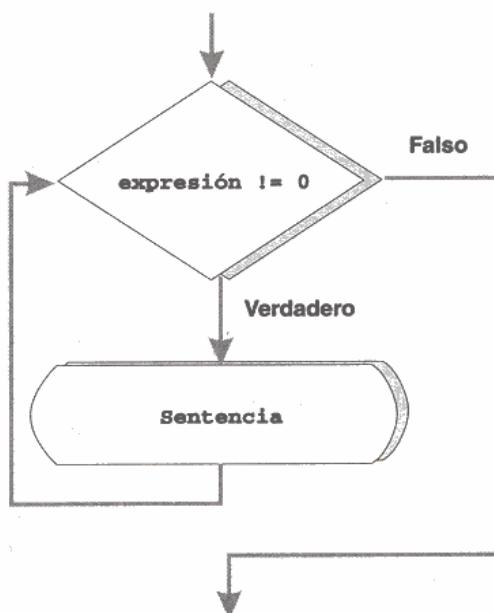


Figura A.5. Diagrama de flujo de la sentencia while.

A.5.5. Sentencia do-while

Su forma general es la siguiente:

```
do
    sentencia
  while (expresion);
```

La sentencia `sentencia` se ejecutará mientras el valor de `expresión` sea verdadero (distinto de 0). En este tipo de bucles, `sentencia` siempre se ejecuta al menos una vez (diferente a `while`). Lo normal es que `sentencia` incluya algún elemento que altere el valor de `expresión`, proporcionando así la condición de salida del bucle. Si la sentencia es compuesta, se encierra entre {} de la siguiente manera:

```
do
{
    sentencia 1
    sentencia 2
    .
    .
    .
    sentencia N
}while (expresion);
```

Para la mayoría de las aplicaciones, es mejor y más natural comprobar la condición antes de ejecutar el bucle (bucle `while`). El diagrama de flujo de esta sentencia es el que se muestra en la Figura A.6.

El siguiente programa lee de forma repetida un número e indica si es par o impar. El programa se repite mientras `numero` sea distinto de 0.

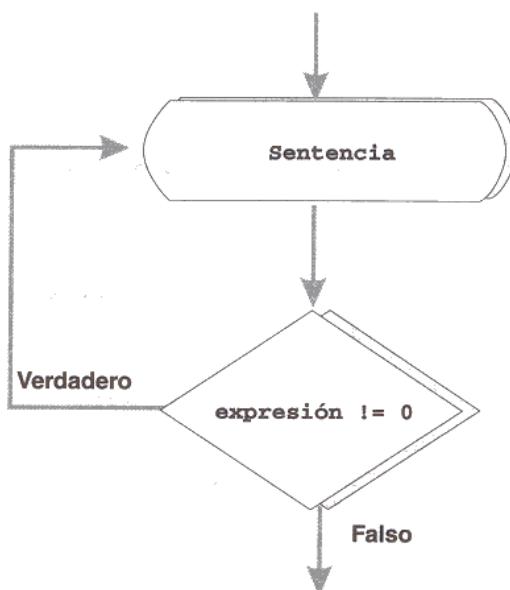


Figura A.6. Diagrama de flujo de la sentencia `do-while`.

```
#include <stdio.h>
main()
{
    int numero;
    do
    {
        /* se lee el numero */
        printf("Introduzca un numero: ");
        scanf("%d", &numero);
        if ((numero % 2) == 0)
            printf("El numero %d es par.\n", numero);
        else
            printf("El numero %d es par.\n", numero);
    } while (numero != 0)
}
```

A.5.6. Sentencia switch

Su forma general es la siguiente:

```
switch (expresion)
{
    case exp 1:
        sentencia 1;
        sentencia 2;
        .
        .
        .
        break;
    case exp N:
    case exp M:
        sentencia N1;
        sentencia N2;
        .
        .
        .
        break;
    default:
        sentencia D;
        .
        .
        .
}
```

La expresión `expresion` devuelve un valor entero (también puede ser de tipo `char`). Las expresiones `exp 1, ..., expM` representan expresiones constantes de valores enteros (también caracteres). El diagrama de flujo de este tipo de sentencias se muestra en la Figura A.7.

A.5.7. Bucles anidados

Los bucles se pueden *anidar* unos en otros. Se pueden anidar diferentes tipos de bucles. Es importante estructurarlos de forma correcta. El siguiente programa calcula $1 + 2 + \dots + N$, mientras N sea distinto de 0.

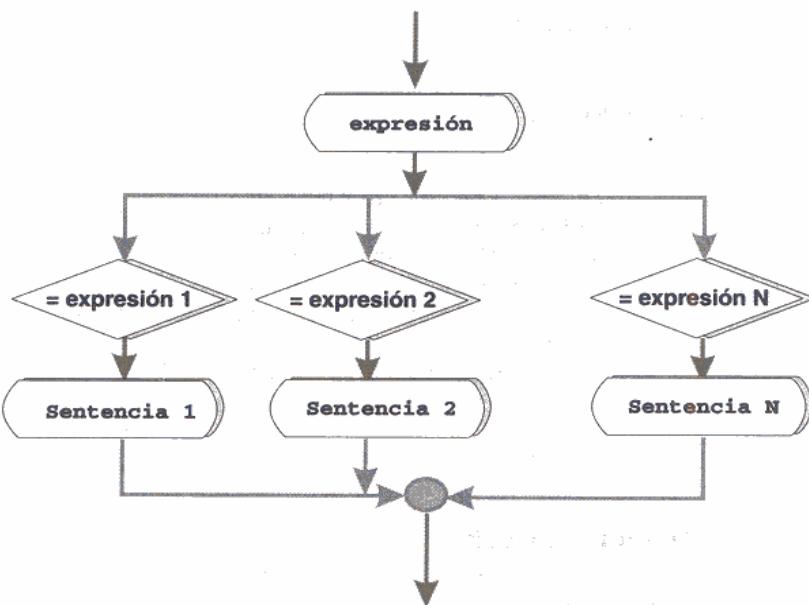


Figura A.7. Diagrama de flujo de la sentencia switch.

```

#include <stdio.h>
main()
{
    int N;
    int suma;
    int j;
    do
    {
        /* leer el numero N */
        printf("Introduzca N: ");
        scanf("%d", &N);
        suma = 0;
        for (j = 0; j <= N; j++) /* bucle anidado */
            suma = suma + j;
        printf("1 + 2 + ... + N = %d\n", suma);
    } while (N > 0); /* fin del bucle do */
}
  
```

A.5.8. Sentencias break y continue

La sentencia **break** se utiliza para terminar la ejecución de bucles o salir de una sentencia **switch**. Es necesaria en la sentencia **switch** para transferir el control fuera de la misma. En caso de bucles anidados, el control se transfiere fuera de la sentencia más interna en la que se encuentre, pero no fuera de las externas. No es aconsejable el uso de esta sentencia en los bucles, puesto que su uso es contrario a las reglas de la programación estructurada. Sin embargo, puede ser útil cuando se detectan errores o condiciones anormales. La sentencia **continue**, cuando se ejecuta dentro de un bucle, transfiere el control directamente a la expresión de evaluación del bucle (véase la Figura A.8).

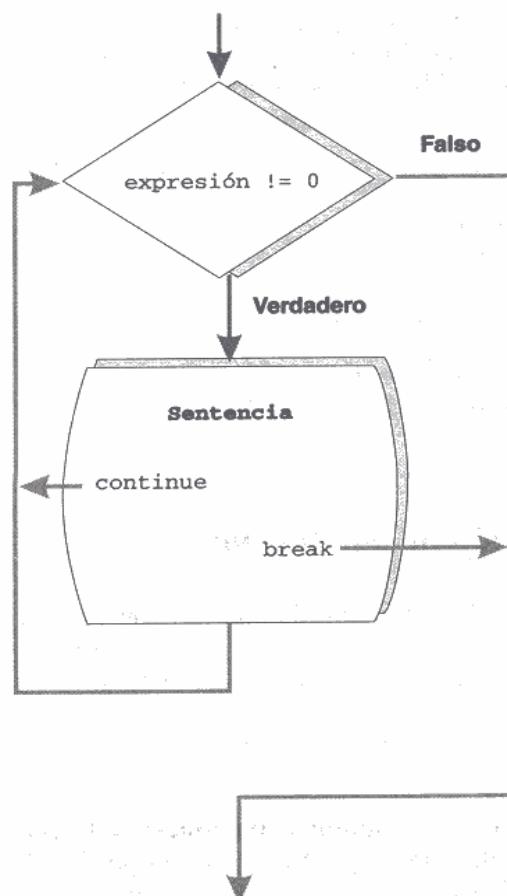


Figura A.8. *Sentencias break y continue.*

A.6. FUNCIONES

Una función es un segmento de programa que realiza una determinada tarea. Todo programa C consta de una o más funciones. Una de estas funciones se debe llamar `main()`. Todo programa comienza su ejecución en la función `main()`. El uso de funciones permite la descomposición y desarrollo modular. Permite dividir un programa en componentes más pequeños: **funciones**. El siguiente programa calcula el máximo de dos números utilizando la función `maximo`.

```

#include <stdio.h>

int maximo(int a, int b); /* prototipo de función */

main()
{
    int x, y;
    int max;

    printf("Introduzca dos numeros: ");
  
```

```

        scanf("%d %d", &x, &y);
        max = maximo(x,y); /* llamada a la funcion */
        printf("El maximo es %d\n", max);
    }

int maximo(int a, int b) /* definicion de la funcion */
{
    int max;

    if (a > b)
        max = a;
    else
        max = b;
    return(max);      /* devuelve el valor maximo */
}

```

A.6.1. Definición de una función

El formato general de definición de una función es el siguiente:

```

tipo nombre(tipo1 arg1, ..., tipoN argN)
{
    /* CUERPO DE LA FUNCION */
}

```

Los argumentos se denominan **parámetros formales**. Como puede apreciarse, la función devuelve un valor de tipo **tipo**. Si se omite tipo, se considera que devuelve un **int**. Cuando la función no devuelve ningún tipo, se indica con **void**. De igual manera, si la función no tiene argumentos, se colocará **void** entre los paréntesis, tal y como se muestra en el siguiente ejemplo:

```
void explicacion(void)
```

Entre llaves se encuentra el *cuerpo* de la función (igual que `main()`), que incluye la sentencia de la función. La sentencia `return` finaliza la ejecución y devuelve un valor a la función que realizó la llamada.

```
    return(expresion);
```

En C hay que tener en cuenta dos aspectos relacionados con las funciones:

1. Una función sólo puede devolver un valor.
2. En C no se pueden anidar funciones.

A.6.2. Declaración de funciones: prototipos

La forma general de declarar una función es la siguiente:

```
tipo nombre(tipo1 arg1, ..., tipoN argN);
```

La *declaración* o *prototipo* de una función especifica el nombre de la función, el valor que devuelve y los tipos de parámetros que acepta. Esta declaración finaliza con ; y no incluye el cuerpo de la función. No es obligatorio declarar un función, pero sí aconsejable, ya que permite la comprobación de errores entre las llamadas a una función y la definición de la función correspondiente.

La siguiente función calcula x elevado a y (con y entero).

```
float potencia (float x, int y); /* prototipo */

float potencia (float x, int y) /* definicion */
{
    int i;
    float prod = 1;

    for (i = 0; i < y; i++)
        prod = prod * x;
    return(prod);
}
```

A.6.3. Llamadas a funciones

Para llamar a una función se especifica su nombre y la lista de argumentos. Por ejemplo:

```
maximo(2, 3);
```

Se denominan **parámetros formales** a los que aparecen en la definición de la función y **parámetros reales** a los que se pasan en la llamada a la función. En una llamada a una función habrá un argumento real por cada argumento formal. Los parámetros reales pueden ser:

- Constantes.
- Variables simples.
- Expresiones complejas

Pero deben ser del mismo tipo de datos que el argumento formal correspondiente. Cuando se pasa un valor a una función se copia el argumento real en el argumento formal. Se puede modificar el argumento formal dentro de la función, pero el valor del argumento real no cambia. A este tipo de paso de parámetros se le denomina **paso de argumentos por valor**.

El proceso de llamada a una función se puede apreciar en la Figura A.9.

A.6.4. Recursividad

Una función **recursiva** es aquella que se llama a sí misma de forma repetida hasta que se cumpla alguna condición. Un ejemplo de función recursiva es el factorial de un número, cuya definición se muestra a continuación.

$$\begin{aligned} f(n) &= 1 && \text{si } n = 0 \\ &= n * f(n - 1) && \text{si } n > 0 \end{aligned}$$

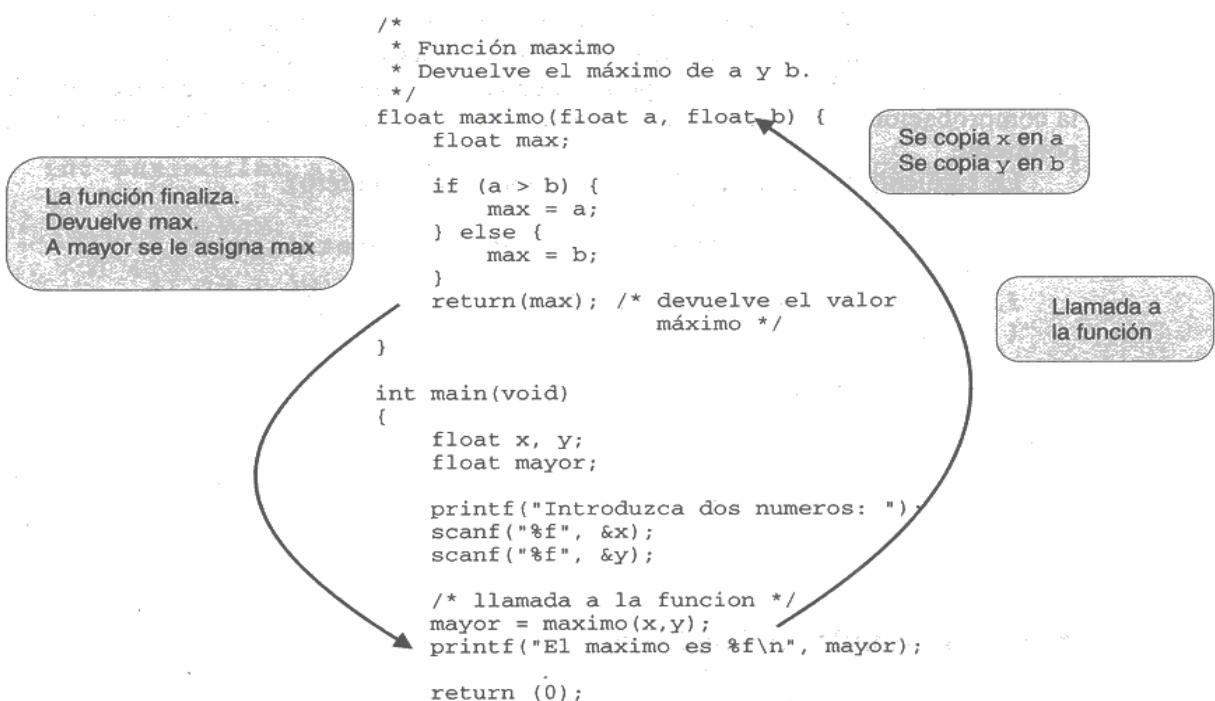


Figura A.9. Proceso de llamada a una función.

El siguiente fragmento de código incluye una función en C que calcula el factorial de forma recursiva.

```

long int factorial(int n)
{
    if (n <= 1)
        return(1);
    else
        return(n * factorial(n-1));
}

```

A.6.5. Macros

La directiva `#define` se puede utilizar para definir macros. Una **macro** es un identificador equivalente a una expresión, sentencia o grupo de sentencias. El siguiente programa calcula el máximo de dos números utilizando una macro denominada `maximo`.

```

#include <stdio.h>

#define maximo(a,b) ((a) > (b) ? (a) : (b))

main()
{
    int x, y;

```

```

int max;

printf("Introduzca dos numeros: ");
scanf("%d %d", &x, &y);
max = maximo(x,y); /* uso de la macro */
printf("El maximo es %d\n", max);
}

```

El preprocesador sustituye todas las referencias a la macro que aparezcan dentro de un programa antes de realizar la compilación, por lo que es importante que no haya blancos entre el identificador y el paréntesis izquierdo. Así, el código anterior se sustituye por el siguiente:

```

main()
{
    int x, y;
    int max;

    printf("Introduzca dos numeros: ");
    scanf("%d %d", &x, &y);
    max = ((x > y) ? x : y);
    printf("El maximo es %d\n", max);
}

```

Cuando se utiliza una macro no se realiza ninguna llamada a función; por tanto, el programa ejecuta con mayor velocidad. Sin embargo, se repite el código en cada uso de la macro, con lo que se obtiene un código objeto más grande. Hay que tener siempre en mente que una macro no es una llamada a función.

Dada la siguiente definición de macro:

```
#define maximo (x,y,z) if (x > y) \
                           z = x; \
                     else \
                           z = y;
```

Cuando el preprocesador encuentra:

```
maximo(a, b, max);
```

Lo sustituiría por:

```
if ( a > b ) max = a ; else max = b ; ;
```

Esta sentencia es equivalente a la siguiente:

```
if (a > b)
    max = a;
else
    max = b;
```

A.7. PUNTEROS

Para entender el concepto de puntero es necesario conocer cómo se encuentra organizada la memoria de la computadora. La memoria de la computadora se encuentra organizada en grupos de

bytes que se denominan **palabras** (véase la Figura A.10). Dentro de la memoria cada dato ocupa un número determinado de bytes: un char, 1 byte; un int, 4 bytes.

A cada byte o palabra se accede por su dirección. Si *x* es una variable que representa un determinado dato, el compilador reservará los bytes necesarios para representar *x* (4 bytes si es de tipo int).

Si *x* es una variable, *&x* representa la dirección de memoria de *x*. Al operador *&* se le denomina **operador de dirección**. Un **puntero** es una variable que almacena la dirección de otro objeto (variable, función, . . .). El siguiente ejemplo presenta el uso de punteros.

```
#include <stdio.h>
main()
{
    int x; /* variable de tipo entero */
    int y; /* variable de tipo entero */
    int *px; /* variable de tipo puntero a entero */

    x = 5;
    px = &x; /* asigna a px la dirección de x */
    y = *px; /* asigna a y el contenido de la
               dirección almacenada en px */
    printf("x = %d\n", x);
    printf("y = %d\n", y);
    printf("*px = %d\n", *px);
}
```

La expresión **px* representa el contenido almacenado en la dirección a la que apunta *px*. El operador *** se denomina **operador de indirección** y opera sobre una variable de tipo puntero. Es importante recordar que un puntero representa la dirección de memoria del objeto al que apunta, **NO** su valor.

A.7.1. Definición de punteros

Los punteros, al igual que el resto de variables, deben definirse antes de usarlos. La definición de una variable de tipo puntero se realiza de la siguiente forma:

```
tipo_dato *variable_ptr;
```

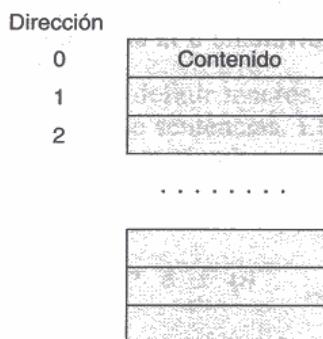


Figura A.10. Organización de la memoria del computador.

El identificador `variable_ptr` es el nombre de la variable puntero y `tipo_dato` se refiere al tipo de dato apuntado por el puntero. La variable `variable_ptr` sólo debe almacenar la dirección de variables de tipo `tipo_dato`. Para usar un puntero se debe estar seguro de que apunta a una dirección de memoria correcta. Es importante recordar que un puntero no reserva memoria. La Figura A.11 ilustra el uso de punteros.

A.7.2. Paso de punteros a una función

Cuando se pasa un puntero a una función se pasa la dirección del dato al que apunta. El uso de punteros permite emular el concepto de paso de argumentos por **referencia**. Cuando un argumento se pasa por valor, el dato se *copia* a la función. Esto quiere decir que un argumento pasado por valor no se puede modificar dentro de la función. Cuando se pasa un argumento por *referencia* (cuando un puntero se pasa a una función) se pasa la *dirección* del dato, lo que implica que el contenido de la dirección se puede modificar en la función. Por tanto, los argumentos pasados por referencia sí se pueden modificar. El uso de punteros como argumentos de funciones permite que el dato sea alterado dentro de la función. La Figura A.9 ilustraba el paso de parámetros por valor.

El siguiente ejemplo muestra cómo se realiza el paso de parámetros por referencia en C.

```
#include <stdio.h>
void funcion(int *a, int *b); /* prototipo */
main()
{
    int x = 2;
    int y = 5;
    printf("Antes x = %d, y = %d\n", x, y);
    funcion(&x, &y);
    printf("Despues x = %d, y = %d\n", x, y);
}
void funcion(int *a, int *b)
{
    *a = 0;
    *b = 0;
    printf("Dentro *a = %d, *b = %d\n", *a, *b);
    return;
}
```

En la Figura A.12 se ilustra el paso de parámetros por referencia.

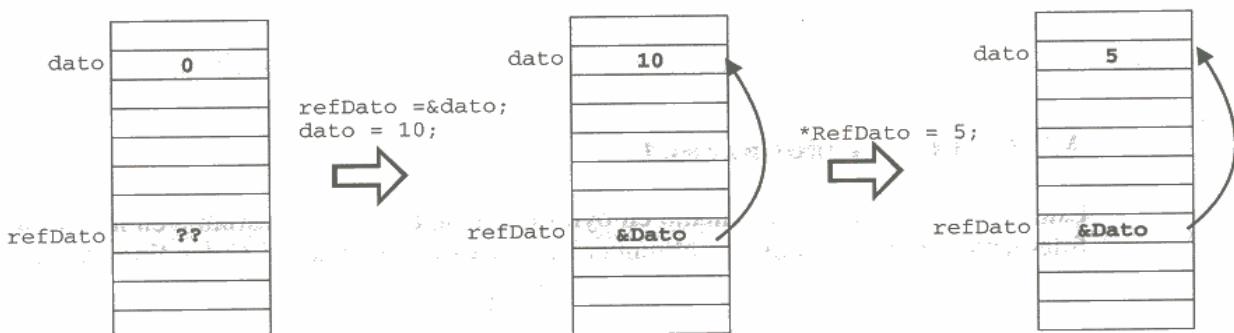


Figura A.11. Uso de punteros.

```

#include <stdio.h>

void incrementar(int *a);

int main(void)
{
    int x = 5;

    printf("Antes de la función: %d\n", x);
    incrementar(&x);
    printf("Después de la función: %d\n", x);
    return;
}

void incrementar(int *a)
{
    *a = *a + 1;

    printf("    Dentro de la función: %d\n", *x);
    return;
}

```

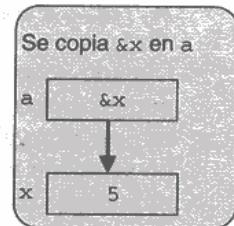


Figura A.12. Paso de parámetros por referencia.

A.7.2.1. Puntero NULL

Cuando se asigna **0** a un puntero, éste no apunta a ningún objeto o función. La constante simbólica **NULL** definida en **stdio.h** tiene el valor **0** y representa el puntero nulo. Es una buena técnica de programación asegurarse de que todos los punteros toman el valor **NULL** cuando no apuntan a ningún objeto o función.

```
int *p = NULL;
```

Para ver si un puntero no apunta a ningún objeto o función, se puede utilizar el siguiente fragmento de código:

```

if (p == NULL)
    printf("El puntero es nulo\n");
else
    printf("El contenido de *p es\n", *p);

```

A.7.3. El operador sizeof

Este operador devuelve el tamaño en bytes que ocupa un tipo o variable en memoria. El siguiente fragmento de código imprime el número de bytes que ocupan distintos tipos de datos.

```

#include <stdio.h>
main()
{

```

```

    float num;
    printf("Un int ocupa %d bytes\n", sizeof(int));
    printf("Un char ocupa %d bytes\n", sizeof(char));
    printf("Un float ocupa %d bytes\n", sizeof(float));
    printf("Un double ocupa %d bytes\n", sizeof(double));
    printf("Un ocupa %d bytes\n", sizeof(num));
}

```

A.8. ÁMBITO DE LAS VARIABLES Y TIPOS DE ALMACENAMIENTO

Existen dos formas de caracterizar una variable:

- Por su *tipo de datos*.
- Por su *tipo de almacenamiento*.

El tipo de datos se refiere al tipo de información que representa la variable (*int*, *char*, ...). El tipo de almacenamiento se refiere a su permanencia y a su *ámbito*. El **ámbito** de una variable es la porción del programa en la cual se reconoce la variable. Según el *ámbito*, las variables pueden ser:

- Variables locales.
- Variables globales.

Según el tipo, las variables pueden ser:

- Variables automáticas.
- Variables estáticas.
- Variables externas.
- Variables de tipo registro.

A.8.1. Variables locales

Las variables locales sólo se reconocen dentro de la función donde se definen. Son invisibles al resto. Una variable local normalmente no conserva su valor una vez que el control del programa se transfiere fuera de la función. Así, en el siguiente ejemplo, la variable local *a* de la función *main* es distinta a la variable *a* de la función *funcion1*.

```

#include <stdio.h>
void funcion1(void);
main()
{
    int a = 1; /* variable local */
    int b = 2; /* variable local */
    funcion1();
    printf("a = %d, b = %d \n", a, b);
}
void funcion1(void)
{

```

```

int a = 3; /* variable local */
int c = 4; /* variable local */
printf("a = %d, c = %d \n", a, c);
return;
}

```

A.8.2. Variables globales

Una variable global se declara fuera de las funciones y antes de su uso y puede ser accedida desde cualquier función. Así, en el siguiente ejemplo a es una variable global y puede accederse desde la función main y la función funcion1.

```

#include <stdio.h>
void funcion1(void);
int a = 1000; /* variable global */
main()
{
    int b = 2; /* variable local */
    funcion1();
    printf("a = %d, b = %d \n", a, b);
}
void funcion1(void)
{
    int c = 4; /* variable local */
    printf("a = %d, c = %d \n", a, c);
    return;
}

```

Las variables globales mantienen los valores que se les asignan en las funciones. El uso de variables globales puede causar errores inesperados. Cualquier función puede cambiar el valor de una variable global, lo que puede provocar efectos secundarios o laterales. Como normal general, deberían seguirse las siguientes recomendaciones:

- Evitar el uso de variables globales.
- Mantener las variables lo más locales que se pueda.
- Cuando se precise hacer accesible el valor de una variable a una función se pasará como argumento.

A.8.3. Variables automáticas

Las variables automáticas son las variables locales que se definen en las funciones. Su ámbito es local y su vida se restringe al tiempo en el que está activa la función. Los parámetros formales se tratan como variables automáticas. Este tipo de variables se pueden especificar con la palabra reservada auto, aunque no es necesario.

```

#include <stdio.h>
main()
{
    auto int valor; /* equivalente a int valor */
}

```

```

        valor = 5;
        printf("El valor es %d\n", valor);
    }
}

```

A.8.4. Variables estáticas

El ámbito de una variable estática es local a la función en la que se define; sin embargo, su vida coincide con la del programa, por lo que retienen sus valores durante toda la vida del programa. Se especifican con la palabra reservada `static`. El siguiente ejemplo hace uso de una variable de tipo `static` para contabilizar el número de veces que se invoca a una función.

```

#include <stdio.h>
void funcion(void);
main()
{
    funcion();
    funcion();
    funcion();
}
void funcion(void)
{
    static int veces = 0;
    veces = veces + 1;
    printf("Se ha llamado %d veces a funcion\n",veces);
}

```

A.8.5. Variables de tipo registro

Este tipo de variables informa al compilador que el programador desea que la variable se almacene en un lugar de rápido acceso, generalmente en los registros del computador. Si no existen registros disponibles, se almacenará en memoria. Este tipo de variables se especifica con la palabra reservada `register`, como se muestra en el siguiente ejemplo.

```

#include <stdio.h>
main()
{
    register int j;

    for(j = 0; j < 10; j++)
        printf("Contador = %d\n", j);
}

```

A.8.6. Variables externas

Una variable *extena* es una variable global. Hay que distinguir entre *definición* y *declaración* de variable externa. Una **definición** se escribe de la misma forma que las variables normales y reserva espacio para la misma en memoria. Una **declaración** no reserva espacio de almacenamiento. Se especifica con `extern`. Este tipo de variables se emplea cuando un programa consta de varios

módulos, de tal manera que en uno de ellos se define la variable y en los demás se declara con la palabra reservada (`extern`). Si se declarara en todos los módulos, el enlazador generaría un error al obtener el ejecutable indicando que se está redefiniendo múltiples veces una variable.

Considere un programa compuesto por dos módulos. El módulo principal que contiene la función `main` (almacenado en el archivo `main.c`) es el siguiente:

```
#include <stdio.h>
extern int valor; /* se declara */
void funcion(void);
main()
{
    funcion();
    printf("Valor = %d\n", valor);
}
```

El módulo auxiliar (`aux.c`) es el siguiente:

```
int valor; /* se define la variable */
void funcion(void)
{
    valor = 10;
}
```

Como puede verse, ambos módulos utilizan la variable `valor` de tipo `int`. La variable se define en el módulo auxiliar (se reserva memoria para ella) y se declara (se utiliza) en los dos. Para obtener el ejecutable, se compilan los módulos por separado de la siguiente forma:

```
gcc -c -Wall main.c
gcc -c -Wall aux.c
```

Estos dos mandatos obtienen dos módulos objetos: `main.o` y `aux.o`. El ejecutable (`prog`) se genera de la siguiente forma:

```
gcc main.o aux.o -o prog
```

A.9. CADENAS DE CARACTERES

Una **cadena de caracteres** es un conjunto o vector de caracteres. La constante '`a`' representa un carácter individual. "`Hola`" representa una cadena de caracteres. "`a`" representa una cadena de caracteres compuesta por un único carácter.

Todas las cadenas de caracteres en C finalizan con el **carácter nulo** de C ('`\0`'). Este carácter indica el fin de una cadena de caracteres. La cadena "`Hola`" se almacena en memoria y su longitud es 4 (no se incluye el carácter nulo). La Figura A.13 ilustra la disposición de la cadena "`Hola`" en memoria.

A.9.1. Definición de cadenas de caracteres

La siguiente definición:

```
char cadena[] = "Hola";
```

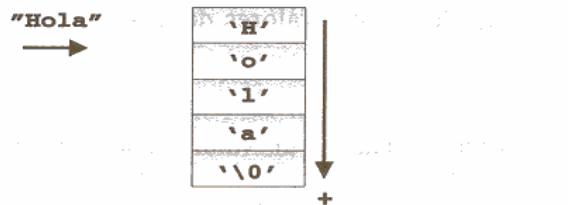


Figura A.13. Disposición de una cadena de caracteres en memoria.

Define una cadena denominada `cadena` y reserva espacio para almacenar los siguientes caracteres:

```
'H' 'o' 'l' 'a' '\0'
```

El siguiente programa muestra cómo leer una cadena y acceder a sus caracteres individuales.

```
#include <stdio.h>
main()
{
    char cadena[] = "Hola";
    printf("La cadena es %s \n", cadena);
    printf("Los caracteres son: \n");
    printf("%c \n", cadena[0]);
    printf("%c \n", cadena[1]);
    printf("%c \n", cadena[2]);
    printf("%c \n", cadena[3]);
    printf("%c \n", cadena[4]);
}
```

`cadena[i]` representa el i -ésimo carácter de la cadena. El primer carácter se encuentra en el índice 0, como ocurre con el resto de vectores.

La definición:

```
char cadena[80];
```

Define una cadena de caracteres, denominada `cadena`, compuesta por ochenta caracteres, incluido el carácter nulo. La definición:

```
char cadena[4] = "Hola";
```

Define una cadena de exactamente cuatro caracteres que no incluye el carácter nulo, por lo que no se tratará de forma correcta como una cadena de caracteres.

A.9.2. Asignación de valores a cadenas de caracteres

La asignación de valores iniciales a una cadena se puede realizar en su declaración:

```
char cadena[5] = "Hola";
char cadena[10] = "Hola";
```

No se puede asignar valores de la siguiente forma:

```
cadena = "Hola";
```

Una forma de asignar un valor a una cadena es la siguiente:

```
strcpy(cadena, "Hola");
```

cadena debe tener suficiente espacio reservado. La función de biblioteca strcpy copia cadena1 en cadena2, incluyendo el carácter nulo. La función strcpy se encuentra en el archivo de cabecera string.h.

A.9.3. Lectura y escritura de cadenas de caracteres

El especificador de formato que se emplea en printf y scanf para imprimir y leer una cadena de caracteres, respectivamente, es %s. El siguiente programa muestra su utilización:

```
#include <stdio.h>
#define TAM_CADENA 80
main()
{
    char cadena[TAM_CADENA];
    printf("Introduzca una cadena: ");
    scanf("%s", cadena);
    printf("La cadena es %s\n", cadena);
}
```

scanf deja de buscar cuando encuentra un blanco. Por tanto, si se introduce Hola a todos sólo se leerá Hola. No es necesario el operador de dirección (&), ya que cadena representa de forma automática la dirección de comienzo.

La función gets lee una línea completa hasta que encuentra el retorno de carro, incluyendo los blancos. La función puts escribe una cadena de caracteres junto con un salto de línea.

La sentencia:

```
puts("La linea es:");
```

es equivalente a:

```
printf("La linea es: \n");
```

A.9.4. Paso de cadenas de caracteres a funciones

Cuando se pasa una cadena a una función, se pasa la dirección de comienzo de la misma. Por tanto, la cadena se puede modificar en la función. Considere el siguiente ejemplo:

```
#include <stdio.h>
#define TAM_LINEA 80
```

```

void leer_linea(char linea[]);
main()
{
    char linea[TAM_LINEA];
    leer_linea(linea);
    puts("La linea es");
    puts(linea);
}

void leer_linea(char linea[])
{
    gets(linea);
    return;
}

```

A.9.5. Funciones de biblioteca para manejar cadenas

En `<string.h>`

Función	Significado
<code>strcpy</code>	Copia una cadena en otra
<code>strlen</code>	Longitud de la cadena
<code>strcat</code>	Concatenación de cadenas
<code>strcmp</code>	Comparación de dos cadenas
<code>strchr</code>	Buscar un carácter dentro de una cadena
<code>strstr</code>	Buscar una cadena dentro de otra

En `<stdlib.h>`

Función	Significado
<code>atoi</code>	Convierte una cadena a un entero (<code>int</code>)
<code>atol</code>	Convierte una cadena a un entero largo (<code>long</code>)
<code>atof</code>	Convierte una cadena a un real (<code>double</code>)

A.10. VECTORES Y MATRICES

Un vector o *array* es un conjunto de valores, todos del mismo tipo, a los que se da un nombre común, distinguiendo cada uno de ellos por su índice. Coincide con el concepto matemático de vector:

$$V = (V_0, V_1, \dots, V_n)$$

El número de índices determina la dimensión del vector. En C, los datos individuales de un vector pueden ser de cualquier tipo (`int`, `char`, `float`, etc.).

A.10.1. Definición de un vector

Un vector unidimensional se define de la siguiente forma:

```
tipo_dato vector[expresion];
```

donde

- `tipo_dato` es el tipo de datos de cada elemento.
- `vector` es el nombre del vector (*array*).
- `expresion` indica el número de elementos del vector.

Ejemplos:

```
int v_numeros[20];
float n[12];
char vector_letras[5];
```

En C, el primer índice del vector es **0**. Sólo se puede asignar valores iniciales a vectores estáticos y globales.

```
int n[5] = {1, 2, 18, 24, 3};
```

A.10.2. Procesamiento de un vector

En C no se permiten operaciones que impliquen vectores completos. Esto quiere decir que:

- No se pueden asignar vectores completos.
- No se pueden comparar vectores completos.

El procesamiento debe realizarse elemento a elemento, tal y como se muestra en el siguiente programa:

```
#include <stdio.h>
#define TAM_VECTOR 10
main()
{
    int vector_a[TAM_VECTOR];
    int vector_b[TAM_VECTOR];
    int j; /* variable utilizada como índice */
    /* leer el vector a */
    for (j = 0; j < TAM_VECTOR; j++)
    {
        printf("Elemento %d: ", j);
        scanf("%d", &vector_a[j]);
    }
    /* copiar el vector */
    for (j = 0; j < TAM_VECTOR; j++)
        vector_b[j] = vector_a[j];
    /* escribir el vector b */
}
```

```

        for (j = 0; j < TAM_VECTOR; j++)
            printf("El elemento %d es %d \n", j, vector_b[j]);
    }

```

A.10.3. Paso de vectores a funciones

Un vector se pasa a una función especificando su nombre sin corchetes. El nombre representa la dirección del primer elemento del vector. Los vectores se pasan por **referencia** y se pueden modificar en las funciones. El argumento formal correspondiente al vector se escribe con un par de corchetes cuadrados vacíos. El tamaño no se especifica. A continuación se muestra un programa que calcula la media de los componentes de un vector:

```

#include <stdio.h>
#define MAX_TAM 4

void leer_vector(int vector[]);
int media_vector(int vector[]);
main()
{
    int v_numeros[MAX_TAM];
    int media;
    leer_vector(v_numeros);
    media = media_vector(v_numeros);
    printf("La media es %d\n", media);
}
void leer_vector(int vector[])
{
    int j;
    for(j=0; j<MAX_TAM; j++)
    {
        printf("Elemento %d: ", j);
        scanf("%d", &vector[j]);
    }
    return;
}
int media_vector(int vector[])
{
    int j;
    int media = 0;
    for(j=0; j<MAX_TAM; j++)
        media = media + vector[j];
    return(media/MAX_TAM);
}

```

A.10.4. Punteros y vectores

El nombre del vector representa la dirección del primer elemento del vector, es decir, dada la siguiente definición:

```
float vector[MAX_TAM];
```

se cumple lo siguiente:

```
vector == &vector[0]
```

El nombre del vector es realmente un puntero al primer elemento del vector. Por tanto:

$\&x[0]$ es equivalente a x

$\&x[1]$ es equivalente a $(x+1)$

$\&x[2]$ es equivalente a $(x+2)$

$\&x[i]$ es equivalente a $(x+i)$

Es decir, $\&x[i]$ y $(x+i)$ representan la dirección del i -ésimo elemento del vector x . Por tanto, $x[i]$ y $*(x+i)$ representan el contenido del i -ésimo elemento del vector x . Cuando un vector se define como un puntero no se le pueden asignar valores, ya que un puntero no reserva espacio en memoria.

`float x[10]` define un vector compuesto por diez números reales. Reserva espacio para los elementos. `float *x` declara un puntero a `float`. Si se quiere que `float x` se comporte como un vector, entonces habrá que reservar memoria para los diez elementos:

```
x = (float *) malloc(10 * sizeof(float));
```

La función de biblioteca `malloc(nb)` (`stdlib.h`) reserva un bloque de memoria de `nb` bytes. Para liberar la memoria asignada se utiliza `free()` (`stdlib.h`)

```
free(x);
```

El uso de punteros permite definir vectores de forma dinámica.

A.10.5. Vectores y cadenas de caracteres

Una cadena de caracteres es un vector de caracteres. Cada elemento del vector almacena un carácter. La siguiente función copia una cadena en otra:

```
void copiar(char *destino, char *fuente)
{
    while (*fuente != '\0')
    {
        *destino = *fuente;
        destino++;
        fuente++;
    }
    *destino = '\0';
    return;
}
```

A.10.6. Vectores multidimensionales

Un vector multidimensional se define de la siguiente forma:

```
tipo_dato vector[exp1] [exp2] ... [expN];
```

La sentencia:

```
int matriz[20][30];
```

define una matriz de veinte filas y treinta columnas.

El elemento de la fila i columna j es `matriz[i][j]`. No confunda esta expresión con esta otra: `matriz[i, j]`.

A.11. ESTRUCTURAS

Una estructura está compuesta de elementos individuales que pueden ser de distinto tipo. Cada uno de los elementos de una estructura se denomina **miembro**. Una estructura se define de la siguiente manera:

```
struct nombre_estructura
{
    tipo1 miembro_1;
    tipo2 miembro_2;
    .
    .
    .
    tipoN miembro_N;
};
```

Los miembros pueden ser de cualquier tipo, excepto `void`.

Por ejemplo:

```
struct fecha
{
    int mes;
    int dia;
    int anno;
};
```

declara una estructura denominada `struct fecha`.

```
struct fecha fecha_de_hoy;
```

define una variable denominada `fecha de hoy` de tipo `struct fecha`.

A.11.1. Procesamiento de una estructura

Los miembros de una estructura se procesan individualmente. Para hacer referencia a un miembro determinado se indica:

```
variable_estructura.miembro
```

El operador `.` se denomina **operador de miembro**.

Para imprimir el miembro `dia` de la variable `hoy` se utiliza el siguiente fragmento de código:

```
struct fecha hoy;
printf("%d: %d: %d\n", hoy.dia,
hoy.mes, hoy.anno);
```

Se pueden copiar estructuras:

```
struct fecha hoy, ayer;
ayer = hoy;
```

Sin embargo, no se pueden comparar estructuras.

A.11.2. Paso de estructuras a funciones

Se pueden pasar miembros individuales y estructuras completas. Las estructuras se pasan por valor. Una función puede devolver una estructura. Por ejemplo, la siguiente función imprime una fecha:

```
void imprimir_fecha(struct fecha f)
{
    printf("Dia: %d\n", f.dia);
    printf("Mes: %d\n", f.mes);
    printf("Anno: %d\n", f.anno);
    return;
}
```

La siguiente función permite leer una fecha:

```
struct fecha leer_fecha(void)
{
    struct fecha f;
    printf("Dia: ");
    scanf("%d", &(f.dia));
    printf("Mes: ");
    scanf("%d", &(f.mes));
    printf("Anno: ");
    scanf("%d", &(f.anno));
    return(f);
}
```

Las funciones anteriores se pueden usar de la siguiente forma:

```
main()
{
    struct fecha fecha_de_hoy;
    fecha_de_hoy = leer_fecha();
    imprimir_fecha(fecha_de_hoy);
}
```

A.11.3. Punteros a estructuras

Se puede definir punteros a estructuras como ocurre con el resto de tipos de datos:

```

struct punto
{
    float x;
    float y;
};

main()
{
    struct punto punto_1;
    struct punto *punto_2;
    punto_1.x = 2.0;
    punto_1.y = 4.0;
    punto_2 = &punto_1;
    printf("x = %f \n", punto_2->x);
    printf("y = %f \n", punto_2->y);
}

```

En una variable de tipo puntero a una estructura, los miembros acceden con el operador `->`. Se pueden pasar punteros a funciones. De esta forma se pueden pasar variables de tipo estructura por referencia. Un puntero a una estructura no reserva memoria. Consideré el siguiente ejemplo:

```

void leer_punto(struct punto *p);
void imprimir_punto(struct punto p);
main()
{
    struct punto *p1;
    p1 = (struct punto *)malloc(sizeof(struct punto));
    leer_punto(p1);
    imprimir_punto(*p1);
    free(p1);
}
void leer_punto(struct punto *p)
{
    printf("x = ");
    scanf("%f", &(p->x));
    printf("y = ");
    scanf("%f", &(p->y));
}
void imprimir_punto(struct punto p)
{
    printf("x = %f\n", p.x);
    printf("y = %f\n", p.y);
}

```

A.11.4. Vectores de estructuras

El siguiente fragmento de código define y usa un vector de estructuras:

```

main()
{
    struct punto vector_puntos[10];
    int j;
}

```

```

        for(j = 0; j < 10; j++)
        {
            printf("x_%d = %f\n", j, vector_puntos[j].x);
            printf("y_%d = %f\n", j, vector_puntos[j].y);
        }
    }
}

```

A.11.5. Uniones

Una unión contiene miembros cuyos tipos de datos pueden ser diferentes (igual que las estructuras). Su declaración es similar a las estructuras:

```

union nombre_estructura
{
    tipo1 miembro_1;
    tipo2 miembro_2;
    .
    .
    .
    tipoN miembro_N;
};

```

Todos los miembros que componen la unión comparten la misma zona de memoria. Una variable de tipo union sólo almacena el valor de uno de sus miembros. A continuación se muestra un uso de este tipo de estructuras.

```

#include <stdio.h>
#include <stdlib.h>
union numero
{
    int entero;
    float real;
};

main()
{
    union numero num;
    /* leer un entero e imprimirllo */
    printf("Entero: ");
    scanf("%d", &(num.entero));
    printf("El entero es %d\n", num.entero);
    /* leer un real e imprimirllo */
    printf("Real: ");
    scanf("%f", &(num.real));
    printf("El entero es %f\n", num.real);
}

```

A.11.6. Tipos enumerados

Un tipo enumerado es similar a las estructuras. Sus miembros son constantes de tipo int. Se definen de la siguiente forma:

```
enum nombre {m1, m2, . . . , mN};
```

La siguiente definición define un tipo enumerado denominado `color`.

```
enum color {negro, blanco, rojo};
```

A partir de la definición anterior, se pueden definir variables de tipo `enum color`, como se muestra a continuación:

```
enum color c = blanco;
```

A.11.7. Definición de tipos de datos (`typedef`)

Un nuevo tipo se define como:

```
typedef tipo nuevo_tipo;
```

Por ejemplo,

```
typedef char letra;
typedef struct punto tipo_punto;
letra c;
tipo_punto p;
```

A.12. ENTRADA/SALIDA

En esta sección se describen brevemente las funciones de la biblioteca estándar de C para el acceso a archivos.

A.12.1. Apertura y cierre de un archivo

Para abrir un archivo se utiliza la función `fopen`:

```
desc = fopen(nombre_archivo, modo);
```

donde `desc` es el descriptor de archivo que devuelve y se declara como:

```
FILE *desc;
```

y `modo` especifica la forma de apertura del archivo. Si `fopen` devuelve `NULL`, el archivo no se pudo abrir.

Modo	Significado
«r»	Abre un archivo existente para lectura.
«w»	Abre un nuevo archivo para escritura. Si existe el archivo, se borra su contenido. Si no existe, se crea.
«a»	Abre un archivo existente para añadir datos al final. Si no existe, se crea.
«r+»	Abre un archivo existente para lectura y escritura.
«w+»	Abre un archivo nuevo para escritura y lectura. Si existe, lo borra. Si no existe, lo crea.
«a+»	Abre un archivo para leer y añadir.

Para cerrar el archivo se utiliza:

```
fclose(desc);
```

A.12.2. Lectura y escritura

Para leer de un archivo previamente abierto se utiliza:

```
fscanf(desc, formato, ...);
```

Para escribir:

```
fprintf(desc, formato, ...);
```

El siguiente programa copia un archivo en otro utilizando funciones de la biblioteca estándar de C.

```
#include <stdio.h>
main()
{
    FILE *fent;
    FILE *fsal;
    char car;

    fent = fopen("entrada.txt", "r");
    if (fent == NULL)
    {
        printf("Error abriendo entrada.txt \n");
        exit(0);
    }
    fsal = fopen("salida.txt", "w");
    if (fsal == NULL)
    {
        printf("Error creando salida.txt \n");
        close(fent);
        exit(0);
    }
    while (fscanf(fent, "%c", &car) != EOF)
        fprintf(fsal, "%c", car);

    fclose(fent);
    fclose(fsal);
    exit(0);
}
```

A.13. ASPECTOS AVANZADOS

Esta última sección describe algunos de los aspectos avanzados que ofrece C.

A.13.1. Argumentos en la línea de mandatos

Cuando se ejecuta un programa se pueden pasar argumentos a la función main desde la línea de mandatos:

```
nombre_programa arg1 arg2 arg3 ... argn
```

El prototipo de la función main es el siguiente:

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

donde

- argc indica el número de argumentos que se pasa incluido en el nombre del programa.
- argv[0] almacena el nombre del programa.
- argv[1] almacena el primer argumento que se pasa.
- argv[i] almacena el *i*-ésimo argumento que se pasa.

El siguiente programa recibe como argumento el nombre de dos archivos compuestos de números enteros y los copia.

```
copiar_enteros archivo_entrada archivo_salida
```

el programa copia archivo_entrada en archivo_salida.

```
#include <stdio.h>
main(int argc, char *argv[])
{
    FILE *fent;
    FILE *fsal;
    int num;
    if (argc != 3)
    {
        printf("Uso: %s fich_entrada fich_salida\n",
               argv[0]);
        exit(0);
    }
    fent = fopen(argv[1], "r");
    if (fent == NULL)
    {
        printf("Error abriendo entrada.txt \n");
        exit(0);
    }
    fsal = fopen(argv[2], "w");
    if (fsal == NULL)
    {
        printf("Error creando entrada.txt \n");
        close(fent);
        exit(0);
    }
    while (fscanf(fent, "%d", &num) != EOF)
        fprintf(fsal, "%d\n", num);
```

```

        fclose(fent);
        fclose(fsal);
        exit(0);
    }
}

```

A.13.2. Operadores de bits

Estos operadores permiten manejar los bits individuales en una palabra de memoria. Existen las siguientes categorías:

- Operador de complemento a uno.
- Operadores lógicos binarios.
- Operadores de desplazamiento.

A.13.2.1. Operador de complemento a uno

El operador de complemento a uno (`~`) invierte los bits de su operando, los unos se transforman en ceros y los ceros en unos. Por ejemplo, si se ejecuta el siguiente programa en una computadora de 32 bits:

```

#include <stdio.h>
main()
{
    unsigned int n = 0x4325;
    printf("%x ; %x\n", n, ~n);
}

```

se obtiene la siguiente salida:

0x4325 0xfffffbcd

A.13.2.2. Operadores lógicos binarios

Los operadores lógicos binarios de C son los siguientes:

Operador	Función
&	AND binario
	OR binario
^	OR exclusivo binario

Las operaciones se realizan de forma independiente en cada par de bits que corresponden a cada operando:

a	b	a & b	a b	a ^ b
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

El siguiente programa:

```
#include <stdio.h>
main()
{
    int a = 8;
    int b = 5;
    int c = 3;

    printf("%d\n", a & b);
    printf("%d\n", b & c);
    printf("%d\n", a | c);
    printf("%d\n", b ^ c);
}
```

imprime: 0, 1, 11, 6

A.13.3. Máscaras

El *enmascaramiento* es un proceso en el que un patrón dado de bits se convierte en otro patrón por medio de una operación lógica a nivel de bits. El segundo operando se llama **máscara**. El siguiente programa obtiene los cuatro bits menos significativos de un número dado:

```
#include <stdio.h>
#define MASCARA 0xF
main()
{
    int n;
    printf("Introduzca n: ");
    scanf("%d", &n);
    printf("bits menos signif. = %d\n", n & MASCARA );
}
```

A.13.4. Operadores de desplazamiento

Los operadores de desplazamiento son los siguientes:

- Desplazamiento a la izquierda: <<
- Desplazamiento a la derecha: >>

Requieren dos operandos. El primero es un operando de tipo entero que representa el patrón de bits a desplazar. El segundo es un entero sin signo que indica el número de desplazamientos. Por ejemplo, si $a = 8$, entonces:

Expresión	Valor
$a <<1$	16
$a <<2$	32
$a <<3$	64
$a >>3$	1

A.13.5. Campos de bits

C permite descomponer una palabra en distintos campos de bits que pueden manipularse de forma independiente. La descomposición puede realizarse declarando una estructura de la siguiente forma:

```
struct marca
{
    miembro_1 : i;
    miembro_2 : j;
    .
    .
    miembro_N : k;
};
```

Cada declaración de miembro puede incluir una especificación que indica el tamaño del campo de bits correspondiente (*i*, *j*, *k*). ISO C permite que los campos de bits sean de tipo `unsigned int`, `signed int` o `int`. La ordenación de los campos de bits puede variar de un compilador a otro. En el siguiente programa se utilizan campos de bits:

```
#include <stdio.h>
struct registro
{
    unsigned a : 2;
    unsigned b : 4;
    unsigned c : 1;
    unsigned d : 1;
};
main()
{
    struct registro r;
    r.a = 5; r.b = 2; r.c = 1; r.d = 0;
    printf("%d %d %d %d\n", r.a, r.b, r.c, r.c);
    printf("r requiere %d\n", sizeof(r));
}
```

A.13.6. Punteros a funciones

C permite definir punteros a funciones. La forma de declarar un puntero a una función es:

```
tipo_dato (*nombre_funcion)(argumentos);
```

La forma de llamar a la función a partir de su puntero es la siguiente:

```
(*nombre_funcion)(argumentos);
```

El siguiente ejemplo utiliza punteros a funciones:

```
#include <stdio.h>
#define FALSE 0
```

```

#define TRUE 1
int sumar(int a, int b)
{
    return(a+b);
}
int restar(int a, int b)
{
    return(a-b);
}

main()
{
    int (*operacion)(int a, int b);
    int a, b, oper, error, res;

    printf("a, b: ");
    scanf("%d %d", &a, &b);

    printf("Operacion (0=suma, 1=resta): ");
    scanf("%d", &oper);
    error = FALSE;
    switch(oper)
    {
        case 0: operacion = sumar;
                  break;
        case 1: operacion = restar;
                  break;
        default: error = TRUE;
    }
    if (!error)
    {
        res = (*operacion)(a,b);
        printf("Resultado = %d\n", res);
    }
}

```

A.13.7. Funciones como argumentos

En C se pueden pasar funciones como argumentos a otras funciones. Cuando una función acepta el nombre de otra como argumento, la declaración formal debe identificar este argumento como un puntero a otra función. Por ejemplo:

```

#define FALSE 0
#define TRUE 1
int sumar(int a, int b)
{
    return(a+b);
}
int restar(int a, int b)
{
    return(a-b);
}

```

```

int ejecutar(int (*f)(int a, int b),
int a, int b)
{
    int res;
    res = (*f)(a,b);
    return(res);
}
main()
{
    int a, b, oper, error, res;
    printf("a, b: ");
    scanf("%d %d", &a, &b);
    printf("Operacion (0=suma, 1=resta): ");
    scanf("%d", &oper);
    error = FALSE;
    switch(oper)
    {
        case 0: res = ejecutar(sumar, a, b); break;
        case 1: res = ejecutar(restar, a, b); break;
        default: error = TRUE;
    }
    if (!error)
        printf("Resultado = %d\n", res);
}

```

A.13.8. Funciones con número variable de argumentos

En C se pueden definir funciones con un número de argumentos variable. El prototipo de estas funciones es:

```
int sumar(int contador, ...);
```

El primer argumento es fijo y obligatorio. Para acceder a los diferentes argumentos hay que utilizar las funciones y macros definidas en `<stdarg.h>`:

- `va_list`, variable local utilizada para acceder a los parámetros.
- `va_start`, inicializa los argumentos.
- `va_arg`, obtiene el valor del siguiente parámetro.
- `va_end`, debería ser llamada una vez procesados todos los argumentos.

Por ejemplo, la siguiente función suma un número variable de enteros.

```

int sumar(int contador, ...)
{
    va_list ap;
    int arg;
    int total = 0;
    int i = 0;
    va_start(ap, contador);
    for (i = 0; i < contador; i++)
    {

```

```

        arg = va_arg(ap, int);
        total = total + arg;
    }
    va_end(ap);
    return(total);
}

```

La siguiente función que imprime un conjunto indeterminado de cadenas de caracteres:

```

#include <stdio.h>
#include <stdarg.h>
void imprimir(char *s1, ...)
{
    va_list ap;
    char *arg;
    va_start(ap, s1);
    printf("%s\n", s1);
    while ((arg = va_arg(ap, char *)) != NULL)
        printf("%s\n", arg);
    va_end(ap);
    return;
}
main()
{
    imprimir("1", "2", "3", NULL);
}

```

A.13.9. Compilación condicional

Las siguientes líneas:

```
#ifdef identificador
#ifndef identificador
```

indican la compilación condicional del texto que las sigue hasta encontrar la línea:

```
#endif
```

Para compilar con este tipo de directivas se debe incluir en la línea de mandatos del compilador, como se muestra a continuación:

```
cc -Didentificador programa.c
```

La compilación condicional es útil en la fase de depuración y prueba de un programa. El siguiente ejemplo hace uso de esta idea:

```

#include <stdio.h>
#include <stdarg.h>

int sumar(int contador, ...)
{

```

```
va_list ap;
int arg;
int total = 0;
int i = 0;
va_start(ap, contador);
for (i = 0; i < contador; i++)
{
    arg = va_arg(ap, int);
#endif DEBUG
    printf("total = %d\n", total);
    printf("arg = %d\n", arg);
#endif
    total = total + arg;
}
va_end(ap);
return(total);
}

main()
{
    int suma;
    suma = sumar(4, 1, 2, 3, 4);
    printf("suma = %d\n", suma);
}
```

B

Programación con lenguajes de *scripts*

A lo largo del libro se han propuesto varias prácticas en las que ha habido que desarrollar scripts que resolviesen un determinado problema. En esta sección se presentan algunos conceptos generales sobre el intérprete de mandatos de UNIX, denominado shell, centrándose en los mecanismos de este intérprete más relacionados con el desarrollo de archivos de mandatos (shell scripts). Es conveniente resaltar, por tanto, que el objetivo de esta sección no es proporcionar un manual de usuario, que el lector puede encontrar fácilmente en las referencias bibliográficas recomendadas en el libro.

B.1. EL SHELL DE UNIX

El intérprete de mandatos de UNIX se denomina *shell*, dado que, como una concha, envuelve al núcleo del sistema operativo ocultándolo al usuario. Puesto que el *shell* es un programa como otro cualquiera, a lo largo de la historia de UNIX ha habido muchos programadores que se han decidido a construir uno de acuerdo con sus preferencias personales. A continuación destacaremos aquellos que han alcanzado mayor difusión:

- El *shell* de Bourne (*sh*). El *shell* original de UNIX. Se trata de un intérprete que ofrece al usuario una herramienta de programación poderosa. Su principal defecto es la falta de ayuda al usuario interactivo para facilitarle la labor de introducir mandatos. No incluye, al menos originalmente, facilidades para el control de trabajos.
- El *C-shell* (*csh*). Fue desarrollado en la Universidad de Berkeley. Tiene una sintaxis que recuerda al lenguaje C. Incluye facilidades para el usuario interactivo y capacidad de control de trabajos. El *tcsesh* es una versión mejorada del mismo con mayor funcionalidad para el usuario interactivo.
- El *shell* de Korn (*ksh*). Se desarrolló en AT&T. A diferencia del resto de los comentados en esta sección, no es de libre distribución. El diseño de este intérprete tomó como punto de partida el *shell* de Bourne, añadiéndole muchas de las características del *C-shell* (y de otros sistemas operativos).
- El *shell* de GNU (*bash*: *Bourne-Again SHeLL*). Los planteamientos de diseño de este intérprete son similares a los del *shell* de Korn, pero, obviamente, se trata de un programa de libre distribución que ha sido adoptado como el intérprete para Linux.

Dada la gran variedad de *shells* existente, la exposición se centrará, siempre que sea posible, en el *shell* de Bourne, ya que es el que está disponible en todos los sistemas y dado que sus características son normalmente aplicables a todos los *shells* derivados del mismo (por ejemplo, *ksh* y *ksh*). Además, la propuesta de *shell* especificada en el estándar POSIX 1003.2 está basada en la familia de *shells* derivados del *sh*. A continuación se presentan las características más relevantes de los *shells* de UNIX.

B.2. ESTRUCTURA DE LOS MANDATOS

Cada mandato es una secuencia de palabras separadas por espacios tal que la primera palabra es el nombre del mandato y las siguientes son sus argumentos. Así, por ejemplo, para borrar los archivos *arch1* y *arch2* se podría especificar el mandato siguiente:

```
rm arch1 arch2
```

Donde *rm* es el nombre del mandato y *arch1* y *arch2* sus argumentos.

Una vez leído el mandato, el *shell* iniciará su ejecución en el contexto de un nuevo proceso (*sub-shell*) y esperará la finalización del mismo antes de continuar (modo de ejecución *foreground*).

La ejecución de un mandato devuelve un valor que refleja el estado de terminación del mismo. Por convención, un valor distinto de 0 indica que ha ocurrido algún tipo de error (valor falso). Así, en el ejemplo anterior, si alguno de los archivos especificados no existe, *rm* devolverá un valor distinto de 0. Como se verá más adelante, este valor puede ser consultado por el usuario o por posteriores mandatos.

Si una línea contiene el carácter #, esto indicará que el resto de los caracteres de la misma que aparecen a partir de dicho carácter se considerarán un comentario y, por tanto, el *shell* los ignorará. Aunque se pueden usar comentarios cuando se trabaja en modo interactivo, su uso más frecuente es dentro de *shell scripts*.

B.3. AGRUPAMIENTO DE MANDATOS

Los *shells* de UNIX permiten que el usuario introduzca varios mandatos en una línea existiendo, por tanto, otros caracteres además del de fin de línea, que delimitan dónde termina un mandato o que actúan de separadores entre dos mandatos. Esta posibilidad va a permitir que el usuario pueda expresar operaciones realmente complejas en una única línea. Cada tipo de carácter delimitador o separador tiene asociado un determinado comportamiento que afecta a la ejecución del mandato o de los mandatos correspondientes. El valor devuelto por una lista será el del último mandato ejecutado (excepto en el caso de una lista asíncrona que devuelve siempre un 0). Dependiendo del delimitador o separador utilizado, se pueden construir los siguientes tipos de listas de mandatos:

- Lista con tuberías (carácter separador)
- Lista O-lógico (carácter separador)
- Lista Y-lógico (carácter separador)
- Lista secuencial (carácter delimitador)
- Lista asíncrona (carácter delimitador)

B.3.1. Lista con tuberías (*pipes*)

Se ejecutan de forma concurrente los mandatos incluidos en la lista, de tal forma que la salida estándar de cada mandato queda conectada a la entrada estándar del siguiente en la lista mediante un mecanismo denominado tubería (*pipe*). En el apartado dedicado a las redirecciones se profundizará más sobre el concepto de entrada y salida estándar. La ejecución de la lista terminará cuando terminen todos los mandatos incluidos en la misma (algunos *shells* relajan esta condición terminando la lista cuando finaliza el mandato que aparece más a la derecha). El siguiente ejemplo muestra una línea de mandatos que imprime (mandato *lpr*), en orden alfabético (mandato *sort*), aquellas líneas que, estando entre las diez primeras del archivo *carta* (mandato *head*), contengan la cadena de caracteres *Juan* (mandato *grep*).

```
head -10 carta | grep Juan | sort | lpr
```

B.3.2. Lista O-lógico (OR)

Se ejecutan de forma secuencial (de izquierda a derecha) los mandatos incluidos en la lista hasta que uno de ellos devuelva un valor igual a 0 (por convención, verdadero). El siguiente ejemplo imprime el archivo sólo si no es un directorio (mandato *test* con la opción -d).

```
test -d archivo || lpr archivo
```

B.3.3. Lista Y-lógico (AND)

Se ejecutan de forma secuencial (de izquierda a derecha) los mandatos incluidos en la lista hasta que uno de ellos devuelva un valor distinto de 0 (por convención, falso). El siguiente ejemplo imprime el archivo sólo si se trata de un archivo ordinario (mandato `test` con la opción `-f`).

```
test -f archivo && lpr archivo
```

B.3.4. Lista secuencial

Se ejecutan de forma secuencial (de izquierda a derecha) todos los mandatos incluidos en la lista. El resultado es el mismo que si cada mandato se hubiera escrito en una línea diferente. La siguiente línea intercambia el nombre de dos archivos usando repetidamente el mandato `mv`.

```
mv arch1 aux; mv arch2 arch1; mv aux arch1
```

B.3.5. Lista asíncrona (*background*)

Se inicia la ejecución de cada mandato de la lista, pero el *shell* no se queda esperando su finalización, sino que una vez arrancados continúa con su labor (modo de ejecución *background*). Así, por ejemplo, si se quiere poder seguir trabajando con el *shell* mientras se compilan (mandato `gcc`) dos programas muy largos, se puede especificar la siguiente línea:

```
gcc programa_muy_largo_1.c & gcc programa_muy_largo_2.c &
```

Los diversos tipos de listas se pueden mezclar para crear líneas más complejas. Cada tipo de delimitador o separador tiene asociado un orden de precedencia que determina cómo interpretará el *shell* una línea que mezcle distintos tipos de listas. En el caso de la misma precedencia, el análisis de la línea se hace de izquierda a derecha. El orden de precedencia, de mayor a menor, es el siguiente:

- Listas con tuberías.
- Listas Y y O (misma preferencia).
- Lista asíncronas y secuenciales (misma preferencia).

Asimismo, existe la posibilidad de agrupar mandatos para alterar las relaciones de precedencia entre los separadores y delimitadores. De manera similar a lo que sucede con las expresiones aritméticas, se usan paréntesis o llaves. Hay algunas diferencias entre el uso de estos símbolos:

- (`lista`): Los mandatos de la lista no los ejecuta el *shell* que los leyó, sino que se arranca un *subshell* para ello. Como se verá más adelante, esto tiene repercusiones en el uso de variables y mandatos internos.
- { `lista`; }: Los mandatos de la lista los ejecuta el *shell* que los leyó. La principal desventaja de esta construcción es que su sintaxis es algo irritante (nótese el espacio que aparece antes de la lista y el punto y coma que hay después).

En el siguiente ejemplo se muestra el uso combinado de varios tipo de listas. La línea que aparece a continuación imprime dentro de una hora (mandato `sleep`), en orden alfabético, de

entre las primeras cien líneas de arc las que contengan la palabra Figura, comprobando antes si arc es accesible (mandato test con la opción -r):

```
(sleep 3600; test -r arc && head -100 arc | grep Figura | sort | lp)&
```

Nótese la necesidad de usar paréntesis para lograr que la ejecución de toda la línea se haga de forma asíncrona, esto es, que el *shell* quede inmediatamente listo para atender más peticiones. Si no se usaran los paréntesis (o llaves), el *shell* se quedaría bloqueado durante una hora, ya que la ejecución en *background* sólo afectaría a los mandatos que están a la derecha del punto y coma.

Este ejemplo muestra la versatilidad de la interfaz de UNIX que permite combinar mandatos que llevan a cabo labores básicas para realizar operaciones complejas. Obsérvese que las listas de mandatos se consideran a su vez mandatos y que, por tanto, cuando a partir de este momento usemos el término mandato, nos estaremos refiriendo tanto a mandatos simples como a listas de mandatos, a no ser que se especifique lo contrario.

B.4. MANDATOS COMPUESTOS Y FUNCIONES

Como se comentó previamente, el *shell* es una herramienta de programación y como tal pone a disposición del usuario casi todos los mecanismos presentes en un lenguaje de programación convencional. Así, proporciona una serie de mandatos compuestos que permiten construir estructuras de ejecución condicionales y bucles. Asimismo, permite definir funciones que facilitan la modularidad de los programas de mandatos. En todas las construcciones que se presentarán a continuación el valor devuelto por las mismas será el del último mandato ejecutado.

B.4.1. El mandato condicional if

Esta construcción ejecutará un mandato o no dependiendo del estado de terminación de otro mandato que ha ejecutado previamente. Su sintaxis es la siguiente:

```
if lista1
then
    lista # si verdadero lista1
elif lista2
then
    lista # si falso lista1 y verdadero lista2
else
    lista # si falso lista1 y lista2.
fi
```

Nótese que tanto la rama correspondiente al elif como la del else son opcionales. El siguiente ejemplo, similar al usado para las listas-Y, imprime el archivo sólo si se trata de un archivo ordinario. En caso contrario, muestra por la pantalla un mensaje de error (mandato echo).

```
if test -f archivo
then
    lpr archivo
else
    echo "Error: Se ha intentado imprimir un archivo no regular!"
fi
```

B.4.2. El mandato condicional case

Este mandato condicional compara la palabra especificada con los distintos patrones y ejecuta el mandato asociado al primer patrón que se corresponda con dicha palabra. La comparación entre la palabra y los distintos patrones seguirá las mismas reglas que se usan en la expansión de nombres de archivos, que se verá más adelante. Su sintaxis es la siguiente:

```
case palabra in
    patron1) lista1;; # ejecutada si palabra encaja en patron1
    patron2|patron3) lista2;; # ejecutada si palabra encaja en
                        # patron2 o patron3
esac
```

En el apartado correspondiente a las funciones se mostrará un ejemplo del uso de esta construcción.

B.4.3. El bucle until

Esta construcción ejecutará un mandato hasta que la ejecución de otro mandato devuelva un valor igual a verdadero. Su sintaxis es la siguiente:

```
until lista1
do
    lista # ejecutada hasta que lista1 devuelva verdadero
done
```

El siguiente ejemplo ejecuta en *background* un bucle until que muestre cada cinco segundos (mandato sleep) la llegada de un mensaje (mandato mail) en cuya cabecera aparezca la palabra URGENTE. Cuando se detecta, se escribe (mandato printf) un pitido y se termina el bucle.

```
until mail -H | grep URGENTE && printf '\a'
do
    sleep 5
done &
```

B.4.4. El bucle while

Esta construcción ejecutará un mandato mientras que la ejecución de otro mandato devuelva un valor igual a verdadero. Su sintaxis es la siguiente:

```
while lista1
do
    lista # ejecutada mientras verdadero lista1
done
```

B.4.5. El bucle for

En cada iteración de este tipo de bucle se ejecutará el mandato especificado tomando la variable el valor de cada uno de los sucesivos elementos que aparecen en la lista de palabras. Su sintaxis es la siguiente:

```

for VAR in palabra1 ... palabraN
do
    lista # en cada iteración VAR toma valor de sucesivas palabras
done

```

Si no aparece la palabra reservada `in` ni la secuencia de palabras, equivale a especificar la variable `$`, que, como se verá en la sección dedicada a las variables, se corresponde con la lista de parámetros de un archivo de mandatos o una función. Así, el siguiente fragmento:

```

for VAR
do
    lista
done

```

Equivaldría a:

```

for VAR in $*
do
    lista # en cada iteración VAR toma el valor de un argumento
done

```

En el siguiente ejemplo se usa un bucle `for` para realizar una copia de seguridad (con la extensión `.bak`) de un conjunto de archivos.

```

for ARCHIVO in m1.c m2.c cap1.txt cap2.txt resumen.txt
do
    test -f $ARCHIVO && cp $ARCHIVO $ARCHIVO.bak
done

```

B.4.6. Funciones

La definición de una función permite asociar con un nombre especificado por el usuario un mandato (simple, lista de mandatos o mandato compuesto). La invocación de dicho nombre como si fuera un mandato simple producirá la ejecución del mandato asociado que recibirá los argumentos especificados. Más adelante, en el apartado dedicado a los parámetros, se tratará en detalle este tema. El valor devuelto por la función será el del último mandato ejecutado dentro de la misma (por legibilidad, se suele usar el mandato interno `return` para terminarla). La sintaxis para definir una función es la siguiente:

```
nombre_funcion() lista
```

Por motivos de legibilidad, se usan normalmente las llaves para delimitar el cuerpo de la función. Así, la estructura resulta similar a la del lenguaje C.

```

nombre_funcion()
{
    lista
}

```

En cuanto a la invocación de la función, como se comentó antes, se realiza utilizando el nombre de la misma como si fuera un mandato simple:

```
nombre_funcion arg1 arg2 ... argn
```

El siguiente ejemplo muestra una función que recibe como argumento un conjunto de archivos y que, de forma similar al ejemplo anterior, realiza una copia de seguridad de cada uno de ellos. Nótese el uso del `case` para evitar sacar copias de seguridad de las propias copias (archivos con extensión `bak` o `BAK`). En dicha construcción se ha usado el patrón que, como se verá en la sección dedicada a la expansión de nombres de archivos, representa a cualquier cadena de cero o más caracteres.

```
copia_seguridad()
{
    for ARCHIVO
    do
        case $ARCHIVO in
            *.BAK|*.bak) ;;
            *) test -f $ARCHIVO && cp $ARCHIVO $ARCHIVO.bak;;
        esac
    done
}
```

La función se invocaría como si fuera un mandato simple, especificando los archivos de los que se quiere sacar una copia de seguridad:

```
copia_seguridad m1.c m2.c cap1.txt cap2.txt resumen.txt
```

B.5. REDIRECCIONES

El *shell* permite al usuario redirigir la entrada y salida que producirá un mandato durante su ejecución. Con este mecanismo, el usuario puede invocar un mandato especificando, por ejemplo, que los datos que lea el programa los tome de un determinado archivo en vez del terminal. Normalmente, este mecanismo se usa para redirigir alguno de los tres descriptores estándar.

La mayoría de los mandatos, y en general de los programas de UNIX, leen sus datos de entrada del terminal y escriben sus resultados y los posibles mensajes de error también en el terminal. Para simplificar el desarrollo de estos programas y permitir que puedan leer y escribir directamente en el terminal sin realizar ninguna operación previa (o sea, sin necesidad de realizar una llamada `OPEN`), los programas reciben por convención tres descriptores, normalmente asociados al terminal, ya preparados para su uso (esto es, para leer y escribir directamente en ellos):

- Entrada estándar (descriptor 0): De donde el programa puede leer sus datos.
- Salida estándar (descriptor 1): Donde el programa puede escribir sus resultados.
- Error estándar (descriptor 2): Donde el programa puede escribir los mensajes de error.

Anteriormente se presentó un primer tipo de redirección asociado a la lista con tuberías que permitía que la salida estándar de un mandato quedase conectada a la entrada estándar de otro. En este apartado nos centraremos en las redirecciones a archivos.

B.5.1. Redirección de salida

El usuario puede redirigir la salida estándar de un programa a un archivo usando el carácter `>` delante del nombre del archivo. En general, la expresión `n> arch` permite redirigir el descriptor

de salida n al archivo arch. Con este tipo de redirección, si el archivo existe, su contenido inicial se pierde, esto es, el *shell* trunca el archivo a una longitud de cero antes de que se produzca la ejecución del mandato. Si lo que el usuario desea es que, en el caso de que el archivo exista, la salida producida por el mandato se concatene detrás del contenido inicial del archivo, deberá usar " (en general, "n" arch) para especificar la redirección.

Además de poder redirigir la salida a un archivo, el usuario puede especificar que la salida asociada a un determinado descriptor se redirija al mismo destino que tiene asociado otro descriptor. Así, la expresión n>&m indica que la salida asociada al descriptor n se redirige al mismo destino que corresponde al descriptor m. Si, por ejemplo, el usuario quiere que tanto la salida estándar como la salida de error de un mandato se redirijan a un archivo, puede usar la siguiente línea:

```
mandato > archivo 2>&1
```

B.5.2. Redirección de entrada

De manera similar a lo visto para la salida, el usuario puede redirigir la entrada estándar de un programa a un archivo usando el carácter < delante del nombre del archivo. En general, la expresión n< arch permite redirigir el descriptor de entrada n al archivo arch. Asimismo, el usuario puede especificar que la entrada asociada a un determinado descriptor se redirija a la misma fuente que tiene asociada otro descriptor. Así, por ejemplo, la expresión n<&m indica que la entrada asociada al descriptor n se redirige a la misma fuente que corresponde al descriptor m.

A continuación se muestra un ejemplo en el que se redirigen los tres descriptores del mandato sort para ordenar los datos contenidos en el archivo entrada, dejando el resultado en el archivo salida y almacenando los posibles mensajes de error en el archivo error:

```
sort > salida < entrada 2> error
```

Otro tipo de redirección de entrada, usada principalmente en *shell scripts*, se corresponde con los denominados documentos in-situ (*here documents*). Con este mecanismo, el usuario especifica que la entrada estándar de un mandato se tomará de las líneas que siguen al propio mandato hasta que aparezca una línea que contenga únicamente una determinada palabra que especificó el usuario en la redirección. Dicho de manera informal, es como si el usuario pegara al mandato los datos que quiere que éste lea. Este tipo de redirección tiene la siguiente estructura:

```
mandato << marca_fin
linea de entrada 1
linea de entrada 2
.....
linea de entrada N
marca_fin
```

Las líneas que se tomarán como entrada del mandato sufrirán previamente las expansiones que realiza habitualmente el *shell* (véase la sección sobre la expansión de argumentos), a no ser que la palabra que aparece en la especificación de la redirección tenga algún carácter con *quoting* (este concepto se trata en el siguiente apartado).

B.6. QUOTING

Algunos caracteres o palabras tienen un significado especial para el *shell*. Cuando al procesar una línea el *shell* encuentra una de estas secuencias, en vez de pasársela directamente al mandato correspondiente, la procesa transformando la línea leída de acuerdo con las acciones asociadas a la misma. El mecanismo de *quoting* permite que el usuario especifique que el *shell* no procese un determinado carácter (o secuencia de caracteres), a pesar de que éste tenga un significado especial para el *shell*. Para ello usará a su vez un metacarácter de protección:

- Colocará el símbolo backslash delante del carácter que pretende proteger.
- Encerrará entre comillas simples la secuencia de caracteres que pretende proteger.
- Encerrará entre comillas dobles la secuencia de caracteres que pretende proteger. En este caso, la protección no es completa y permite que se realicen algunos tipos de procesamiento. Concretamente, no impide la expansión de variables y la sustitución de mandatos.

El siguiente ejemplo muestra cómo se pueden borrar los archivos *mio\;tuyo* y *nombreraro*:

```
rm mio\;tuyo nombre';>'raro
```

Nótese que por comodidad el usuario normalmente aplicaría las comillas a todo el nombre del segundo archivo.

B.7. EXPANSIÓN DE ARGUMENTOS

Ciertos caracteres tienen un significado especial para el *shell*, y siempre que aparecen sin proteger en una determinada línea, el *shell* llevará a cabo el procesado correspondiente transformando la línea original de acuerdo con la funcionalidad asociada a cada uno de ellos. El *shell* ejecutará la línea resultante del tratamiento como si la hubiese introducido de esta forma el usuario. A esta acción de procesar los metacaracteres presentes en un argumento la denominaremos expansión de argumentos, y en esta sección trataremos los siguientes tipos de expansión: expansión de tilde, expansión de variables, sustitución de mandatos, expansión aritmética y expansión de nombres de archivos.

B.7.1. Expansión de tilde

El carácter tilde, no disponible en el *shell* de Bourne, pero sí en el resto de los comentados, se sustituye (se expande) por el nombre del directorio base (HOME) del usuario (o de otro usuario). Por ejemplo, el siguiente mandato muestra todos los archivos (mandato *ls*) contenidos en el subdirectorio *bin* del directorio base del usuario y en el subdirectorio *pub* del directorio base del usuario *jgarcia*.

```
ls ~/bin ~jgarcia/pub
```

B.7.2. Expansión de variables

Como es habitual en cualquier lenguaje de programación convencional, cada variable está identificada por un nombre y almacena un valor. Para obtener el valor de una variable debe especificar-

se el carácter \$ delante de su nombre. Cuando ocurre esto en una línea, el *shell* lo sustituye por su valor asociado. Más adelante, en la sección que trata sobre los parámetros, se volverá a tratar este tema.

B.7.3. Sustitución de mandatos

Este mecanismo permite que el usuario especifique que en una línea se reemplace un determinado mandato por la salida que produce su ejecución. Para ello, el mandato debe estar encerrado entre ' (o también dentro de la construcción \$(mandato) en el caso del ksh y bash). Cuando el *shell* detecta esta construcción durante el tratamiento de una línea, ejecuta el mandato afectado por la misma y sustituye en la línea original el mandato por la salida que éste produce al ejecutarse. El siguiente ejemplo borra (mandato rm) los archivos que contenga la cadena de caracteres CADENA (el mandato grep con la opción -l imprime los nombres de los archivos que contenga la cadena especificada).

```
rm -f 'grep -l CADENA arch1 arch2 arch3 arch4 arch5'
```

Suponiendo que arch2 y arch4 contienen dicha cadena de caracteres, la sustitución del mandato produciría la siguiente transformación:

```
rm -f arch2 arch4
```

B.7.4. Expansión aritmética

La mayoría de los *shells*, a excepción del de Bourne, poseen la capacidad de evaluar expresiones aritméticas. En el caso del ksh y bash, la expresión debe estar incluida en la construcción \${((expresión))}. Cuando el *shell* detecta esta construcción durante el tratamiento de una línea, evalúa la expresión y la sustituye en la línea original por el resultado de la misma. El siguiente ejemplo muestra el uso combinado de la sustitución de mandatos y de la expansión aritmética para mostrar por la pantalla la primera mitad de las líneas del archivo archivo (el mandato wc -l cuenta el número de líneas de un archivo).

```
head -$((`wc -l < archivo` / 2)) archivo
```

Suponiendo que archivo tiene 16 líneas, la sustitución del mandato produciría la siguiente transformación:

```
head -$((16 / 2)) archivo
```

Y la expansión aritmética tendría como resultado:

```
head -8 archivo
```

Como se comentó previamente, el *shell* de Bourne no incluye estas facilidades, por lo que hay que recurrir al mandato expr para realizar operaciones aritméticas. El ejemplo anterior habría que reescribirlo de la siguiente forma:

```
head -`expr `wc -l < archivo` / 2` archivo
```

B.7.5. Expansión de nombres de archivos

El *shell* facilita la labor de especificar un nombre de archivo proporcionando al usuario un conjunto de caracteres que actúan como comodines (*wildcards*). Cuando el *shell* encuentra uno de estos caracteres en una palabra, considera que se trata de un patrón de búsqueda y sustituye la palabra original por todos los nombres de archivo que encajan en ese patrón. En este tipo de expansión están involucrados los siguientes metacaracteres:

- El carácter ***** representa a cualquier cadena de cero o más caracteres dentro del nombre de un archivo.
- El carácter **?** se corresponde con un único carácter, sea éste cual sea.
- Una secuencia de caracteres encerrada entre llaves cuadradas se corresponde con un único carácter de los especificados, presentándose las siguientes excepciones:
 - Si el primer carácter de la secuencia es **!**, se corresponde con cualquier carácter, excepto los especificados.
 - Si aparece un **-** entre dos caracteres de la secuencia, se corresponde con cualquier carácter en el rango entre ambos.

El siguiente ejemplo borraría los archivos cuyo nombre comience y termine por A (incluido el archivo AA), aquellos cuyo nombre empieza por la letra p seguida de otros cuatro caracteres cualesquiera, los que tengan un nombre de dos caracteres ambos comprendidos entre la a y la z minúsculas y, por último, los archivos cuyo nombre tengan una longitud de al menos dos caracteres tal que el primer carácter sea 1 o 2 y el segundo no sea numérico.

```
rm -f A*A p???? [a-z][a-z] [12][!1-9]*
```

La siguiente línea invoca la función `copia_seguridad`, antes presentada, pasándole como argumentos todos los archivos del directorio actual de trabajo.

```
copia_seguridad *
```

B.8. PARÁMETROS

Dado que el *shell* es una herramienta de programación, además de proporcionar los mecanismos para estructurar programas que se presentaron previamente, ofrece al usuario la posibilidad de utilizar variables, que en este entorno se las suele denominar de forma genérica como parámetros. Para obtener el valor almacenado en un parámetro se debe colocar un **\$** delante del nombre del mismo (o delante del nombre encerrado entre llaves). Hay tres tipos de parámetros: parámetros posicionales, parámetros especiales y variables propiamente dichas.

B.8.1. Parámetros posicionales

Se corresponden con los argumentos con los que se invoca un *script* o una función. Su identificador es un número que corresponde con su posición. Así, **\$1** será el valor del primer argumento, **\$2** el del segundo y, en general, **\$i** se referirá al *i*-ésimo argumento. El usuario no puede modificar de forma individual un parámetro posicional, aunque puede reasignar todos con el mandato inter-

no set. Después de ejecutar este mandato, los parámetros posicionales toman como valor los argumentos especificados en el propio mandato set.

B.8.2. Parámetros especiales

Se trata de parámetros mantenidos por el propio *shell*, por lo que el usuario no puede modificar su valor. A continuación se muestran algunos de los más frecuentemente usados:

- \$0: Nombre del *script*.
- \$# : Número de parámetros posicionales.
- \$@ : Lista de parámetros posicionales.
- \$\$: Valor devuelto por el último mandato ejecutado.
- \$\$: Identificador de proceso del propio *shell*.
- \$\$! : Identificador de proceso asociado al último mandato en *background* arrancado.

B.8.3. Variables

Este tipo de parámetros se corresponde con el concepto clásico de variable presente en los lenguajes de programación convencionales; pero dada la relativa simplicidad de este entorno, la funcionalidad asociada a las variables es reducida. En primer lugar, las variables no se declaran, creándose cuando se les asigna una valor usando la construcción *variable=valor* (o con el mandato interno *read*). No existen diferentes tipos de datos: todas las variables son consideradas del tipo cadena de caracteres. Cuando se intenta acceder a una variable que no existe no se producirá ningún error, sino que simplemente el *shell* la expandirá como un valor nulo.

Otro aspecto importante relacionado con las variables es su posible exportación a los procesos creados por el propio *shell* durante la ejecución de los distintos mandatos. Por defecto, las variables no se exportan y, por tanto, los procesos creados no obtienen una copia de las mismas. Si el usuario requiere que una variable sea exportada a los procesos hijos del *shell* debe especificarlo explícitamente mediante el mandato interno *export*. Nótese que, aunque una variable se exporta, las modificaciones que haga sobre ella el proceso hijo no afectan al padre, ya que el hijo obtiene una copia de la misma.

El *shell* durante su fase de arranque crea una variable por cada una de las definiciones presentes en el entorno del proceso que ejecuta el *shell*. Estas variables se consideran exportadas de forma automática. Como ejemplo típico de este tipo de variables podemos considerar la variable HOME, que contiene el nombre del directorio base del usuario, y la variable PATH, que representa la lista de directorios donde el *shell* busca los mandatos.

B.9. SHELL SCRIPTS

El comportamiento del *shell* va a depender de los argumentos y opciones que se especifiquen en su invocación:

- Si recibe la opción -i o no se le especifican argumentos, el *shell* trabajará en modo interactivo, leyendo de su entrada estándar los mandatos que el usuario introduce.
- Si recibe la opción -c *cadena_de_caracteres*, el *shell* ejecutará únicamente los mandatos contenidos en *cadena_de_caracteres* y terminará.

- Si recibe como argumento el nombre de un archivo (*shell script*), el *shell* ejecutará los mandatos contenidos en el mismo.

En este apartado nos centraremos en este último caso, ya que, como se comentó al principio de la sección dedicada al *shell* de UNIX, esta exposición está centrada en los aspectos relacionados con la programación de *scripts*, dejando a un lado las cuestiones asociadas al modo de trabajo interactivo.

La ejecución de un *script*, por tanto, se realiza de la siguiente manera:

```
sh script arg1 arg2 ... argn
```

Con lo que quedaría el parámetro \$0 con un valor igual a *script*, \$1 igual a *arg1* y así sucesivamente. El valor devuelto por el *script* será el del último mandato ejecutado (por legibilidad, se suele usar el mandato interno *exit* para terminarla).

Por motivos de comodidad para el usuario y para proporcionar una manera uniforme de invocar todos los programas en el sistema, los *scripts* también pueden arrancarse de la siguiente forma:

```
script arg1 arg2 ... argn
```

Evidentemente, esta segunda forma es ventajosa, pero presenta el problema de cómo determinar qué *shell* quiere el usuario que se arranque para ejecutar el *script* (nótese que normalmente en un sistema UNIX hay varios *shells* disponibles). Va a ser la primera línea del *script* la que especifique qué intérprete se arrancará usándose la siguiente sintaxis:

```
#!interprete
```

Donde *interprete* debe especificar el nombre del archivo que contiene el ejecutable del intérprete. Así, si la primera línea del archivo *script* es la siguiente:

```
#!/bin/sh
```

Ejecutar la siguiente línea:

```
script arg1 arg2 ... argn
```

Equivaldría a lo siguiente:

```
/bin/sh arg1 arg2 ... argn
```

Nótese que se trata de un mecanismo genérico que permite especificar cualquier tipo de intérprete para procesar el archivo. Así, por ejemplo, el archivo podría contener mandatos de una determinada base de datos y el intérprete podría ser el programa de interfaz de la base de datos.

Existe una forma alternativa de ejecutar un *script* mediante el uso del mandato interno. Cuando se invoca un *script* de esta forma, no se arranca un *shell* para ejecutar los mandatos contenidos en el mismo, sino que es el propio *shell* actual el que los lee y ejecuta. Esto hace que, a diferencia de lo que sucede cuando se ejecuta sin este mandato interno, tanto las funciones definidas en el *script* como las variables actualizadas afecten al *shell* actual una vez que finaliza la ejecución del *script*.

```
script arg1 arg2 ... argn
```

A continuación se presentarán tres ejemplos de *scripts* cuyo estudio se deja al lector:
 El primero, denominado *endir*, comprueba si en un determinado directorio, especificado como primer argumento, está presente una serie de archivos que se le pasan como los siguientes argumentos:

```

#!/bin/sh
# Este script determina si un archivo esta presente en un
# directorio.
#   - Primer argumento: el nombre de un directorio
#   - Argumentos restantes: los archivos que debe comprobarse
# Control de errores:
#   - Debe comprobase que al menos se reciben dos argumentos.
#     Si no es asi se imprime un mensaje de error y termina
#     devolviendo un 1.
#   - Debe comprobase que el primer argumento es un directorio.
#     Si no es asi se imprime un mensaje de error y termina
#     devolviendo un 2.
#   - Si alguno de los archivos recibidos como argumentos no existen
#     en el directorio o existiendo no son archivos regulares,
#     se imprime un mensaje de error y se continua
#     con el resto. Al final el script devolvera 3.
# NOTA: Este script solo tiene interes pedagogico. Esta no seria
# la forma normal de hacer esta operacion

# Funcion que imprime un mensaje de error y termina. Recibe como
# parametro el valor que devolvera
Error()
{
    echo "Uso: 'basename $0' directorio archivo ..." >&2
    exit $1
}

# Si el numero de argumentos es menor que 2, llama a Error (valor=1)
test $# -lt 2 && Error 1

# Si el primer argumento no es un directorio, llama a Error (valor=2)
test -d $1 || Error 2

# Variable que mantiene el valor que devolvera el script
ESTADO=0

# Obtiene el nombre del directorio
DIRECTORIO=$1

# Desplaza los argumentos mediante el mandato interno shift:
# $2->$1, $3->$2, ...
shift

# Bucle que hace una iteracion por cada archivo recibido como
# argumento. En cada una, ARCHIVO contiene dicho argumento
for ARCHIVO
do
    # Si el nombre del archivo contiene informacion del path
    # (p.ej. /etc/passwd) la elimina (dejarria passwd)

```

```

        ARCHIVO='basename $ARCHIVO'

# Comprueba si existe un archivo regular con ese nombre
if test -f $DIRECTORIO/$ARCHIVO
then
    echo "$DIRECTORIO: $ARCHIVO"
else
    echo "$DIRECTORIO: $ARCHIVO no existe o no es regular">>&2
ESTADO=3
fi
done
exit $ESTADO

```

El segundo *script*, denominado *listdir*, muestra el contenido de un directorio.

```

#!/bin/sh
# Este script imprime el contenido de un directorio escribiendo un "/" 
# al final de los nombres de los subdirectorios.
#      - Argumento: el nombre de un directorio
# Control de errores:
#      - Debe comprobarse que se recibe un solo argumento y que
# este es un directorio.
#      Si no es asi se imprime un mensaje de error y termina
#          devolviendo un 1.

# NOTA: Este script solo tiene interes pedagogico. Esta no seria
# la forma normal de hacer esta operacion

# Funcion que imprime un mensaje de error y termina.
Error()
{
    echo "Uso: 'basename $0' directorio">>&2
    exit 1
}

# Si el numero de argumentos no es 1 o el argumento no es un
# directorio, llama a Error
test $# -eq 1 && test -d $1 || Error

# Obtiene el nombre del directorio
DIRECTORIO=$1

# Bucle que hace una iteracion por cada archivo existente (*) en
# DIRECTORIO.
for ARCHIVO in $DIRECTORIO/*
do
    if test -d $ARCHIVO
    then
        echo "$ARCHIVO/"
    else
        echo "$ARCHIVO"
    fi
done

```

El tercer ejemplo, denominado `elige1`, escribe los argumentos que recibe y permite que el usuario elija uno de ellos tecleando el número que le corresponde.

```
#!/bin/sh
# Este script escribe por la pantalla sus argumentos y permite que
# el usuario seleccione interactivamente uno de ellos.
# El dialogo con el usuario esta asociado directamente al terminal
# (/dev/tty) de esta forma se asegura que, aunque se redirija
# el script, el dialogo sera con el terminal.

# Comprueba que al menos recibe un argumento
test $# -ge 1 || exit 1
# Bucle que imprime los argumentos recibidos
NUMERO=0
for i
do
    NUMERO='expr $NUMERO + 1'
    echo "$i ($NUMERO)" > /dev/tty
done

ENTRADA=0
while test $ENTRADA -lt 1 || test $ENTRADA -gt $NUMERO
do
    echo -n "Elige introduciendo el numero correspondiente: ">/dev/tty
    read ENTRADA < /dev/tty
done

# imprime por la salida estandar el argumento seleccionado
shift 'expr $ENTRADA - 1'
echo $1
```

B.10. EJECUCIÓN DE UN MANDATO

Una vez que el *shell* ha analizado la línea leída, llevado a cabo todas las expansiones y sustituciones anteriormente explicadas, identificado el o los mandatos simples implicados y realizadas las redirecciones, se realizaría la siguiente labor por cada mandato simple involucrado:

- Si el nombre del mandato coincide con el de una función, se invoca la función que recibirán los argumentos especificados como sus parámetros posicionales.
- Si no es una función, pero se trata de un mandato interno, se invoca el mandato interno con los argumentos especificados.
- Si no es ni función ni mandato interno y el nombre no incluye ningún carácter /, se busca si existe un archivo ejecutable con ese nombre en alguno de los directorios almacenados en la variable PATH. Si es así, se arranca un proceso (una llamada fork seguida por una llamada exec) para ejecutar el programa. El programa arrancado recibirá los argumentos especificados y como entorno todas las variables exportadas. En el caso de que el ejecutable sea un script, los argumentos los obtendrá como sus parámetros posicionales y el entorno como variables.
- En el caso de que el nombre contenga algún /, el *shell* comprobará si existe un archivo con ese nombre y es ejecutable. Si es así, se procede de la misma forma que en el caso anterior, arrancando un proceso (una llamada fork seguida por una llamada exec) para ejecutarlo.

B.11. MANDATOS DE UNIX

La interfaz de usuario no consiste únicamente en el *shell*, sino también en un gran número de programas del sistema disponibles para el usuario. Nótese que al tratarse de un intérprete con mandatos externos, el *shell* es meramente un lanzador de programas y, por tanto, el sistema debe incluir una serie de mandatos para que el usuario pueda realizar las operaciones que desea. El estándar POSIX 1003.2, por consiguiente, no sólo trata de los aspectos relacionados con el *shell*, sino que también especifica qué mandatos deben existir y cuál debe ser su comportamiento. Hay que recordar en este punto que, a pesar del modelo de mandatos externos del *shell*, algunos mandatos se tienen que implementar como internos debido a que su efecto sólo puede lograrse si es el propio intérprete el que ejecuta el mandato. A grandes rasgos, los mandatos disponibles se pueden dividir en las siguientes categorías:

- Mandatos para manipular archivos y directorios.
- Herramientas para el desarrollo de aplicaciones.
- Filtros.
- Administración del sistema.
- Utilidades misceláneas.

La mayoría de los mandatos son simples y realizan labores relativamente sencillas, pero están diseñados para poderse usar conjuntamente creando listas de mandatos que pueden llevar a cabo operaciones bastante complejas. Mención aparte merece un tipo de mandatos a los que típicamente se les denomina filtros. Estos mandatos (como, por ejemplo, grep, sed, awk o sort) leen unos datos de entrada, realizan un determinado procesamiento de los mismos (filtrado) y producen una salida. Este tipo de mandatos se usan muy frecuentemente para el desarrollo de *shell scripts* y, por tanto, los programadores de este entorno deben conocerlos a fondo.

En esta sección haremos una descripción muy breve (casi una mera enumeración) de algunos de los mandatos de uso más frecuente tanto de los de carácter interno como de los externos. El lector se deberá remitir a las páginas correspondientes del manual o algunos de los libros incluidos en la bibliografía de la asignatura para obtener más información de los mismos.

B.12. MANDATOS INTERNOS

El *shell* ejecuta directamente este tipo de mandatos, o sea, el propio *shell* incluye código para realizar la funcionalidad asociada a un mandato de este tipo sin tener que activar ningún programa externo. Como se ha comentado previamente, este tratamiento especial se debe al carácter intrínsecamente interno de estos mandatos. Sin embargo, algunos intérpretes, por motivos de eficiencia y frecuencia de uso, implementan como internos algunos mandatos que no necesitarían serlo. Así, por ejemplo, el mandato echo, que imprime por la salida estándar un mensaje, está implementado en muchos *shells* como interno a pesar de no requerirlo. A continuación se presenta una lista de algunos de los mandatos internos más frecuentemente usados:

- break: Termina la ejecución de un bucle.
- cd: Cambia el directorio de trabajo actual.
- continue: Prosigue con la siguiente iteración de un bucle.
- echo: Escribe sus argumentos por la salida estándar.
- eval: Construye un mandato simple concatenando sus argumentos y hace que el *shell* lo ejecute.

- exec: Reemplaza el *shell* por otro programa. Si no se especifica ningún programa, permite redirigir los descriptores del propio *shell*.
- exit: Termina la ejecución del *shell*.
- export: Establece que las variables especificadas se exportarán.
- pwd: Imprime el directorio de trabajo actual.
- read: Lee una línea de entrada estándar asignando a variables lo leído.
- readonly: Establece que las variables especificadas no pueden modificarse.
- return: Termina la ejecución de una función.
- set: Establece valor de opciones y de parámetros posicionales.
- shift: Desplaza a la izquierda parámetros posicionales (1, ...).
- test: Evalúa expresiones condicionales.
- trap: Permite el manejo de señales.
- umask: Establece la máscara de creación de archivos.
- unset: Elimina una variable o función.

B.13. MANDATOS EXTERNOS

Antes de pasar a enumerar algunos de los mandatos externos más frecuentemente usados, es preciso resaltar que algunos de estos mandatos son complejos, incluso más que el propio *shell*, y requieren un esfuerzo considerable para lograr conocerlos bien. Así, por ejemplo, el programa make, que facilita el proceso de actualización de programas, puede tener un tamaño mayor que algunos *shells*:

- ar: Construye y mantiene bibliotecas de archivos.
- awk: Lenguaje de procesamiento y búsqueda de patrones.
- bc: Calculadora programable.
- basename: Imprime nombre de archivo eliminando información de directorio.
- cat: Concatena e imprime archivos.
- cc: Compilador del lenguaje C.
- chmod: Cambia el modo de protección de un archivo.
- chown: Cambia el propietario de un archivo.
- cmp: Compara dos archivos.
- comm: Imprime las líneas comunes de dos archivos.
- cp: Copia archivos.
- cut: Extrae los campos seleccionados de cada línea de un archivo.
- date: Imprime la fecha y la hora.
- dd: Copia un archivo realizando conversiones.
- diff: Imprime las diferencias entre dos archivos.
- dirname: Complementario de basename, ya que devuelve la información de directorio de un nombre de archivo.
- expr: Evalúa sus argumentos como una expresión.
- false: Devuelve un valor de falso.
- find: Encuentra los archivos que cumplen una condición.
- grep: Busca las líneas de un archivo que sigan un patrón.
- head: Imprime las primeras líneas de un archivo.
- id: Imprime la identidad de un usuario.
- kill: Manda una señal a un proceso.
- ln: Crea un enlace a un archivo.

- **lpr:** Manda un archivo a la impresora.
- **ls:** Lista el contenido de un directorio.
- **make:** Mantiene, actualiza y regenera programas.
- **man:** Manual del sistema.
- **mkdir:** Crea un directorio.
- **mkfifo:** Crea un FIFO.
- **mv:** Mueve o renombra un archivo.
- **od:** Muestra el contenido de un archivo en varios formatos.
- **paste:** Combina varios archivos como columnas de un único archivo.
- **pr:** Formatea un archivo para imprimirllo.
- **printf:** Escribe con formato.
- **ps:** Muestra el estado de los procesos del sistema.
- **rm:** Borra un archivo.
- **rmdir:** Borra un directorio.
- **sed:** Editor no interactivo.
- **sh:** Invoca el *shell* de Bourne.
- **sleep:** Suspende la ejecución durante un intervalo de tiempo.
- **sort:** Ordena un archivo.
- **stty:** Fija las opciones de un terminal.
- **tail:** Imprime las últimas líneas de un archivo.
- **tee:** Copia la entrada estándar a la salida estándar y a un archivo.
- **touch:** Cambia las fechas de acceso y modificación de un archivo.
- **tr:** Traduce (convierte) caracteres.
- **true:** Devuelve un valor de verdadero.
- **tty:** Devuelve el nombre del terminal del usuario.
- **uname:** Devuelve el nombre del sistema.
- **uniq:** Elimina líneas adyacentes idénticas.
- **wait:** Espera por la finalización de procesos.
- **wc:** Cuenta el número de caracteres, palabras y líneas de un archivo.

C

Entorno de programación de sistemas operativos

Para desarrollar la mayor parte de las prácticas propuestas en este libro es necesario programar aplicaciones de tamaño pequeño tanto en entorno Linux como en Windows. El objetivo de este apéndice es proporcionar unas nociones básicas de ambos entornos de programación, suficientes para que el alumno pueda realizar las prácticas. Estas nociones se pueden ampliar con los manuales interactivos de Linux y de Windows, así como en las referencias bibliográficas recomendadas en el libro.

C.1. INTRODUCCIÓN

La programación de aplicaciones sobre sistemas operativos supone conocer y usar las bibliotecas con las llamadas al sistema operativo. Para hacer una aplicación con llamadas al sistema operativo es necesario indicar en los programas los archivos con:

- La definición de prototipos y tipos de datos, por ejemplo, <windows.h>, para que puedan compilarse los módulos del programa, así como el directorio, o directorios, en que se encuentran.
- Los archivos de bibliotecas del sistema que deben enlazarse con la aplicación para crear un archivo ejecutable, incluyendo el camino dónde localizar dichas bibliotecas.
- Las opciones de compilación que deben activarse para compilar y enlazar la aplicación.

Cada compilador proporciona un entorno de programación, más o menos integrado, para simplificar el proceso de construcción del *software*. Dentro de este entorno existe habitualmente un proyecto, o *makefile*, que almacena la información que especifica cómo compilar y enlazar una aplicación.

En este apéndice se muestra brevemente el entorno de programación en el sistema operativo Windows y en UNIX/Linux. Ambos incluyen herramientas para desarrollar programas, pero el nivel de integración de las mismas suele ser muy distinto. Los entornos que se estudian en este apéndice corresponden al compilador de *Visual C++*, de Microsoft, y al compilador *gcc* del lenguaje C para UNIX/Linux. Como ejemplo de trabajo se utilizará una versión simplificada de un programa Reloj que se va describiendo en este apéndice.

C.2. MAKEFILES DE UNIX

El archivo *makefile* describe, en UNIX y Linux, cómo construir una aplicación, incluyendo las dependencias de bibliotecas y archivos. El programa *make* utiliza esta información para determinar qué archivos deben recompilarse y enlazarse para producir una unidad ejecutable de la aplicación, de forma que sólo se compilen aquellos que han cambiado o que dependen de un archivo que ha cambiado.

C.2.1. Estructura de un archivo *makefile*

El Listado C.1 contiene el *makefile* del programa Reloj, que especifica cómo construir el ejecutable *reloj*. Por convención, una especificación de *makefile* se almacena típicamente en un archivo llamado *Makefile* o *makefile*. Para ejecutar el programa *make* y construir una aplicación sólo hay que teclear el mandato *make* dentro de un intérprete de mandatos (*shell*). El programa *make* busca en el directorio actual un archivo llamado *Makefile* o *makefile* y lo procesa.

Listado C.1. *Makefile de reloj*

```

#
# Las siguientes líneas especifican que los archivos de C
# tendrán una extensión c.

.SUFFIXES:

```

```

.SUFFIXES: .c $(SUFFIXES)

# Asignar a CC el nombre del compilador de C usado.

CC = gcc

# Asignar a DIR_APOYO la ruta del directorio que contiene
# los directorios del material de apoyo.
# Las bibliotecas del sistema se incluyen por defecto.

DIR_APOYO = ./apoyo

#
# La variable CFLAGS especifica dónde encontrar
# los archivos a incluir desde el material de apoyo.

CFLAGS=-I$(DIR_APOYO)/include -g

# La siguiente regla le indica a make cómo procesar los archivos con
# extensión c. Normalmente esto no es necesario porque la extensión
# .c está definida para make.

.c.o:
    $(CC) $(CFLAGS) -c $<

#
# La variable LDFLAGS especifica dónde encontrar
# los archivos de las bibliotecas. Las bibliotecas del sistema se
# incluyen por defecto.

LDFLAGS=-L$(DIR_APOYO)/lib

#
# La variable LIBS especifica al compilador qué bibliotecas
# de archivos objeto se deben usar para construir la aplicación,
# además de las del sistema.

LIBS= -lapoyo

#
# La variable OBJS especifica al compilador qué archivos
# objeto se deben crear para construir la aplicación.
#

OBJS=reloj.o clock_task.o hardware.o

#
# La siguiente regla especifica cómo construir
# el ejecutable del programa, así como las dependencias
# de archivos #include (.h)
#

```

```

reloj: $(OBJS)
      $(CC) $(OBJS) $(LDFLAGS) $(LIBS) -o reloj

reloj.o: reloj.h
clock_task.o: reloj.h
hardware.o: reloj.h

# Regla clean. La ejecución de 'make clean' borra todos los archivos
# objeto y el ejecutable

clean:
rm -f *.o reloj

```

Para compilar en UNIX o Linux hay que ir al directorio donde se encuentra el *makefile* de la aplicación y ejecutar el mandato make, tal y como se muestra en la Figura C.1.

En un *makefile*, un comentario comienza con el carácter #. La especificación de un *makefile* puede ser muy oscura y difícil de mantener si no se comenta adecuadamente.

Las líneas siguientes del *makefile* especifican al programa make las extensiones de los archivos que se van a compilar. En este caso, se indica que los archivos de C tienen la extensión .c.

```

.SUFFIXES:
.SUFFIXES: .c $(SUFFIXES)

```

Algunos lenguajes, por ejemplo, C++, esperan archivos que tengan la extensión .cpp o .cc. Si esto es lo que sucede, se debería cambiar el .c de la segunda línea a .cpp o .cc o añadir dichos sufijos a la línea de definición. Todos los archivos de C proporcionados en este libro tienen la extensión .c.

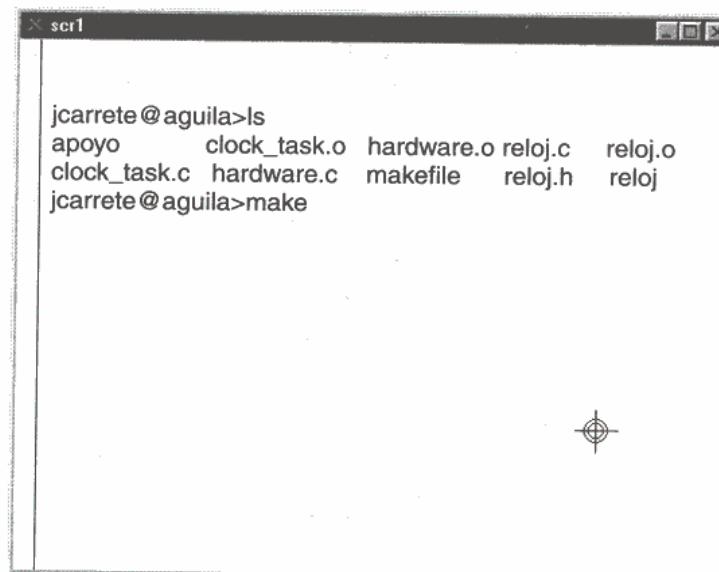


Figura C.1. Compilando con «make» en UNIX.

La siguiente línea define el compilador que se debe usar para compilar los programas:

```
CC = gcc
```

En este ejemplo se asigna a la variable CC el nombre del mandato que corresponde con el compilador de C que traduce el código fuente. En este fragmento se le da el valor gcc, que es el compilador de GNU para el lenguaje C en UNIX y Linux.

La línea:

```
DIR_APOYO = ./apoyo
```

Asigna la ruta que se corresponde con los directorios de inclusión y de la biblioteca del material de apoyo para la construcción del programa reloj. Es siempre mejor usar variables relativas a la situación del programa a construir, porque así se puede instalar la aplicación en distintos directorios cada vez sin que haya que modificar el archivo *makefile*.

La línea:

```
CFLAGS=-I $(DIR_APOYO)
```

establece una variable que informa al compilador de C dónde debe buscar los archivos de inclusión del material de apoyo del usuario y los del sistema. Además de buscar en este directorio, el compilador siempre busca en los directorios del sistema, que suelen ser /usr/include y sus subdirectorios.

De manera similar, la asignación siguiente:

```
LDFLAGS=-L$(DIR_APOYO) /lib
```

define una variable que informa al cargador de UNIX, ld, dónde debe buscar los archivos de biblioteca. Además de buscar en este directorio, el compilador siempre busca en los directorios del sistema, que suelen ser /usr/lib y sus subdirectorios.

La variable LIBS especifica al compilador qué bibliotecas de archivos objeto se deben usar para construir la aplicación.

```
LIBS= -lapoyo
```

Las líneas siguientes:

```
.c.o:  
$(CC) $(CFLAGS) -c $<
```

son una *regla de dependencia implícita* que especifica que un tipo de archivo se construye a partir de otro y describe cómo realizar su construcción. En este ejemplo las líneas anteriores especifican que los archivos .o se construyen a partir de los archivos .c mediante el compilador de C.

La asignación siguiente:

```
OBJS=reloj.o clock_task.o hardware.o
```

da valor a la variable OBJS con los nombres de los archivos objeto que forman la aplicación. La aplicación reloj tiene tres módulos objeto de aplicación: reloj.o, clock_task.o y hardware.o. Para una aplicación diferente habría que modificar esta línea para asignar a OBJS los módulos objeto de dicha aplicación.

El siguiente conjunto de líneas es el corazón del *makefile*. Estas líneas son *reglas de dependencia* que especifican cómo construir la aplicación. Por ejemplo, las siguientes líneas:

```
reloj: $(OBJS)
       $(CC) $(OBJS) $(LDFLAGS) $(LIBS) -o reloj
```

del *makefile* de Reloj especifican que *reloj* depende de *OBJS*, que tiene el valor *reloj.o*, *clock_task.o* y *hardware.o*. Además, depende de *LIBS*, que tiene el valor *libapoyo.a*. Si cualquiera de estos archivos objeto es más reciente que *reloj*, make ejecuta la línea de mandato que produce una nueva versión de *reloj* más reciente que los archivos objeto. Si todos los archivos objeto son más antiguos que *reloj*, significa que *reloj* está actualizado y, por tanto, no se necesita realizar ninguna acción.

Después de que se hayan construido todos los archivos objeto necesarios, su fecha de actualización será más reciente que la de *reloj*, por lo que se ejecuta el mandato que construye *reloj*.

La lista completa de opciones del programa make se puede obtener mediante el siguiente mandato:

```
man make
```

Para ejecutar una aplicación en UNIX basta con teclear el nombre de la aplicación en el *prompt* del intérprete de mandatos (véase Figura C.2).

C.2.2. Gestor de bibliotecas

Existe en UNIX y Linux una utilidad para la creación y mantenimiento de bibliotecas de archivos. Su principal uso es crear bibliotecas de objetos, es decir, agrupar un conjunto de objetos relacionados dentro una entidad lógica que se puede usar como un elemento de compilación.

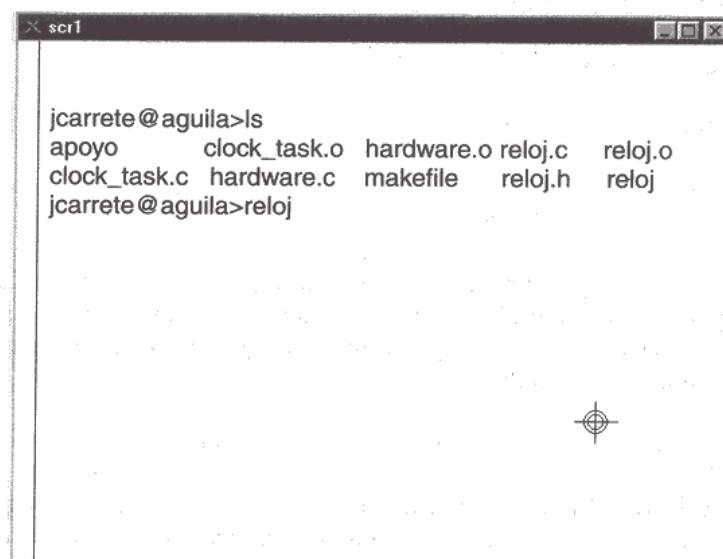


Figura C.2. Ejecución de *reloj* en UNIX.

En la línea de compilación se especifica la biblioteca (por convención, `libnombre.a`) en vez de los objetos que hay dentro de ella. El enlazador extraerá de la biblioteca los objetos que contienen las variables y funciones requeridas y los insertará dentro del programa ejecutable o incluirá referencias dinámicas a dichos objetos.

Formato del mandato: `ar [opciones] biblioteca archivos...`

Algunas opciones de este mandato son:

- d Elimina de la biblioteca los archivos especificados.
- r Añade (o reemplaza si ya existe) a la biblioteca los archivos especificados. Si no existe la biblioteca, se crea.
- ru Igual que -r, pero sólo reemplaza si el archivo es más nuevo.
- t Muestra una lista de los archivos contenidos en la biblioteca.
- v *Verbose*.
- x Extrae de la biblioteca los archivos especificados.

A continuación se muestran algunos ejemplos de aplicación de este mandato:

- Obtención de la lista de objetos contenidos en la biblioteca estándar de C.
`ar -tv /usr/lib/libc.a`
- Creación de una biblioteca con objetos que manejan distintas estructuras de datos.
`ar -rv $HOME/lib/libest.a pilas.o lista.o`
`ar -rv $HOME/lib/libest.a arbol.o hash.o`
- Creación de la biblioteca con material de apoyo que incluye el objeto del programa que simula el dispositivo reloj.
`ar -rv $HOME/apoyo/libapoyo.a dispositivo.o`

Hay dos formas posibles de compilar un programa que use una biblioteca: con nombre absoluto y con nombre relativo. En este último caso es necesario tener una variable de entorno para indicar dónde está la biblioteca, como se ha hecho en el *makefile anterior*. A continuación se muestran ejemplos de uso de ambas formas:

```
gcc -o reloj reloj.c clock_task.c hardware.c
                               -lm $HOME/apoyo/libapoyo.a
gcc -o reloj reloj.c clock_task.c hardware.c
                               -L$HOME/apoyo -lapoyo
```

Observe que la forma de especificar la biblioteca es distinta en ambos casos. Cuando se especifica el nombre absoluto, se indica el nombre completo del archivo donde está la biblioteca. Cuando se especifica un nombre relativo, sólo se indica una porción de dicho nombre: se asume una extensión válida y se elimina el prefijo `lib`.

C.2.3. Depuración de una aplicación en UNIX o Linux

En todas las versiones del sistema operativo UNIX, incluyendo Linux, existe un programa de depuración de programas. En el caso de Linux, existe un depurador denominado `gdb`. El depura-

dor permite que el usuario pueda controlar la ejecución de un programa y observar su comportamiento interno mientras ejecuta. Estos programas son muy útiles cuando se programa o prueban aplicaciones, como las prácticas de alumnos. Su uso puede ahorrar mucho tiempo de desarrollo y facilitar la tarea de los programadores.

Para poder depurar un programa, el compilador debe incluir información especial dentro del mismo. Por ello, para poder depurar un programa compilado con el `gcc`, se debe compilar con la opción `-g`.

A continuación se describen algunas de las funciones de un depurador genérico:

- Establecer puntos de parada en la ejecución del programa (*breakpoints*).
- Examinar el valor de variables.
- Ejecutar el programa línea a línea.

El formato del mandato para ejecutar el depurador en Linux es:

```
gdb programa_executable
```

En el caso del *makefile* del ejemplo, se ha creado un ejecutable denominado `reloj`. Para depurarlo habría que ejecutar el mandato:

```
gdb reloj
```

Algunos mandatos internos del `gdb` son:

`run`: Arranca la ejecución del programa.

`break`: Establece un *breakpoint* (un número de línea o el nombre de una función).

`list`: Imprime las líneas de código especificadas.

`print`: Imprime el valor de una variable.

`continue`: Continúa la ejecución del programa después de un *breakpoint*.

`next`: Ejecuta la siguiente línea. Si se trata de una llamada a función, la ejecuta completa.

`step`: Ejecuta la siguiente línea. Si se trata de una llamada a función, ejecuta sólo la llamada y se para al principio de la misma.

`quit`: Termina la ejecución del depurador.

C.3. ENTORNO DE PROGRAMACIÓN DEL VISUAL C++ DE MICROSOFT

Este compilador proporciona un entorno de programación y desarrollo totalmente integrado. A través del mismo se puede acceder a todas las utilidades necesarias para escribir, compilar y probar un programa. En la Figura C.3 se muestra la ventana principal del compilador Visual C++ de Microsoft. En la parte superior de la ventana hay un menú con mandatos tales como `File`, `Edit`, `View` y `Help`. Debajo de los elementos del menú hay una barra de herramientas con varios botones que proporcionan acceso rápido a muchos de los mandatos usados frecuentemente: `open`, `close`, `help`, etc. Se puede obtener ayuda de cualquiera de las características del IDE ejecutando el mandato `Help`.

C.3.1. Creación de un espacio de trabajo

El primer paso para crear un archivo de proyecto para una aplicación de sistemas operativos es crear un espacio de trabajo y un proyecto mediante el menú: `File->New`. Dentro de la caja de

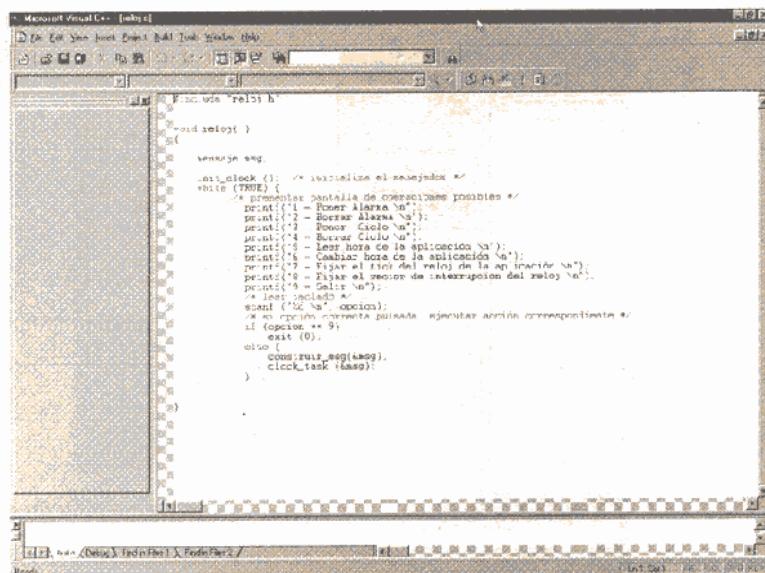


Figura C.3. Ventana principal del Visual C++ de Microsoft.

diálogo New debe especificarse el tipo de aplicación que se está construyendo. La elección correcta depende de que la aplicación requiera una consola. Cualquier aplicación de sistemas operativos que acepte entrada de teclado del usuario mediante el objeto de *iostream* *cin* o escriba en la pantalla mediante el objeto de *iostream* *cout* requiere una consola. Para este tipo de aplicación de sistemas operativos, la selección apropiada es Win32 Console Application. Si la aplicación no utiliza la biblioteca *iostream*, la selección apropiada es Win32 Application.

El paso final es especificar la posición del proyecto. En el ejemplo, se quiere que el proyecto resida en `c:\jesus\docencia\sos2\apendice2\reloj`, por lo que habrá que navegar hasta `c:\jesus\docencia\sos2\apendice2\reloj` y teclear `reloj` en el campo Project name:. Para crear el proyecto se debe pulsar el botón `OK`.
Para compilar el archivo `reloj.c` sólo hay que indicarlo en la opción Build, que se muestra en la Figura C.4.

El compilador de Visual C++ de Microsoft almacena la información de cómo construir una aplicación en un archivo de proyecto. Los archivos de proyecto tienen la extensión .dsp. En el caso del ejemplo, habría un nuevo archivo denominado reloj.dsp. Para salvar un archivo de proyecto se usa el menú: File->Save WorkSpace.

Después de crear el archivo de proyecto, inicialmente el proyecto está vacío y habrá que añadirle los archivos fuente del proyecto. El programa de ejemplo, Reloj, consta de varios módulos fuente: `reloj.c`, `clock_task.c` y `hardware.c`. Además, como material de apoyo a los alumnos, se proporciona una biblioteca y un archivo de definición de estructuras de datos (`*.h`). La Figura C.5 muestra un diagrama simplificado de la estructura de módulos de Reloj.

Para activar la caja de diálogo para añadir archivos al proyecto se ejecuta el mandato: Project->Add to Project->Files. Este paso activa la caja de diálogo que se muestra en la Figura C.6.

Para poder compilar y enlazar el proyecto es necesario definir correctamente las opciones del proyecto para poder encontrar los archivos a incluir y las bibliotecas que se van a usar. Esto se consigue mediante el mandato: Tools->Options. Para ello se ejecuta el menú Show

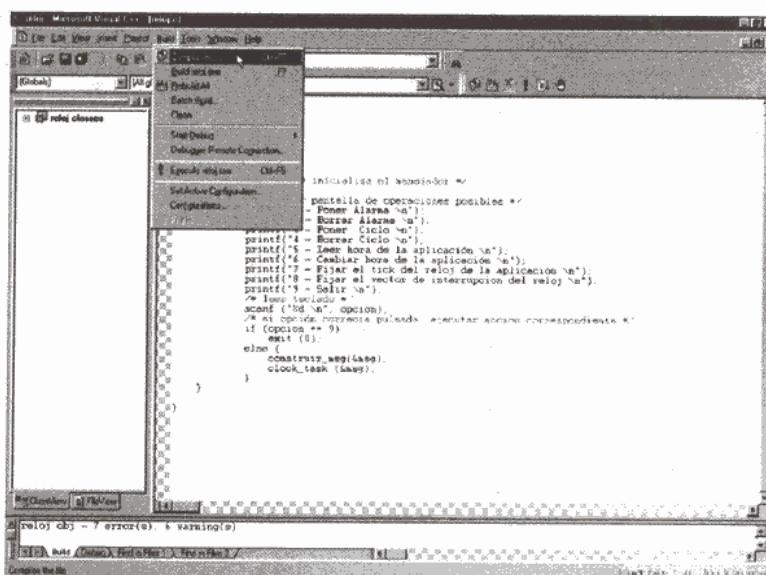


Figura C.4. Compilación de reloj en el Visual C++ de Microsoft.

directories for -> Include files y se teclea la ruta en la ventana Directories. En el ejemplo del reloj, esta ruta es c:\jesus\docencia\sos2\apendice2\reloj\apoyo. El valor por defecto del campo Directories indica la posición de los archivos de inclusión del sistema. La Figura C.7 muestra la caja de diálogo Options después de que se ha añadido la entrada del directorio de inclusión del programa reloj. De igual manera se puede definir el directorio para la biblioteca de apoyo, pero usando el menú Show directories for -> Libraries.

Para **compilar** y **enlazar** la aplicación se usa el menú: Build->Build reloj.exe. Este mandato hace que se compilen todos los módulos que han cambiado desde la última compilación y que después se enlacen todos los archivos objeto y las bibliotecas pedidas en una unidad ejecutable. El ejecutable se escribe en el archivo reloj.exe, puesto que ése es el nombre del proyecto.

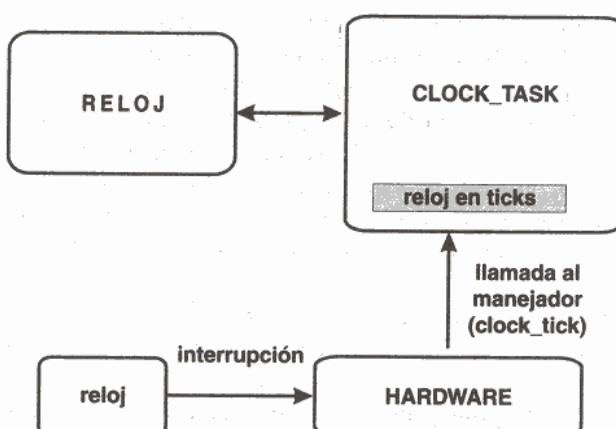


Figura C.5. Estructura de módulos del programa Reloj.

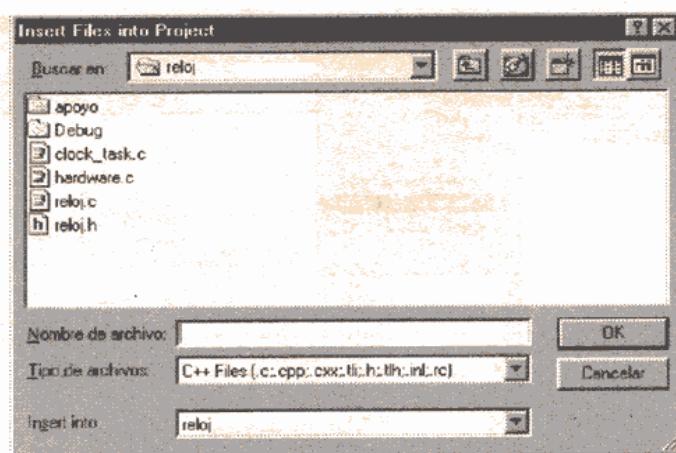


Figura C.6. Caja de diálogo «Insert Files into Project» del Visual C++ de Microsoft.

C.3.2. Ejecución de una aplicación en Visual C++

Para ejecutar una aplicación utilizando el C++ de Microsoft existen dos opciones:

- Ejecución desde el IDE de Visual C++.
- Ejecución desde una consola de MS-DOS.

Para ejecutar desde el IDE de Visual C++, lo único que hay que hacer es ejecutar la opción: Build->Execute reloj.exe. Este mandato solicita al IDE que ejecute el archivo reloj.exe usando para ello todos los recursos que sean necesarios. Si se usa entrada/salida por consola, se abre una ventana consola. La Figura C.8 muestra la ventana Build y la opción a usar para ejecutar el archivo reloj.exe.

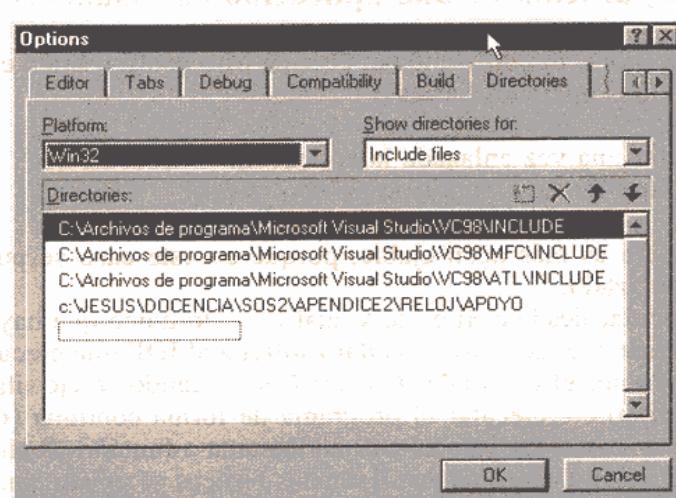


Figura C.7. Caja de diálogo «Options» del Visual C++ de Microsoft.

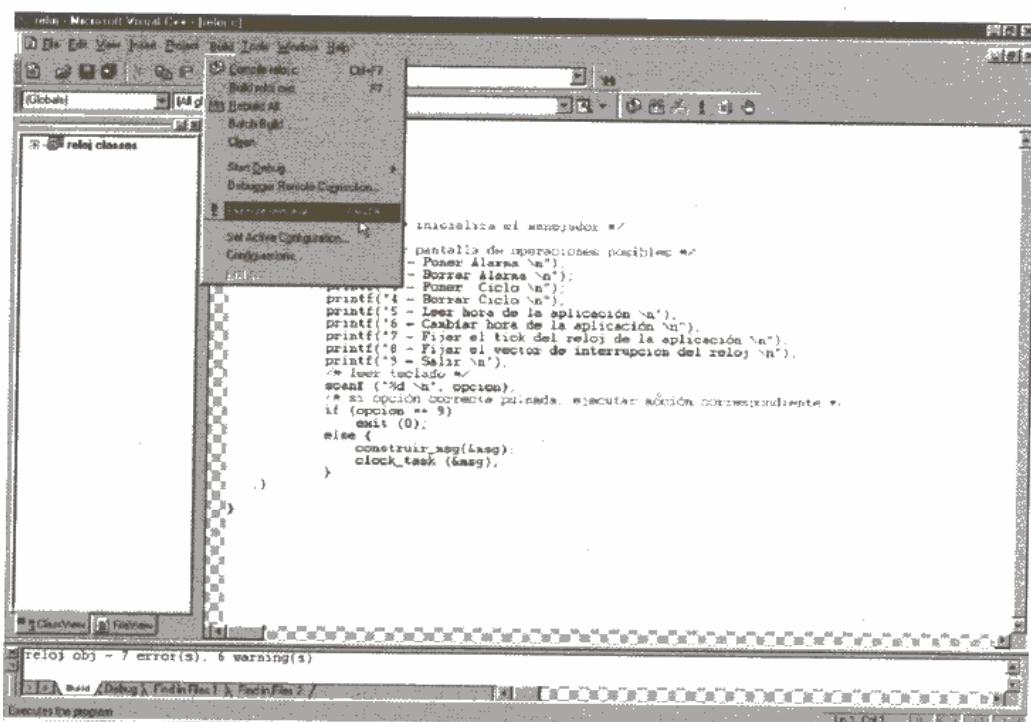


Figura C.8. Ejecución de reloj.exe en el Visual C++ de Microsoft.

La otra opción es ejecutar el programa desde una ventana consola o desde MS-DOS, que se puede crear activando el ícono de MS-DOS en el menú **Inicio**. Una vez en la ventana consola, se debe cambiar el directorio a la posición del ejecutable de la aplicación y a continuación ejecutar la aplicación tecleando su nombre.

C.3.3. Depuración de una aplicación en Visual C++

Depurar una aplicación utilizando el C++ de Microsoft, usando el IDE de Visual C++, es realmente fácil. Hay dos opciones básicas:

- Depurar una vez enlazada la aplicación.
 - Enlazar y depurar al mismo tiempo

La primera es más aconsejable, porque en este caso se puede estar seguro de que no hay errores de enlazado.

Para ejecutar desde el IDE de Visual C++, lo único que hay que hacer es ejecutar la opción: **Build->Start Debug**. Este mandato solicita al IDE que ejecute el archivo `reloj.exe` de forma controlada por el depurador (Figura C.9). Cuando se ejecuta esta opción, aparece una nueva ventana que permite ejecutar el programa de forma continua (*Go*), ejecutar hasta donde está el cursor (*Run to cursor*) o pararse en una llamada a función y saltar dentro del código fuente de la misma (*Step into*). Con estas tres llamadas básicas se puede controlar la ejecución del programa. Los errores salen en la pantalla de debajo del código fuente, mientras que el flujo de ejecución se controla en esta ventana.

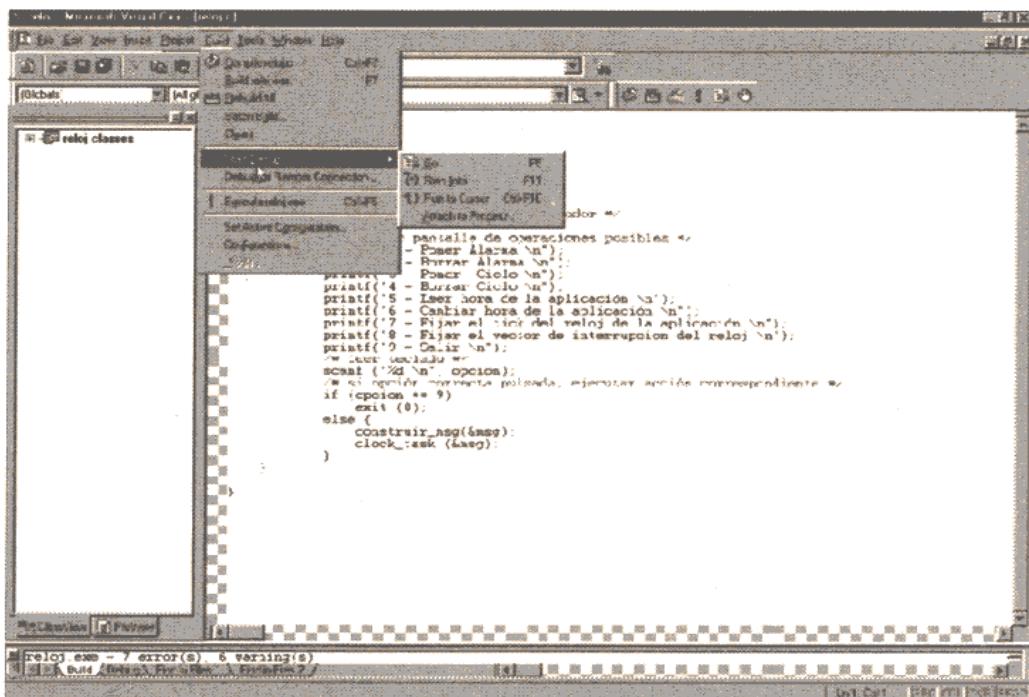


Figura C.9. Depuración de reloj.exe en el Visual C++ de Microsoft.

PRÁCTICAS DE SISTEMAS OPERATIVOS

De la base al diseño

Jesús CARRETERO PÉREZ

Félix GARCÍA CARBALLEIRA Fernando PÉREZ COSTOYA

Este libro está pensado como un texto general de prácticas de las asignaturas Sistemas Operativos y Diseño de Sistemas Operativos, pudiendo cubrir tanto la parte introductoria de los aspectos de programación de sistemas, como aspectos avanzados de programación y diseño de Sistemas Operativos (programación de shell scripts, programación con llamadas al sistema, programación de módulos del sistema operativo, etc.). Obviamente, este libro resulta un complemento natural al libro *Sistemas Operativos: una visión aplicada* escritos por algunos de los autores de esta propuesta y ya publicado por McGraw-Hill. Si bien **se puede usar de forma autónoma**, complementándolo con cualquier otro libro de teoría de Sistemas Operativos.

Este libro incluye básicamente tres aspectos novedosos frente a los libros clásicos de sistemas operativos:

- Una **panoplia de prácticas** que cubren todos los temas clásicos de la teoría. Se proponen varias prácticas por tema, lo que permitirá a los profesores cambiar o elegir las prácticas de forma cíclica. Las prácticas se han clasificado en tres niveles: INTRODUCCIÓN, INTERMEDIO y DISEÑO. Cada uno de estos niveles se ajusta bien a los requisitos necesarios para los distintos cursos de Sistemas Operativos.
- **Proyectos completos** de prácticas, incluyendo enunciados, soluciones, etc. Las soluciones solamente estarán disponibles para los profesores a través de la página web del libro.
- **Descripciones de las herramientas** a usar para resolver las prácticas y enlaces web a los orígenes de las mismas.
- **Página web**. Este libro tiene asociada la página web: <http://www.arcos.inf.uc3m.es/~ssoo-va/>. A través de esta página web se puede acceder a los materiales complementarios del libro, como material de ayuda para la realización de las prácticas, prácticas propuestas, enlaces a compiladores gratuitos del lenguaje C, etc. Dicha página incluye una sección dedicada a los profesores que vayan a usar el libro, con acceso restringido a los mismos mediante registro previo. Esta sección incluye:
 - a. Una guía para el profesor con los métodos de solución de prácticas.
 - b. Soluciones para las prácticas propuestas.
 - c. Material de apoyo para la realización de las prácticas.
 - d. Código fuente completo de las soluciones de las prácticas.



9 788448 136628

<http://www.mcgraw-hill.es>

McGraw-Hill Interamericana
de España, S. A. U.

A Subsidiary of The McGraw-Hill Companies



ISBN: 84-481-3662-4