

Programación Linux 2.0

***API de sistema
y funcionamiento de núcleo***

**Rémy CARD Éric DUMAS
Franck MÉVEL**

"Ouvrage publié avec l'aide du Ministère chargé de la Culture / Obra publicada con la ayuda del Ministerio francés responsable de la Cultura"

 **EYROLLES**



EDICIONES GESTIÓN 2000, S.A.

Prefacio

Hi dear Reader,

The book you hold in your hand will hopefully help you understand the Linux operating system kernel better, and you can delve into the strange and wonderful world of systems programming. Because it really *is* a strange and wonderful world, full of subtle details ranging from how to control the physical hardware to how to manage multiple different users at the same time with limited resources.

I certainly still, after more than five years, enjoy working on the Linux kernel, and hope that others will find it as fascinating as I have.

Linus Torvalds
University of Helsinki

Índice

PRÓLOGO	1
CAPÍTULO 1: LINUX: INTRODUCCIÓN	5
1 Historia	5
2 Descripción de Linux y sus funcionalidades	7
3 ¡Gracias, Internet!	8
4 Referencias	9
CAPÍTULO 2: PRESENTACIÓN GENERAL	13
1 Los diferentes tipos de sistemas operativos	13
2 Función del sistema operativo	14
2.1 Máquina virtual	14
2.2 Compartir el procesador	15
2.3 Gestión de la memoria	16
2.4 Gestión de recursos	16
2.5 Centro de comunicación de la máquina	16
3 Estructura general del sistema	17
4 Modo núcleo, modo usuario	18
4.1 Principio	18
4.2 Llamadas al sistema	19
CAPÍTULO 3: DESARROLLO BAJO LINUX	21
1 Los productos GNU	21
1.1 Presentación	21
1.2 Herramientas GNU	22
1.2.1 Presentación	22
1.2.2 gcc	22
2 Las herramientas del desarrollador	23
2.1 Las diferentes fases de la compilación	23

2.2	gcc	25
2.3	Depurador	27
2.4	strace	27
2.5	make	28
2.5.1	Presentación	28
3	Formato de los ejecutables	28
3.1	a.out: el ancestro	29
3.2	ELF	29
3.2.1	Bienvenida al mundo ELF	29
3.2.2	ELF y su formato	30
4	Presentación de las bibliotecas	32
4.1	Utilidad	32
4.2	Utilización de bibliotecas	33
5	Organización del código fuente del núcleo	33
5.1	Núcleo	33
5.2	Archivos de cabecera	34
6	Funcionamiento del núcleo	36
6.1	Llamadas al sistema	36
6.1.1	Implementación de una llamada al sistema	36
6.1.2	Creación de una llamada al sistema	37
6.1.3	Códigos de retorno	39
6.2	Funciones de utilidad	40
6.2.1	Manipulación de espacio de direccionamiento	41
6.2.2	Asignaciones y liberaciones	41
CAPÍTULO 4: PROCESOS		43
1	Conceptos básicos	43
1.1	Noción de proceso	43
1.2	Estado de un proceso	44
1.3	Atributos de un proceso	45
1.4	Identificadores de un proceso	45
1.5	Filiación	46
1.6	Grupos de procesos	46
1.7	Sesiones	48
1.8	Multiprogramación	48
2	Llamadas al sistema de base	49
2.1	Creación de procesos	49
2.2	Finalización del proceso actual	50
2.3	Espera de la finalización de un proceso hijo	50
2.4	Lectura de los atributos del proceso actual	53
2.5	Modificación de atributos	54
2.6	Información de contabilidad	56

2.7	Límites	57
2.8	Grupos de procesos	58
2.9	Sesiones	59
2.10	Ejecución de programa	59
3	Conceptos avanzados	61
3.1	Coordinador	61
3.2	Personalidades	62
3.3	Clonado	63
4	Llamadas al sistema complementarias	64
4.1	Cambio de personalidad	64
4.2	Modificación de la coordinación	65
4.3	Prioridades de los procesos	67
4.4	Control de la ejecución de un proceso	68
4.5	Clonado	70
5	Presentación general de la implementación	74
5.1	La tabla de procesos	74
5.1.1	<i>Descriptor de proceso</i>	74
5.1.2	<i>Organización de la tabla de procesos</i>	78
5.1.3	<i>Manipulación de la tabla de procesos</i>	79
5.2	Registros del procesador	79
5.3	Sincronización de procesos	81
5.3.1	<i>Principio</i>	81
5.3.2	<i>Colas de espera</i>	81
5.3.3	<i>Semáforos</i>	81
5.4	Los timers	83
5.5	Ámbitos de ejecución	83
5.6	Formatos de los archivos ejecutables	84
6	Presentación detallada de la implementación	85
6.1	Funciones internas	85
6.1.1	<i>Sincronización de procesos</i>	85
6.1.2	<i>Coordinación</i>	86
6.1.3	<i>Los timers</i>	88
6.1.4	<i>Espera con demora</i>	88
6.2	Implementación de las llamadas al sistema	89
6.2.1	<i>Creación de procesos</i>	89
6.2.2	<i>Terminación de proceso</i>	90
6.2.3	<i>Obtención de los atributos</i>	92
6.2.4	<i>Modificación de atributos</i>	92
6.2.5	<i>Grupos de procesos y sesiones</i>	93
6.2.6	<i>Control de procesos</i>	93

CAPÍTULO 5: SEÑALES	97
1 Conceptos básicos	97
1.1 Introducción	97
1.2 Definición de señales	97
1.3 Lista de señales	99
1.4 Visualización de las señales	100
2 Llamadas al sistema de base	101
2.1 Emisión de una señal	102
2.2 Desviación de una señal	103
2.3 Espera de una señal	105
3 Conceptos avanzados	106
3.1 Las llamadas al sistema interrumpibles	106
3.2 Las funciones reentrantes	106
3.3 Los grupos de señales	107
4 Llamadas al sistema complementarias	108
4.1 La gestión avanzada de las señales	108
4.1.1 La interrupción de las llamadas al sistema	108
4.1.2 El bloqueo de las señales	108
4.1.3 El desvío de señales	111
4.1.4 La espera de señales	112
4.2 La gestión de las alarmas	113
4.2.1 La llamada al sistema alarm	113
4.2.2 Una gestión más precisa del tiempo	114
4.3 La señal SIGCHLD	117
5 Presentación general de la implementación	119
5.1 Las estructuras de datos	119
5.1.1 La creación y la terminación de procesos	119
5.1.2 La emisión de señales	120
5.1.3 La recepción de las señales	120
6 Presentación detallada de la implementación	121
6.1 Funciones de biblioteca	121
6.1.1 La gestión de los grupos de señales	121
6.1.2 La función raise	122
6.2 Las llamadas al sistema	122
6.2.1 El bloqueo de las señales	122
6.2.2 El desvío de señales	123
6.2.3 Las señales en espera	124
6.2.4 La suspensión del proceso en espera de señales	124
6.2.5 El envío de señales	124
6.2.6 La gestión de las alarmas	125

CAPÍTULO 6: SISTEMAS DE ARCHIVOS	127
1 Conceptos básicos	127
1.1 Organización de los archivos	127
1.2 Tipos de archivos	128
1.3 Enlaces con los archivos	129
1.4 Atributos de archivos	130
1.5 Primitivas de entrada/salida	131
1.6 Descriptores de entradas/salidas	132
2 Llamadas básicas al sistema	132
2.1 Entradas/salidas sobre archivos	132
2.1.1 Apertura y cierre de archivos	132
2.1.2 Lectura y escritura de datos	134
2.1.3 Posicionamiento en un archivo	136
2.1.4 Guardar los datos modificados	138
2.2 Manipulación de archivos	139
2.2.1 Creación de enlaces	139
2.2.2 Supresión de archivos	139
2.2.3 Cambio de nombre de un archivo	140
2.2.4 Cambio de tamaño de un archivo	141
2.2.5 Derechos de acceso sobre un archivo	142
2.2.6 Modificación del usuario propietario	143
2.3 Gestión de directorios	144
2.3.1 Creación de directorios	144
2.3.2 Supresión de directorios	145
2.3.3 Directorio actual	145
2.3.4 Directorio raíz local	146
2.3.5 Exploración de los directorios	147
2.4 Enlaces simbólicos	151
3 Conceptos avanzados	152
3.1 i-nodos	152
3.2 Descriptores de entrada/salida	153
3.3 Compartir descriptores	154
3.4 Bloqueo de archivos	156
3.5 Montaje de sistemas de archivos	157
3.6 Cuotas de disco	157
4 Llamadas al sistema complementarias	159
4.1 Lectura y escritura de varias memorias intermedias	159
4.2 Duplicación de descriptor de entrada/salida	160
4.3 Atributos de archivos	161
4.4 Fechas asociadas a los archivos	164
4.5 Propiedades de los archivos abiertos	165

4.6	Control del proceso bdfush	166
4.7	Bloqueo	168
4.7.1	Bloqueo de un archivo	168
4.7.2	Bloqueo de una sección de un archivo	168
4.8	Montaje de sistemas de archivos	169
4.9	Informaciones sobre un sistema de archivos	171
4.10	Información sobre los tipos de sistemas de archivos soportados	172
4.11	Manipulación de cuotas de disco	173
5	Presentación general de la implementación	175
5.1	El sistema virtual de archivos	175
5.1.1	Principio	175
5.1.2	Operaciones aseguradas por el SVA	176
5.2	Estructuras del SVA	177
5.2.1	Tipos de sistemas de archivos	177
5.2.2	Sistemas de archivos montados	178
5.2.3	I-nodos en curso de utilización	179
5.2.4	Archivos abiertos	181
5.2.5	Archivos abiertos por un proceso	181
5.2.6	Descriptores de bloqueos	182
5.2.7	Descriptores de cuotas de disco	183
5.2.8	Memorias intermedias del búfer caché	184
5.2.9	Caché de nombres	186
5.3	Operaciones genéricas	187
5.3.1	Operaciones sobre superbloques	187
5.3.2	Operaciones sobre i-nodos	189
5.3.3	Operaciones sobre archivos abiertos	192
5.3.4	Operaciones sobre cuotas de disco	194
6	Presentación detallada de la implementación	195
6.1	Funciones internas del SVA	195
6.1.1	Gestión de descriptores de sistemas de archivos montados	195
6.1.2	Gestión de los i-nodos	197
6.1.3	Gestión de descriptores de archivos abiertos	201
6.1.4	Gestión de descriptores de cuotas de disco	202
6.1.5	El caché de nombres	207
6.1.6	Funciones de soporte de gestión de i-nodos	209
6.1.7	Gestión de los nombres de archivos	210
6.1.8	Gestión de bloqueos	211
6.2	El búfer caché	213
6.2.1	Presentación	213
6.2.2	Gestión de las listas de memorias intermedias	213
6.2.3	Entradas/salidas	216

6.2.4	<i>Modificación del tamaño del búfer caché</i>	217
6.2.5	<i>Funciones de gestión de dispositivos</i>	219
6.2.6	<i>Funciones de acceso a las memorias intermedias</i>	219
6.2.7	<i>Reescritura de las memorias intermedias modificadas</i>	220
6.2.8	<i>Gestión de clusters</i>	222
6.2.9	<i>Inicialización del búfer caché</i>	223
6.3	<i>Implementación de las llamadas al sistema</i>	224
6.3.1	<i>Organización de los archivos fuente</i>	224
6.3.2	<i>Manipulación de archivos</i>	224
6.3.3	<i>Gestión de los atributos de archivos</i>	225
6.3.4	<i>Entrada/salida sobre archivos</i>	226
6.3.5	<i>Lectura de directorio</i>	228
6.3.6	<i>Gestión de bloqueos</i>	228
6.3.7	<i>Gestión del búfer caché</i>	230
6.3.8	<i>Gestión de los sistemas de archivos</i>	230
6.3.9	<i>Manipulación de descriptores de entrada/salida</i>	232
6.3.10	<i>Herencia de descriptores</i>	233
6.3.11	<i>Cambio de directorio</i>	234
6.3.12	<i>Gestión de las cuotas de disco</i>	234
6.4	<i>Sistemas de archivos soportados</i>	236
6.5	<i>Implementación del sistema de archivos Ext2</i>	237
6.5.1	<i>Características de Ext2</i>	237
6.5.2	<i>Estructura física de un sistema de archivos Ext2</i>	238
6.5.3	<i>El superbloque</i>	239
6.5.4	<i>Los descriptores de grupo de bloques</i>	240
6.5.5	<i>Estructura de un i-nodo</i>	241
6.5.6	<i>Entrada de directorio</i>	242
6.5.7	<i>Operaciones vinculadas al sistema de archivos</i>	243
6.5.8	<i>Asignación y liberación de bloques e i-nodos</i>	244
6.5.9	<i>Gestión de los i-nodos en disco</i>	246
6.5.10	<i>Gestión de directorios</i>	248
6.5.11	<i>Entradas/salidas sobre archivos</i>	249
6.6	<i>Implementación del sistema de archivos /proc</i>	250
6.6.1	<i>Presentación</i>	250
6.6.2	<i>Entradas de /proc</i>	252
6.6.3	<i>Operaciones sobre sistema de archivos</i>	253
6.6.4	<i>Gestión de directorios</i>	254
6.6.5	<i>Operaciones sobre i-nodos y sobre archivos</i>	255

CAPÍTULO 7: ENTRADAS/SALIDAS	257
1 Conceptos	257
2 Llamadas al sistema	258

2.1	Creación de un archivo especial	258
2.2	Entradas/salidas sobre dispositivos	259
2.3	Multiplexado de entradas/salidas	260
2.4	Operación de control sobre un dispositivo	262
3	Presentación general de la implementación	262
3.1	Dispositivos soportados por el núcleo	262
3.2	Entradas/salidas en disco	263
4	Presentación detallada de la implementación	264
4.1	Gestión de los dispositivos efectuados	264
4.2	Entradas/salidas de disco	265
4.3	Entradas/salidas sobre dispositivos en modo bloque	266
4.4	Multiplexado de entradas/salidas	266
4.4.1	<i>Principio</i>	<i>266</i>
4.4.2	<i>Funciones de utilidad</i>	<i>267</i>
4.4.3	<i>La operación sobre archivo select</i>	<i>268</i>
4.4.4	<i>Implementación de la primitiva select</i>	<i>269</i>
4.5	Gestión de las interrupciones	269
4.6	Gestión de los canales DMA	270
4.7	Acceso a los puertos de entrada/salida	271
4.8	Ejemplo de dispositivo en modo bloque: el disco en memoria	272
4.8.1	<i>Presentación</i>	<i>272</i>
4.8.2	<i>Acceso al contenido del disco en memoria</i>	<i>272</i>
4.8.3	<i>Operaciones sobre archivos</i>	<i>273</i>
4.8.4	<i>Inicialización</i>	<i>273</i>
4.8.5	<i>Carga de disco de memoria</i>	<i>274</i>
4.9	Ejemplo de dispositivo en modo carácter: la impresora	275
4.9.1	<i>Funcionamiento</i>	<i>275</i>
4.9.2	<i>Descripción de los puertos</i>	<i>275</i>
4.9.3	<i>Funciones de gestión de la impresora con exploración</i>	<i>276</i>
4.9.4	<i>Funciones de gestión de la impresora con interrupciones</i>	<i>276</i>
4.9.5	<i>Operaciones de entrada/salida sobre archivo</i>	<i>277</i>
4.9.6	<i>Funciones de inicialización</i>	<i>278</i>
CAPÍTULO 8: GESTIÓN DE LA MEMORIA		279
1	Conceptos básicos	279
1.1	Espacio de direccionamiento de un proceso	279
1.2	Asignación de memoria	281
2	Llamadas al sistema de base	281
2.1	Cambio del tamaño del segmento de datos	281
2.2	Funciones de asignación y liberación de memoria	282
3	Conceptos avanzados	283
3.1	Regiones de memoria	283

3.2	Protección de la memoria	284
3.3	Bloqueo de zonas de memoria	285
3.4	Proyección de archivos en memoria	286
3.5	Dispositivos de swap	287
4	Llamadas al sistema complementarias	287
4.1	Protección de páginas de memoria	287
4.2	Bloqueo de páginas de memoria	288
4.3	Proyección en memoria	289
4.4	Sincronización de páginas en memoria	292
4.5	Gestión de los dispositivos de swap	293
5	Presentación general de la implementación	294
5.1	Gestión de las tablas de página	294
5.1.1	Segmentación	294
5.1.2	Paginación	296
5.1.3	Tablas de páginas gestionadas por Linux	298
5.1.4	Compartir páginas	298
5.1.5	Tipos	299
5.2	Gestión de las páginas de memoria	299
5.2.1	Descriptores de página	299
5.3	Asignación de memoria para el núcleo	301
5.3.1	Asignación de páginas de memoria	301
5.3.2	Asignación de zonas de memoria	302
5.4	Espacios de direccionamiento de los procesos	304
5.4.1	Descriptores de regiones de memoria	304
5.4.2	Descriptores de espacio de direccionamiento	305
5.4.3	Organización de los descriptores de regiones	306
5.5	Asignación de memoria no contigua al núcleo	309
5.6	Gestión del swap	309
5.6.1	Formato de los dispositivos de swap	309
5.6.2	Descriptores de dispositivos de swap	310
5.6.3	Direcciones de entradas del swap	311
5.6.4	Selección de páginas a descartar	311
5.7	Operaciones de memoria	312
6	Presentación detallada de la implementación	314
6.1	Asignación de memoria para el núcleo	314
6.1.1	Asignación de páginas de memoria	314
6.1.2	Asignación de regiones de memoria	315
6.2	Gestión de las tablas de páginas	315
6.3	Gestión de las regiones de memoria	319
6.4	Tratamiento de las excepciones	320
6.5	Acceso al espacio de direccionamiento de los procesos	322

6.6	Modificación de regiones de memoria	323
6.6.1	Creación y supresión de regiones de memoria	323
6.6.2	Bloqueo de regiones de memoria	324
6.6.3	Modificaciones de protecciones de memoria	326
6.6.4	Reasignar regiones de memoria	327
6.7	Creación y supresión de espacio de direccionamiento	328
6.8	Proyección de archivos en memoria	329
6.9	Gestión del swap	332
6.9.1	Gestión de los dispositivos de swap	332
6.9.2	Entrada/salida de páginas de swap	333
6.9.3	Eliminación de páginas de memoria	334
CAPÍTULO 9: TERMINALES POSIX		337
1	Conceptos básicos	337
1.1	Presentación general	337
1.2	Configuración de un terminal	339
1.2.1	Modo canónico - modo no canónico	339
1.2.2	La estructura <code>termios</code>	340
1.2.3	Funcionamiento de una comunicación	341
1.2.4	Configuración de los modos de entrada	341
1.2.5	Configuración de los modos de salida	342
1.2.6	Configuración de los modos de control	343
1.2.7	Configuración de los modos locales	344
1.3	Los caracteres especiales y la tabla <code>c_cc</code>	345
1.4	El mandato <code>stty</code>	346
1.5	El mandato <code>setserial</code>	347
1.6	Grupos de terminales - sesión	347
1.7	Pseudoterminales	348
2	Las funciones POSIX	349
2.1	Acceder y modificar los atributos de un terminal	349
2.2	Un ejemplo de configuración de línea	351
2.3	La velocidad de transmisión	352
2.4	Control de línea	355
2.5	Identificación del terminal	357
2.6	Grupos de procesos	359
2.7	Pseudoterminales	360
3	Organización en el núcleo - estructuras de datos	363
3.1	Organización	363
3.2	Estructuras de datos internas	364
3.3	La estructura <code>tty_struct</code>	364
3.4	La estructura <code>tty_driver</code>	366
3.5	La estructura <code>tty_ldisc</code>	369

3.6	La estructura winsize	371
3.7	La estructura tty_flip_buffer	371
4	Implementación	372
4.1	La inicialización	372
4.2	Las entradas/salidas en los terminales	373
4.3	Otras funciones generales	374
4.4	Gestión de la desconexión	375
4.5	La gestión de la disciplina de la línea	376
4.6	Los pseudoterminales	378
CAPÍTULO 10: COMUNICACIÓN POR TUBERÍAS		381
1	Conceptos básicos	381
1.1	Presentación	381
1.2	Tuberías anónimas y tuberías con nombre	382
2	Llamadas al sistema	383
2.1	Tuberías anónimas	383
2.1.1	Creación de una tubería	383
2.1.2	Un ejemplo de uso	385
2.1.3	Creación de una tubería y ejecución de un programa	388
2.2	Las tuberías con nombre	388
3	Presentación general de la implementación	390
3.1	Las tuberías con nombre	390
3.2	Las tuberías anónimas	391
4	Presentación detallada de la implementación	391
4.1	Creación del i-nodo de una tubería	391
4.2	Creación de una tubería con nombre	391
4.3	Creación de una tubería anónima	392
4.4	Operaciones de entradas/salidas	392
CAPÍTULO 11: IPC SYSTEM V		395
1	Conceptos básicos	395
1.1	Introducción	395
1.2	La gestión de claves	396
1.3	Los derechos de acceso	398
2	Llamadas al sistema	398
2.1	Colas de mensajes	399
2.1.1	Las estructuras básicas	399
2.1.2	Creación y búsqueda de colas de mensajes	400
2.1.3	Control de las colas de mensajes	401
2.1.4	Emisión de mensajes	402
2.1.5	Recepción de mensajes	403
2.1.6	Un ejemplo de uso	404

2.2	Semáforos	411
2.2.1	Las estructuras básicas	412
2.2.2	Creación y búsqueda de grupos de semáforos	413
2.2.3	Operaciones sobre los semáforos	414
2.2.4	El control de los semáforos	415
2.3	Memorias compartidas	417
2.3.1	Las estructuras básicas	417
2.3.2	Creación y búsqueda de una zona de memoria compartida	418
2.3.3	Vinculación de una zona de memoria	419
2.3.4	Desvincular una zona de memoria	420
2.3.5	Control de las zonas de memoria compartidas	420
2.3.6	Interacción con otras llamadas al sistema	421
2.3.7	Ejemplo de uso	421
3	Conceptos avanzados	428
3.1	Opción de compilación del núcleo	428
3.2	Los programas ipcs e ipcrm	429
3.2.1	ipcs	429
3.2.2	ipcrm	430
4	Presentación general de la implementación	431
4.1	Funciones comunes	431
4.2	Algoritmos	432
4.3	Gestión de las claves	433
5	Presentación detallada de la implementación	433
5.1	Colas de mensajes	433
5.1.1	Representación interna de las colas de mensajes	434
5.1.2	La inicialización	435
5.1.3	Creación de una cola de mensajes	435
5.1.4	El envío de un mensaje	435
5.1.5	La recepción de un mensaje	435
5.1.6	El control de una cola de mensajes	435
5.2	Semáforos	436
5.2.1	Representación interna de los semáforos	436
5.2.2	La inicialización	436
5.2.3	Creación de semáforos	437
5.2.4	Control de los semáforos	437
5.2.5	Modificación de los valores de semáforos	438
5.2.6	La finalización	438
5.3	Memorias compartidas	439
5.3.1	Representación interna de las memorias compartidas	439
5.3.2	La inicialización	440
5.3.3	Creación de una zona de memoria compartida	440

5.3.4 Vinculación de una zona de memoria	441
5.3.5 Desvinculación de una zona de memoria compartida	441
5.3.6 Control de las zonas de memoria compartidas	441
5.3.7 Herencia de las zonas de memoria compartidas	442
CAPÍTULO 12: LOS MÓDULOS CARGABLES	443
1 Conceptos básicos	443
1.1 Presentación	443
1.2 Compilación	444
1.3 Operaciones en línea de mandatos: la carga manual	445
1.4 Carga dinámica	446
2 Conceptos avanzados	448
2.1 La realización de un módulo cargable	448
2.1.1 Presentación	448
2.1.2 Ejemplo	448
2.1.3 Compilación y ejecución	450
2.2 Las llamadas al sistema específicas de los módulos	451
2.2.1 <i>get_kernel_syms</i>	451
2.2.2 <i>create_module</i>	452
2.2.3 <i>init_module</i>	452
2.2.4 <i>delete_module</i>	454
3 Implementación de los módulos cargables	454
3.1 Presentación	454
3.2 Implementación de las llamadas al sistema	455
3.2.1 <i>create_module</i>	455
3.2.2 <i>init_module</i>	455
3.2.3 <i>delete_module</i>	455
3.2.4 <i>get_kernel_syms</i>	456
3.3 Gestión de los archivos virtuales	456
3.4 Funciones anexas	457
4 Carga automática de módulos (kernel)	457
4.1 Presentación	457
4.2 Detalle de la implementación	458
CAPÍTULO 13: ADMINISTRACIÓN DEL SISTEMA	461
1 Conceptos básicos	461
1.1 Informaciones destinadas a los procesos	461
1.2 Informaciones que controlan la ejecución	462
1.3 Cambio de estado del núcleo	463
1.4 Configuración dinámica del sistema	463
1.4.1 Presentación	463
1.4.2 Otro método: <i>/proc/sys</i>	463

2 Llamadas al sistema de administración	464
2.1 Informaciones respecto a la estación	464
2.2 Control de la ejecución	467
2.2.1 <i>El tiempo</i>	467
2.2.2 <i>Modo de visualización de los mensajes del núcleo</i>	469
2.2.3 <i>Estado de la máquina</i>	471
2.2.4 <i>Sincronización de relojes</i>	473
2.3 Cambio de estado del núcleo	474
2.4 Configuración dinámica del sistema	474
3 Implementación	477
3.1 sethostname, gethostname, setdomainname y uname	477
3.2 reboot	478
3.3 syslog	478
3.4 gettimeofday, settimeofday, time y stime	479
3.4.1 <i>time</i>	479
3.4.2 <i>stime</i>	479
3.4.3 <i>gettimeofday</i>	479
3.4.4 <i>settimeofday</i>	480
3.5 sysctl	480
3.6 adjtimex	481
APÉNDICE A: FASES DE UNA COMPILACIÓN C	483
1 Preprocesador	483
2 Compilador	484
3 Ensamblador	485
4 Enlazador	485
APÉNDICE B: UTILIZACIÓN DE GDB	487
APÉNDICE C: UTILIZACIÓN DE MAKE	491
1 Funcionamiento de make	491
2 Escritura de un archivo makefile	491
APÉNDICE D: GESTIÓN DE LAS BIBLIOTECAS	493
1 Herramientas de creación y de manipulación	493
2 Bibliotecas estáticas	494
3 Bibliotecas dinámicas a.out	495
4 Bibliotecas dinámicas ELF	495
5 La carga dinámica de bibliotecas	496
BIBLIOGRAFÍA	499

Prólogo

Desde hace algunos años, el sistema operativo Linux se implanta cada vez más en los medios universitarios e industriales. Este clónico de Unix cuya estabilidad y potencia le permiten rivalizar sin complejos con los otros sistemas operativos comerciales, posee ciertas especificidades que lo hacen muy interesante. La libre difusión de su código fuente en Internet ha contribuido ampliamente a popularizar Linux, que se ha convertido así en un tema de estudio ideal.

Organización del libro

Esta obra, cuya redacción ha ocupado alrededor de un año, tiene como objetivo presentar los diferentes aspectos de Linux 2.0 desde el punto de vista de un desarrollador, y asimismo analizar su núcleo, que constituye el centro del sistema operativo Linux. Este libro no es solamente un libro de programación de sistemas para Unix (y para Linux en particular): es también un medio para comprender el funcionamiento interno del núcleo.

El libro se divide en 13 capítulos que detallan la mayor parte de aspectos de este sistema, excepto la gestión de los protocolos de red, que necesitarían un libro completo.

Los tres primeros capítulos son introductorios:

- El primer capítulo presenta el origen y las funcionalidades del sistema Linux. También proporciona una lista de referencias.
- El segundo capítulo es una presentación rápida de los distintos componentes de un sistema operativo y de Linux en particular. En él se explica la noción de modo núcleo y modo usuario y se detalla el mecanismo de las llamadas al sistema.

Contiene una descripción de la organización de las fuentes del núcleo y algunas referencias sobre la instalación y la manipulación de las herramientas básicas de Linux.

- El tercer capítulo describe las herramientas de desarrollo disponibles bajo Linux, e introduce las nociones de programación del sistema.

Los capítulos siguientes describen en detalle el sistema Linux. Cada uno de ellos presenta primero la interfaz de sistema relacionada con el concepto que describe seguida de la implementación de dicho concepto en el núcleo Linux:

- Los capítulos 4 y 5 tratan la gestión de los procesos y las señales.
- El capítulo 6 presenta la gestión de archivos. La implementación describe el sistema virtual de archivos y dos ejemplos de sistemas de archivos, Ext2 y /proc.
- El capítulo 7 aborda las entradas/salidas sobre dispositivos y analiza particularmente el controlador de impresora y el gestor de disco en memoria.
- El capítulo 8 presenta un concepto importante de los sistemas actuales, la gestión de la memoria virtual.
- El capítulo 9 se interesa por la gestión de los terminales.
- Los capítulos 10 y 11 describen las comunicaciones interprocesos, el pipeline y los IPC System V.
- Los dos últimos capítulos tratan sobre la carga dinámica de módulos en el núcleo y sobre funciones útiles para la administración del sistema.

Para terminar, cuatro apéndices detallan la utilización de las herramientas de desarrollo bajo Linux (compilador de C, depurador, make y herramientas de gestión de bibliotecas).

Cada capítulo se compone de varias partes: las primeras presentan los conceptos expuestos, así como las llamadas de sistema tratadas, mientras que las siguientes detallan el funcionamiento del núcleo, describiendo las estructuras de datos utilizadas y las funciones internas de Linux.

Esta estructuración permite al lector recorrer un capítulo yendo de los conceptos generales hasta los detalles de la implementación. La lectura puede llevarse a cabo, pues, de manera secuencial por capítulos, pero también de manera aleatoria, en función de las necesidades, del tiempo, la curiosidad...

Diversos ejemplos (en lenguaje C), figuras y tablas completan el contenido de esta obra que pretende ser clara, completa e ilustrada. El lector encontrará al final una bibliografía relativa a los sistemas operativos y a la documentación técnica existentes sobre Linux.

Público y prerrequisitos

Este libro va destinado a los desarrolladores de sistemas, ya sean estudiantes, profesionales o investigadores en informática. Puede utilizarse como apoyo de cursos sobre sistemas, como ayuda en el desarrollo de programas, o en el desarrollo en el núcleo. La propia estructura de los capítulos lo hace accesible a un público muy variado: el principiante podrá limitarse al principio a leer las primeras partes de un capítulo, mientras que el hacker (programador del sistema) se centrará más bien en el funcionamiento interno del núcleo.

Sin embargo, es necesario precisar que se requiere el conocimiento del lenguaje C y de su biblioteca estándar. Además, el lector debe estar familiarizado con los conceptos generales relacionados con los sistemas operativos en general, y con Unix en particular, para comprender los ejemplos y la descripción del funcionamiento interno del núcleo.

La lectura de estas partes es más ardua, por lo que se aconseja estudiar el código fuente del núcleo de forma paralela.

Convenciones utilizadas

Todas las llamadas al sistema presentadas en un capítulo se agrupan en un cuadro al principio de cada capítulo:

Primitivas detalladas

llamada1, llamada2, ...

Se utilizan tablas para aclarar la descripción de las estructuras y uniones presentadas. También se usan para las listas de errores correspondientes a las llamadas al sistema, las listas de constantes y las listas de macroinstrucciones.

Los prototipos de las funciones de la biblioteca estándar del C y las diferentes llamadas al sistema presentadas se escriben en forma de teletipo.

Los nombres de llamadas al sistema, de funciones de biblioteca y de mandatos se representan en cursiva. Se utiliza un tipo de letra teletipo para los nombres de tipos, de estructuras, de variables y de funciones.

Agradecimientos

La redacción de este libro no habría sido posible sin la colaboración de numerosas personas.

Agradecemos a todos los desarrolladores de Linux, y especialmente al primero de ellos, Linus Torvalds. Sin ellos, este sistema operativo no habría nacido.

Nuestro agradecimiento a Benjamin Bayart por su valiosa ayuda sobre el uso y la programación de LaTeX.

Queremos agradecer a las personas que han aceptado hacer el trabajo ingrato de relectura de esta obra y han participado en gran medida a mejorar su contenido gracias a sus numerosas críticas constructivas y sugerencias: Éric Commelin, Pierre David, Pierre Ficheux, Thomas Quinot, Pierre-Guillaume Raverdy, y Stéphan Voisin.

Finalmente, nuestro agradecimiento a Julien Simon, que ha contribuido ampliamente a la definición y a la estructuración de este libro.

Evidentemente, un libro así contiene necesariamente errores o imprecisiones. Invitamos a los lectores a transmitirnos sus comentarios, sugerencias y correcciones de errores por correo electrónico, a la dirección `info@gestion2000.com`.

Rémy Card (Remy.Card@linux.org)

Éric Dumas (dumas@freenix.fr)

Franck Mével (mevel@Linux.EU.Org)

Capítulo 1

Linux: Introducción

Linux, Linux, Linux... pero, ¿por qué el nombre Linux? Antes de adentrarnos en los vericuetos de este sistema operativo y de su núcleo, se impone una breve presentación de su historia y de su concepción, pues su historia es particularmente original en el mundo de los sistemas operativos. A continuación de esta presentación se exponen las funcionalidades del núcleo y se proporciona una lista de referencias.

1 Historia

La historia de Linux empieza en Finlandia en 1991, cuando un tal Linus B. Torvalds (*Linus.Torvalds@cs.helsinki.fi*), estudiante de la universidad de Helsinki, compra un PC con un procesador 386 para estudiar su funcionamiento. MS/DOS no aprovechaba las características del procesador 386 (por *ejemplo* el modo llamado protegido), por lo que Linus utilizó otro sistema operativo comercializado: *Minix*, desarrollado principalmente por Andrew Tanenbaum. El sistema *Minix* era un pequeño sistema Unix.

Sin embargo, debido a las limitaciones de este sistema, Linus empezó a reescribir algunas partes, a añadir funcionalidades, etc. Posteriormente, difundió el código fuente de su trabajo por Internet, de manera gratuita, con el nombre de Linux, contracción de Linus y Unix. ¡Había nacido Linux! La primera difusión de Linux tuvo lugar el mes de agosto de 1991. Se trataba de la versión 0.01.

Esta primera versión era lo que se podría denominar un embrión. Ni siquiera hubo anuncio oficial. La primera versión «oficial» se hizo pública el 5 de octubre de 1991 (versión 0.02). Esta versión permitía el funcionamiento de algunos programas GNU como *bash*, *gcc*,... pero poca cosa más.

Estas primeras versiones eran muy limitadas (¡Linux 0.01 sólo podía funcionar bajo Minix!). Sin embargo, el hecho que el código fuente se difundiera permitió una rápida evolución del sistema. Con el paso de los años, el número de desarrolladores no ha dejado de aumentar. Al principio, sólo algunos apasionados se enteraron de la aparición de este sistema y se interesaron en él. Actualmente, Linux lo desarrollan decenas de personas situadas en todos los rincones del mundo, que en su mayor parte no se han visto jamás: unas 80 personas contribuyeron a la versión 1.0, y hay más de 190 desarrolladores referenciados para la versión 2.0.

El papel de la red mundial Internet es importante porque ha permitido una rápida evolución del sistema. Es fácil imaginar a un desarrollador instalando Linux en su máquina, encontrando un error, corrigiéndolo y enviando el archivo fuente corregido a Linus. Unos días más tarde (en ocasiones tan sólo unos minutos), el núcleo corregido puede difundirse de nuevo.

Si las primeras versiones de Linux eran relativamente inestables (¡cuántas reinstalaciones había que llevar a cabo!), la primera versión considerada estable (1.0) se hizo pública alrededor de marzo de 1994. El número de versión asociado al núcleo tiene un sentido muy particular porque está relacionado con su desarrollo, ya que la evolución de Linux se efectúa en una sucesión de dos fases:

- Una fase de desarrollo: el núcleo cuya estabilidad no está asegurada y cuyo objetivo es añadirle funcionalidades, optimizaciones y nuevos conceptos. Esta fase se caracteriza por un número de versión impar: 1.1, 1.3. En este momento es cuando se efectúa la mayor parte del trabajo sobre el núcleo.
- Una fase de estabilización cuyo objetivo es obtener el núcleo más estable posible. En este caso, las únicas modificaciones efectuadas en el núcleo son generalmente correcciones y algunas mejoras menores. Los números de versión de estos núcleos llamados estables son pares como 1.0, 1.2 y, más recientemente, 2.0.

Actualmente, Linux es un sistema Unix completo, estable, que sigue evolucionando. Además de gestionar los últimos dispositivos del mercado (memorias flash, discos ópticos, etc.), su rendimiento es comparable al de ciertos sistemas Unix comerciales, y es incluso superior en algunos puntos. Finalmente, aunque Linux ha permanecido mucho tiempo encerrado en el mundo universitario (a menudo debido al acceso a Internet, con el que sólo contaban los universitarios), actualmente empieza a implantarse en el mundo de la empresa. Su gratuidad, su potencia y especialmente su flexibilidad seducen a un número creciente de empresas.

En este momento, la última versión estable es la versión 2.0, y es la versión analizada en este libro.

2 Descripción de Linux y de sus funcionalidades

Linux se diseñó inicialmente como un clónico de Unix distribuido libremente que funcionaba en máquinas PC con procesador 386, 486 o superior. Aunque se desarrolló inicialmente para la arquitectura x86, en la actualidad funciona sobre otras plataformas como los procesadores Alpha, Sparc, ciertas plataformas basadas en los 68000 como Amiga y Atari, las máquinas de tipo MIPS y sobre PowerPC.

Linux es una implementación de Unix que respeta las especificaciones POSIX pero que posee también ciertas extensiones propias de las versiones System V y BSD de Unix. Esto simplifica la adaptación del código de aplicaciones desarrolladas inicialmente para otros sistemas Unix.

El término POSIX significa Portable Operating System Interface. Se trata de documentos producidos por el IEEE y estandarizados por el ANSI y el ISO. El objetivo de POSIX es permitir tener un código fuente transportable. La versión Linux 2.0 respeta la norma POSIX.1 y algunas llamadas definidas por POSIX.4 (véase [Lewine 1991] y [Gallmeister 1995]).

El de Linux es un código original, que no es propietario en absoluto y cuyos programas en código fuente se distribuyen libremente bajo cobertura de la licencia GPL, que es la licencia pública general GNU.

Las funcionalidades de este sistema operativo son múltiples y corresponden a la idea que puede hacerse de un sistema Unix moderno:

- multitarea, multiprocesador: puede ejecutar programas al mismo tiempo, ya sea con uno o varios procesadores;
- multiplataforma: véase más arriba;
- multiusuario: como en todo sistema Unix, Linux permite trabajar a varios usuarios al mismo tiempo en la misma máquina;
- soporte de comunicaciones interprocesos (*pipes*, *IPC*, *sockets*);
- gestión de las diferentes señales;
- gestión de terminales según la norma POSIX. Linux proporciona también los pseudoterminales y los controles de procesos;

- soporte de un gran número de dispositivos ampliamente extendidos (tarjetas de sonido, gráficas, de red, SCSI...);
- *búfer caché*: zona de memoria intermedia para las entradas/salidas de los diferentes procesos;
- gestión de memoria a la carta: una página sólo se carga si es necesaria en memoria;
- bibliotecas compartidas y dinámicas: las bibliotecas dinámicas sólo se cargan cuando son necesarias y su código se comparte si varias aplicaciones las utilizan;
- sistemas de archivos que permiten gestionar tanto particiones Linux con el sistema de archivos Ext2, por ejemplo, como particiones en otros formatos (MS-DOS, iso9660...);
- soporte de la capa TCP/IP y de otros protocolos de red.

Linux es ante todo un sistema Unix rápido y completo, que puede instalarse fácilmente. Además, su difusión entre el gran público le permite evolucionar rápidamente. Finalmente, algunas distribuciones permiten la fácil instalación de este sistema, por lo que le han permitido adquirir una cierta popularidad.

3 ¡Gracias, Internet!

Si Linux es hoy tan popular se debe a que sus fuentes se han difundido por la red mundial Internet de manera libre y gratuita. Esta red ha borrado las distancias entre los desarrolladores. Los servicios de correo electrónico han permitido a los desarrolladores dialogar fácilmente y, ante todo, rápidamente. Las sedes ftp también han facilitado la difusión rápida de los fuentes y las herramientas de desarrollo.

En estos momentos, la comunicación entre los desarrolladores se efectúa por medio de foros Usenet, pero especialmente por correo electrónico. Así, existen muchas listas de difusión de correo sobre temas variados respecto a ciertos ámbitos del desarrollo de Linux.

Existen varias listas de desarrolladores que están abiertas a todos. También es posible discutir problemas encontrados, así como aportar posibles extensiones, añadir nuevas funcionalidades al núcleo, y así sucesivamente. Se proporcionan algunas direcciones electrónicas en la próxima sección.

4 Referencias

Presentamos una lista no exhaustiva de referencias y enlaces que el usuario y el desarrollador pueden consultar libremente: es rica en enseñanzas respecto a numerosos documentos.

Libros de referencia

Se da una amplia bibliografía al fin de esta obra, con la gran mayoría de las obras de referencia existentes. Sin embargo, es necesario citar tres obras:

- la obra [Beck et al. 1996] en inglés que detalla el funcionamiento interno del núcleo...pero de la versión 1.2 de Linux;
- el libro [Welsh and Kaufman 1995] que es la referencia en materia de iniciación y administración bajo Linux;
- finalmente, citemos [Kirch 1995] que es el libro de referencia respecto a la configuración e instalación de una red bajo Unix.

Foros Usenet

En lengua inglesa:

- comp.os.linux.advocacy: ventajas de Linux respecto a otros sistemas operativos;
- comp.os.linux.announce: anuncios para la comunidad Linux (grupo moderado);
- comp.os.linux.answers: difusión de ciertos documentos como las FAQ (Frequently Asked Questions), HowTo's,... (grupo moderado);
- comp.os.linux.development.apps: escritura y portabilidad de aplicaciones bajo Linux;
- comp.os.linux.networking: Linux y la red;
- comp.os.linux.setup: instalación y administración del sistema;
- comp.os.linux.x: X Windows bajo Linux;
- comp.os.linux.misc: todo lo que no se trata en los otros grupos.

Listas de correo electrónico

Existen numerosas listas de difusión respecto a Linux. Su multiplicación impide establecer una lista completa, por lo que presentamos las principales:

- `linux-kernel@vger.rutgers.edu`: destinada a los desarrolladores;
- `linux-scsi@vger.rutgers.edu`: dedicada a los problemas y al desarrollo de controladores SCSI;
- `linux-gcc@vger.rutgers.edu`: para todo lo que respecta a la biblioteca estándar C y gcc;
- `linux-security@tarsier.cv.nrao.edu`: lista que trata de problemas de seguridad;
- `linux-alert@tarsier.cv.nrao.edu`: lista de difusión de mensajes de alerta.

Sedes FTP

Todas las herramientas necesarias para el correcto funcionamiento de Linux se agrupan en dos sedes principales:

- `sunsite.unc.edu:/pub/Linux`;
- `tsx-11.mit.edu:/pub/linux`.

Las actualizaciones de los núcleos se efectúan en las sedes siguientes:

- `ftp.cs.helsinki.fi:/pub/Software/Linux/Kernel`;
- `ftp.funet.fi:/pub/Linux/PEOPLE/Linus`.

Sin embargo, existen numerosos espejos de estas sedes por todas partes, y es interesante buscar la más cercana al lector.

Sedes Web

A medida que Linux evoluciona, el número de sedes Web dedicadas a él no cesa de aumentar. Presentamos aquí las dos sedes principales a partir de las que son accesibles numerosas informaciones y enlaces a otras páginas:

- <http://www.linux.org>;
- <http://www.cs.Helsinki.FI/linux/>.

Las dos direcciones siguientes se encuentran en inglés y francés:

- <http://www.freenix.fr/linux>;
- <http://www.loria.fr/linux>.

Distribuciones

Linux es el núcleo del sistema operativo. La instalación en una máquina (con la recompilación de todas las herramientas) no es cosa realmente fácil. Por esta razón, se difunden un cierto número de distribuciones «llaves en mano». Se encuentran en los principales servidores ftp, y se difunden también ampliamente en soportes como el CD-ROM a precios que desafían toda competencia. Para referencia, aquí tiene algunas de las distribuciones comúnmente difundidas:

- Red Hat: <ftp://ftp.redhat.com/pub/redhat>;
- Slackware: <ftp.cdrom.com:/pub/Linux/slackware>;
- Débian: <ftp.debian.org:/debian>;
- Jurix: <susix.jura.uni-sb.de:/pub/linux>.

Estas distribuciones se agrupan en las sedes centrales Linux; lo mismo puede decirse de los numerosos espejos de los que hemos hablado anteriormente.

Capítulo 2

Presentación general

Este capítulo se dedica a presentar las características de un sistema de tipo Unix en general y de Linux en particular. No nos cansaremos de repetir que Linux es ante todo un sistema Unix, y por tanto hereda todas las características de este tipo de sistema operativo.

No cabe duda que, aunque se sabe bien el papel que juega una aplicación, sigue flotando una nebulosa sobre el término *sistema operativo*. Efectivamente, se emplea (demasiado) a menudo inadecuadamente, y sin embargo ocupa un lugar esencial en cualquier sistema. Este capítulo presenta la función que debe cumplir un sistema operativo así como su estructura y su funcionamiento. Posteriormente, entrando en mayor profundidad, se detallarán las nociones de modo núcleo y usuario.

1 Los diferentes tipos de sistemas operativos

El sistema operativo es un término genérico que designa, en realidad, varias «familias» de sistemas:

- **Sistemas monotarea:** sólo puede ejecutarse un programa a la vez. Por tanto, sólo una persona puede trabajar a la vez en la máquina. Sin embargo, puede aprovechar todos los recursos y la potencia de la máquina.
- **Sistemas multitarea:** pueden ejecutarse varios procesos en paralelo. El tiempo se divide en pequeñas unidades y cada uno de los procesos se ejecuta durante ese intervalo, para que los procesos no se vean perjudicados, se implementa un complejo algoritmo de ordenamiento y prioridad de ejecución para asegurar un cierto reparto entre ellos. Estos sistemas pueden ser multiusuario o no, y multiprocesador.

Las primeras versiones de Unix para el gran público eran sistemas multitarea y multiusuario. Pero actualmente, gracias al progreso de la tecnología y la informática, numerosos sistemas Unix son capaces además de utilizar las máquinas multiprocesador.

Las primeras versiones de Linux sólo funcionaban sobre PC monoprocesador, pero la versión 2.0 ya permite gestionar sistemas multiprocesador.

2 Función del sistema operativo

El sistema Unix se ha desarrollado en un lenguaje de alto nivel (lenguaje C), a diferencia de otros sistemas que se han desarrollado en ocasiones en lenguaje ensamblador. La utilización de un lenguaje de alto nivel ha permitido la portabilidad del sistema a muchas máquinas diferentes. Lo mismo ocurre con Linux. Por ello, era primordial que el código de las aplicaciones pudiera compilarse de manera transparente (o casi), sea cual sea la máquina y los dispositivos utilizados.

De hecho, el transporte de Linux a otra máquina se resume en adaptar la parte del código que es específica de la máquina. Los módulos independientes de la arquitectura pueden reutilizarse tal cual.

2.1 Máquina virtual

Un sistema operativo ofrece una máquina virtual al usuario y a los programas que ejecuta. Se ejecuta en una máquina física que posee una interfaz de programación de bajo nivel, y proporciona abstracciones de alto nivel y una nueva interfaz de programación y uso más evolucionada.

En los primeros sistemas operativos que existían en los años cincuenta, el programador debía conocer la interfaz física de la máquina y programarla directamente. Un sistema moderno ofrece una interfaz de más alto nivel y traduce las peticiones de alto nivel, efectuadas por el usuario, en peticiones físicas de bajo nivel.

El sistema operativo es pues una interfaz entre las aplicaciones y la máquina, como se representa en la figura 2.1.

Por ello es por lo que todas las tareas físicas (acceso a dispositivos externos o internos, a la memoria...) se delegan en el sistema operativo. Esta encapsulación del hardware (y

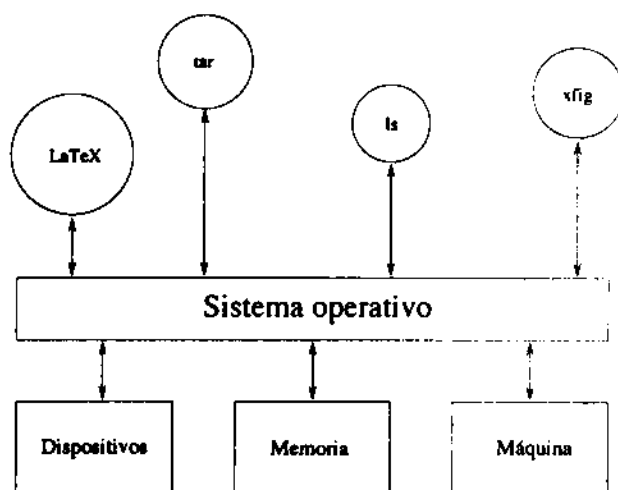


FIG. 2.1 – *Entre las aplicaciones y la máquina*

de su diversidad) libera a los desarrolladores de la complejidad de gestionar todos los dispositivos existentes: el sistema es quien se encarga de ello.

Esto evita también que el usuario se limite a una máquina. Si el sistema está disponible sobre varias arquitecturas, la interfaz de usuario y de programación será la misma en todas.

Cuando un desarrollador desea, por ejemplo, leer el contenido de un archivo, efectúa la misma operación sin importar si el archivo está en una cinta, en disco, en CD-ROM o en otro soporte. El código del programador es el mismo, pero el núcleo efectúa operaciones diferentes en función del dispositivo sobre el que se encuentre el archivo. Es más simple efectuar una apertura de archivo y lecturas de datos que leer físicamente bytes proporcionando una dirección física formada por números de disco, cilindro, cabeza de lectura/escritura y sector.

2.2 Compartir el procesador

Una de las características principales del sistema Unix es que es multitarea, es decir, que varios programas (*procesos*) pueden ejecutarse al mismo tiempo. Por ejemplo, el lanzamiento de una impresión, el formateo de un disquete, el envío o la recepción de correo electrónico pueden realizarse al mismo tiempo.

Para ello, el sistema implementa un sistema de ordenamiento, que se presenta en el capítulo 4, sección 1.8, y que tiene por objetivo distribuir el procesador entre los procesos. De hecho, si sólo hay un procesador, se ejecuta un solo programa en un momen-

to dado. Pero al efectuar una rotación muy rápida, el o los usuarios de la máquina tienen la impresión de que todos los procesos se ejecutan de manera paralela.

Esta gestión debe ser muy sofisticada para no perjudicar a nadie, y se trata de uno de los mecanismos clave del sistema.

2.3 Gestión de la memoria

El sistema se encarga de gestionar la memoria física del ordenador. En efecto, en un entorno multiusuario y por tanto multitarea, el sistema debe efectuar una gestión muy rigurosa de la memoria. Como la memoria física de la máquina es a menudo insuficiente, el sistema utiliza entonces una parte del disco duro como memoria virtual (área de intercambio o *swap*).

El sistema operativo debe ser capaz de gestionar hábilmente la memoria para poder satisfacer las peticiones de los distintos procesos tan deprisa como sea posible. Además, debe asegurar la protección de las zonas de memoria asignadas a los procesos para evitar las modificaciones no autorizadas.

La gestión de la memoria se expone en el capítulo 8.

2.4 Gestión de recursos

De manera general, el sistema operativo se encarga de gestionar los recursos disponibles (entre los cuales el procesador y la memoria son casos particulares).

El sistema ofrece a los procesos una interfaz que permite compartir los recursos (discos, impresoras, etc.) de la máquina física. Implementa un sistema de protección que permite a los usuarios y a los administradores proteger el acceso a sus datos.

El sistema mantiene listas de recursos disponibles y en curso de utilización, lo que le permite atribuirlos a los procesos que los necesiten. En todo momento conoce los procesos que utilizan los recursos de la máquina, y puede detectar así los conflictos de acceso.

2.5 Centro de comunicación de la máquina

Una de las tareas del sistema operativo es gestionar los diferentes eventos provenientes del hardware (interrupciones) o de las aplicaciones (llamadas al sistema). Estos

eventos son importantes, y el sistema debe tratarlos y, llegado el caso, enviarlos a los procesos afectados.

Pero si el sistema operativo debe ser capaz de responder a un evento, también debe ser capaz de comunicar varios procesos. De esta manera, los procesos podrán comunicar entre sí, intercambiar informaciones, sincronizarse, etc. Unix pone a disposición del desarrollador varios mecanismos que van de las señales a los IPC, pasando por las tuberías (*pipes*) y los *sockets* (que no se tratan en este libro).

Es por ello por lo que el núcleo es el centro de comunicación de la máquina. Cuando dos procesos intercambian datos, es porque el sistema operativo implementa mecanismos sofisticados junto con recursos específicos.

3 Estructura general del sistema

El sistema operativo tal como se ha presentado se compone de varios elementos. Unix y Linux en particular son sistemas bastante importantes, su desarrollo se ha realizado de manera modular de modo que se puedan distinguir fácilmente las diferentes partes que los componen.

La ventaja de esta estructuración, aparte del hecho que puede estudiarse fácilmente, es que permite su modificación y mejora sin excesiva dificultad. La adición de ciertos elementos, como las llamadas al sistema, controladores de dispositivos u otros, es bastante simple y no obliga a rediseñar la estructura del sistema. Por esta razón, Unix, aunque se diseñó a fines de los sesenta, ha evolucionado y sigue en activo en nuestros días.

Los distintos elementos que encontramos en el núcleo del sistema Unix son los siguientes:

- llamadas al sistema: implementación de operaciones que deben ejecutarse en modo núcleo;
- sistemas de archivos: entradas/salidas de los dispositivos;
- *búfer caché*: memoria intermedia sofisticada para entradas/salidas;
- controladores de dispositivos: gestión de bajo nivel de discos, tarjetas, impresoras...;
- gestión de la red: protocolos de comunicación de red;
- interfaz con la máquina: código (generalmente en ensamblador) de acceso de bajo nivel al hardware;

- centro del núcleo:
 - gestión de procesos (creación, duplicación, destrucción...);
 - gestor de órdenes;
 - señales;
 - módulos cargables (carga de ciertas partes del núcleo cuando se requieren);
 - gestión de memoria (gestión de la memoria física y de la memoria virtual).

La figura 2.2 representa la estructura del sistema operativo Unix así como la interacción entre las diferentes partes que lo componen. Este esquema ha sido adaptado ligeramente para Linux y difiere poco respecto a la estructura tradicional que se puede encontrar en [Bach 1993].

Este esquema evidencia la frontera que existe entre la máquina y las aplicaciones. Bajo Unix, no se trata de acceder directamente a la memoria física de la máquina, a la tarjeta de vídeo ni a cualquier otro dispositivo físico. Además, la separación del núcleo en entidades distintas facilita su depuración y su desarrollo.

Finalmente, para acceder a las funcionalidades ofrecidas por el sistema operativo, el sistema Unix pone a disposición de los procesos una interfaz de comunicación llamada «llamadas al sistema».

4 Modo núcleo, modo usuario

4.1 Principio

Un proceso en un sistema Unix posee dos niveles de ejecución: *núcleo* y *usuario*.

El modo núcleo constituye un modo privilegiado: en este modo, no se impone ninguna restricción al núcleo del sistema. Este último puede utilizar todas las instrucciones del procesador, manipular toda la memoria y dialogar directamente con todos los controladores de dispositivos.

El modo usuario es el modo de ejecución normal de un proceso. En este modo, el proceso no posee ningún privilegio: ciertas instrucciones están prohibidas, sólo tiene acceso a las zonas que se le han asignado, y no puede interactuar con la máquina física. Un

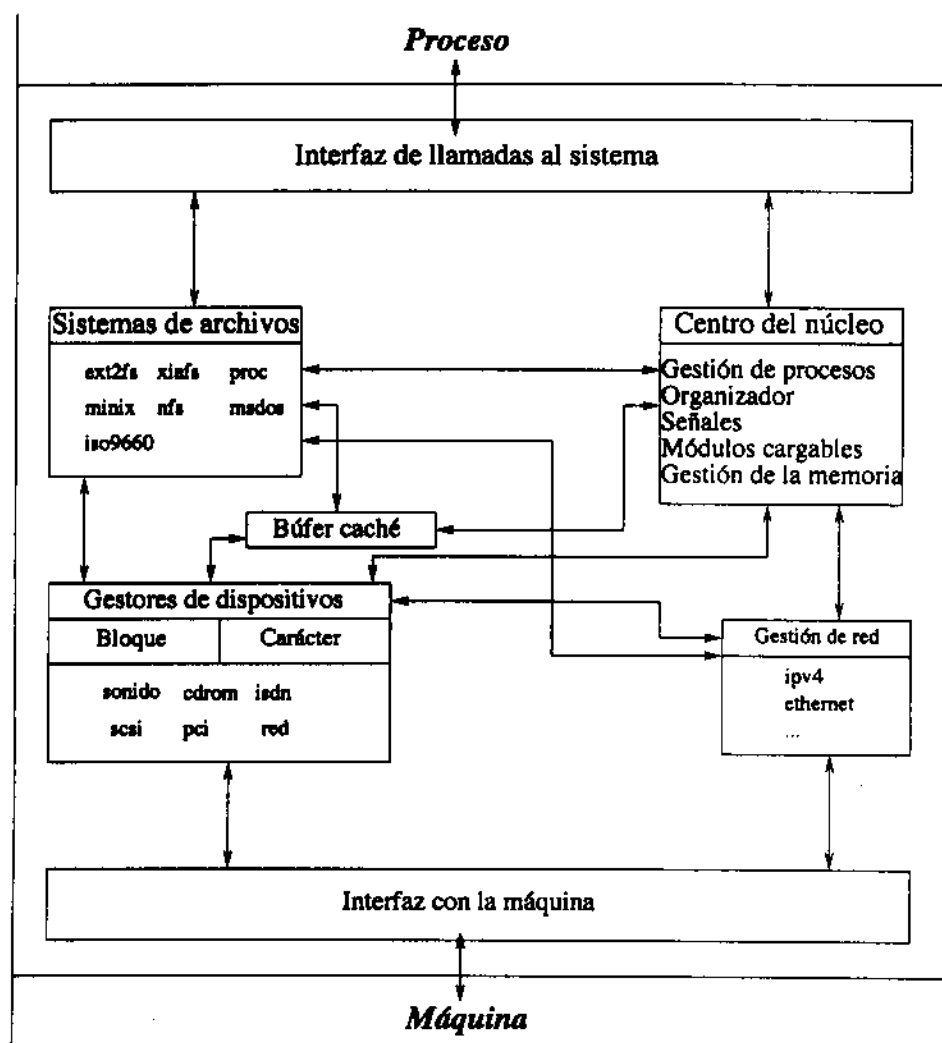


FIG. 2.2 – Estructura del sistema

proceso en modo usuario efectúa operaciones en su entorno, sin interferir con los demás procesos, y puede ser interrumpido en cualquier momento. Esto no obstaculiza su funcionamiento.

4.2 Llamadas al sistema

Un proceso que se ejecute en modo usuario no puede acceder directamente a los recursos de la máquina. Para ello, debe efectuar *llamadas al sistema*.

Una llamada al sistema es una petición transmitida por un proceso al núcleo. Este último trata la petición en modo núcleo, con todos los privilegios, y envía el resultado al proceso, que prosigue su ejecución.

Bajo Unix, la llamada al sistema provoca un cambio: el proceso ejecuta una instrucción del procesador que le hace pasar al modo núcleo. Seguidamente ejecuta una función de tratamiento vinculada a la llamada al sistema que ha efectuado, y luego vuelve al modo usuario para proseguir su ejecución. De este modo, el propio proceso trata su llamada al sistema, ejecutando una función del núcleo. Esta función se supone que es fiable y puede ejecutarse pues en modo privilegiado, contrariamente al programa ejecutado por el proceso en modo usuario.

Capítulo 3

Desarrollo bajo Linux

Este capítulo está destinado a presentar las diferentes herramientas de desarrollo, así como las técnicas utilizadas bajo Linux. Linux es el núcleo del sistema operativo, y va acompañado por toda una serie de herramientas destinadas a los desarrolladores. La característica común a todos estos productos es que son gratuitos, disponibles para su descarga en la mayoría de sedes ftp del planeta, como el propio sistema. A pesar de ser de libre distribución, estos productos son generalmente tan rápidos y potentes como los programas de desarrollo del comercio, o incluso más.

En este capítulo se detalla la organización de los programas fuente del núcleo. También se introducen algunas nociones de programación del sistema.

1 Los productos GNU

1.1 Presentación

GNU son las iniciales de Gnu's Not Unix. La palabra inglesa gnu designa el ñu (numerosos programas llevan nombres de animales: yacc, bison...), una especie de búfalo de África del Sur.

Se trata de un vasto proyecto iniciado por Richard Stallman (llamado rms) en 1983. Cuando se anunció este proyecto, RMS era el autor de un tratamiento de texto potente: *emacs*. Tras su entrada en el laboratorio de inteligencia artificial del MIT, decidió el lanzamiento de un vasto proyecto orientado a realizar un sistema operativo cuyo código

go fuente se distribuiría libremente. Sin embargo, el proyecto GNU se orientó rápidamente hacia la realización de todas las herramientas que deben acompañar a un sistema así, desde el compilador hasta el tratamiento de texto.

Pero Linux no es GNU, y no hay que confundirlos: Linux es el sistema operativo, y algunas de estas herramientas de desarrollo provienen del proyecto GNU. Todas estas herramientas se difunden con una licencia particular de uso: la GPL, Gnu Public License, que corresponde a un *CopyLeft* (¡a diferencia de un Copyright!). Esta licencia permite difundir libremente el código fuente, modificarlo, etc.

Es necesario precisar que el sistema operativo deseado por RMS no es Linux. Este sistema se está desarrollando y lleva el nombre de **Hurd** (véase <http://www.gnu.ai.mit.edu/> para más información).

1.2 Herramientas GNU

1.2.1 Presentación

La cantidad de herramientas GNU es muy importante: abarca desde compiladores (*gcc*, *gnat* ...) a tratamiento de textos de propósito general (*emacs*) pasando por un editor de curvas (*gnuplot*) y juegos (*nuchgess* ...). Se pueden encontrar productos GNU que cubren la mayor parte de los ámbitos, tanto para desarrolladores como para el usuario final. Estas herramientas son generalmente de buena calidad y bastante potentes. Se incluyen en la mayoría de distribuciones Linux y se difunden a menudo con el código fuente.

Estas herramientas se difunden ampliamente en las distintas sedes ftp (consulte la sede principal de las herramientas GNU: [prep.ai.mit.edu](ftp://prep.ai.mit.edu)). Gracias a la difusión del código fuente, estas herramientas se corrigen, se mejoran y optimizan con las diferentes actualizaciones. Para estar al corriente de estas actualizaciones, basta con abonarse al foro Usenet gnu.announce que difunde estos anuncios.

Una herramienta en particular ha marcado a Linux: el compilador *gcc*.

1.2.2 gcc

Como todo sistema operativo de tipo Unix, Linux está implementado en lenguaje C, y ha sido necesario desde el principio de su desarrollo disponer de un compilador C. El programa *gcc*, de *GNU C Compiler*, fue el primer compilador C que permitió la gene-

ración de código para Linux. *gcc* se considera como uno de los mejores productos GNU existentes. Hay que observar que existe un documento particular ([Barlow 1996]) que proporciona ciertos detalles respecto a Linux y *gcc*.

El compilador *gcc* se considera generalmente como uno de los compiladores más potentes. Además de ser eficaz en términos de rapidez, de código generado y de optimización, soporta todos los estándares de programación en C, tanto el ANSI como la forma llamada *K&R* (Kernighan y Ritchie), u otras extensiones del lenguaje C. Para más detalles respecto a las formas del lenguaje C, consulte [Kernighan and Ritchie 1992].

gcc es también un compilador de C++, que gestiona prácticamente todas las características de la última versión del lenguaje (3.0) definida por Margaret Ellis y Bjarne Stroustrup en [Ellis and Stroustrup 1990]. Gestiona por ejemplo las *plantillas* (*templates*), las excepciones, la herencia múltiple...

Finalmente, *gcc* permite compilar también un lenguaje bastante particular: *Objective C*. Se trata de un lenguaje orientado a objetos que no ha gozado de la misma difusión que C++ (fue el lenguaje elegido para el sistema operativo NextStep).

2 Las herramientas del desarrollador

Para un desarrollador, el compilador, el enlazador y el depurador son vitales. El compilador le indica si hay errores de sintaxis en su código y el enlazador resuelve los diferentes símbolos.

Sin embargo, no basta con que un programa pase con éxito la etapa de la compilación para que funcione. Aquí es donde entra en juego el depurador, que permitirá ejecutar paso a paso la aplicación deseada.

2.1 Las diferentes fases de la compilación

La fase de compilación que empieza con un código en C y produce un código ejecutable es de hecho el resultado de un cierto número de acciones. El objetivo de este libro no es detallar con precisión lo que ocurre desde el punto de vista del análisis sintáctico, léxico o semántico. Algunas obras especializadas en este ámbito como [Aho et al. 1991] detallan todo el funcionamiento de un compilador. Por esta razón, sólo se presenta el funcionamiento de las herramientas a disposición del desarrollador.

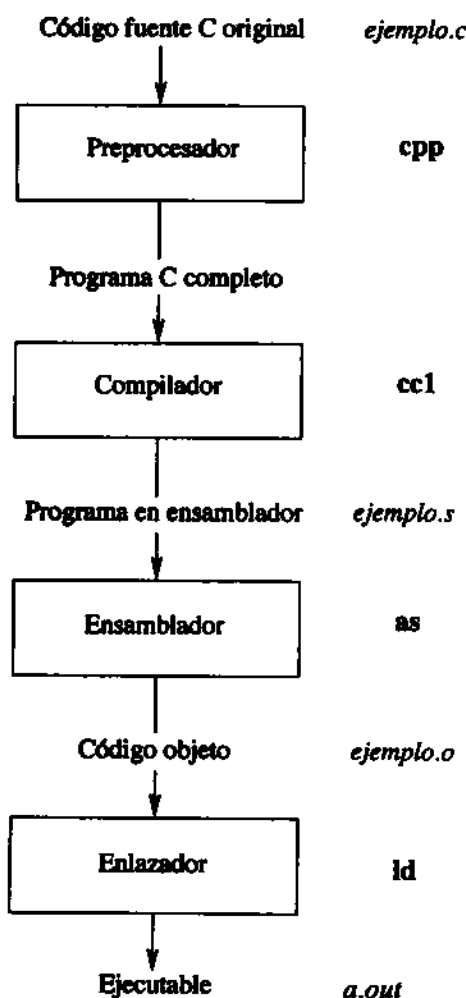


FIG. 3.1 – Desarrollo de una compilación

El desarrollo de una compilación, como se puede constatar en la figura 3.1, está constituido por una multitud de etapas:

- El código C se envía a través de un preprocesador que efectúa la inclusión de archivos de cabecera así como el reemplazo de las diferentes macroinstrucciones definidas. El programa *cpp* es quien se encarga de esta operación.
- Se efectúa la compilación efectiva del código para generar un código ensamblador. El compilador no sólo convierte el código, sino que realiza un buen número de optimizaciones. Esta operación la realiza el programa *cc1*.

- La operación de ensamblado efectuada por el programa *as* consiste en generar el archivo objeto asociado.
- Finalmente, la última operación consiste en generar el ejecutable propiamente dicho del programa. Se trata de reunir todos los archivos objeto y efectuar el enlazado. Esta operación compleja la realiza el programa *ld*.

El apéndice A presenta las diferentes etapas de la compilación de un programa en C.

2.2 gcc

El mandato *gcc* permite encadenar las diferentes fases de la compilación de un programa. Se trata de un lanzador de programas que ejecuta las diferentes etapas (preprocesador, compilación, ensamblado y enlazado) transmitiendo a los programas las opciones proporcionadas por el programador.

La opción *-v* permite visualizar los diferentes programas lanzados por *gcc*:

```
randalf# gcc -v exemple.c
Reading specs from /usr/lib/gcc-lib-i486-linux/2.7.2/specs
gcc version 2.7.2

/usr/lib/gcc-lib/i486-linux/2.7.2/cpp -lang-c -v -undef -D__GNUC__=2
-D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux -D__ELF__ -D__unix__
-D_i386__ -D__linux__ -D__unix__ -D_i386__ -D__linux__ -Asystem(unix)
-Asystem(posix) -Acpu(i386) -Amachine(i386) -D_i486__ exemple.c
/tmp/cca01212.i
GNU CPP version 2.7.2 (i386 Linux/ELF)
#include "... " search starts here:
#include <...> search starts here:
/usr/local/include
/usr/i486-linux/include
/usr/lib/gcc-lib/i486-linux/2.7.2/include
/usr/include
Arch of search list.

/usr/lib/gcc-lib/i486-linux/2.7.2/cc1 /tmp/cca01212.i -quiet -dumpbase
exemple.c -version -o /tmp/cca01212.s
GNU C version 2.7.2 (i386 Linux/ELF) compiled by GNU C version 2.7.2

/usr/i486-linux/bin/as -V -Qy -o /tmp/cca01212.o /tmp/cca01212.s
GNU assembler version 960228 (i486-linux), using BFD version 2.6.0.12

/usr/i486-linux/bin/ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.1
/usr/lib/crti.o /usr/lib/crti.o /usr/lib/crtbegin.o
-L/usr/lib/gcc-lib/i486-linux/2.7.2 -L/usr/i486-linux/lib /tmp/cca01212.o
-lgcc -lc -lgcc /usr/lib/crtend.o /usr/lib/crtn.o
```

Además de las diferentes opciones definidas de modo predeterminado (como `__linux__`), se puede observar que se indican todos los números de versión de las diferentes herramientas, lo cual es muy práctico cuando el administrador desea actualizar el entorno de la máquina.

Como indica la primera línea que sigue a la llamada a `gcc -v`, el compilador recupera las opciones predeterminadas en el archivo `/usr/lib/gcc-lib/i486-linux/2.7.2/specs`. El camino de acceso es variable según la versión del compilador y de la instalación efectuada. En algunos casos, puede ser interesante modificar ciertas opciones o integrar otras, a fin por ejemplo de especializar el compilador para su procesador.

Veamos un pequeño resumen de las principales opciones que pueden pasarse directamente a `gcc`:

- `-M`: parada tras la etapa del preprocesador;
- `-S`: parada tras la generación del código ensamblador;
- `-c`: parada tras la generación del código objeto;
- `-g`: inserción de los símbolos para la depuración;
- `-onondest`: nombre del archivo generado;
- `-DMACRO`: define la macroinstrucción;
- `-DMACRO=valor`: define la macroinstrucción y le da un valor;
- `-IDIRECTORIO`: directorio donde `cpp` debe ir a buscar los archivos de cabecera;
- `-LDIRECTORIO`: directorio donde `ld` debe ir a buscar las bibliotecas;
- `-lbiblioteca`: incluye la biblioteca para el enlazado;
- `-pipe`: encadenamiento de las diferentes opciones de compilación sin utilizar archivos temporales. Esta opción acelera la compilación pero requiere más memoria.
- `On`: nivel de optimización ($0 \rightarrow 3$).

El número de opciones de `gcc` es enorme. Consulte la página de manual, o bien [Stallman 1995] para más detalles.


```
fstat(1, (dev 0 0 ino 35430 mode 010600 nlink 1 uid 501 gid 100
size 0 ...)) = 0
mmap(0, 4096, READ|WRITE, PRIVATE, 4294967295, 0) = 0x40007000
write(1, "Cha\enee initiale : exemple \n Cha"... , 68 Cadena inicial : ejemplo
Cadena duplicada : ejemplo -> ejemplo
) = 68
exit(1074283076) = ?
```

En el ejemplo anterior, el programa empieza por cargarse y seguidamente busca la biblioteca dinámica C. Tras un fracaso (la biblioteca no se encuentra en */usr/lib*), se carga la *libc* que se encuentra en */lib*.

Las diferentes llamadas a *mmap* y a *mprotect* (*SYS_125*) realizan las cargas del programa y de la biblioteca. Las dos llamadas *brk* corresponden a dos asignaciones de memoria efectuadas en el programa.

Finalmente, las llamadas a *fstat* y *write* corresponden a la escritura en la salida estándar del resultado de la operación. El programa termina por una llamada a *exit*.

2.5 make

2.5.1 Presentación

La herramienta *make* facilita la compilación de proyectos. Es una herramienta estándar instalada en todo sistema Unix. Este programa permite efectuar una compilación inteligente de programas, en función de los archivos modificados que necesitan realmente ser recompilados.

Sin embargo, la sintaxis de los archivos *Makefile* es relativamente compleja de escribir porque utiliza reglas de reescritura. Una muestra de las capacidades de esta herramienta se presenta en el apéndice C. Para más detalles, se aconseja referirse a la página de manual de *make*, o a la documentación de la versión GNU [Stallman and McGrath 1995].

3 Formato de los ejecutables

Linux soporta dos formatos de programas ejecutables. El primer formato binario utilizado por Linux fue el formato llamado *a.out*. Sin embargo, este formato era de uso poco práctico para implementar bibliotecas dinámicas, y ahora el formato binario utilizado es ELF.

3.1 a.out: el ancestro

El formato a.out (Assembler.OUTPUT) es el formato de las bibliotecas y ejecutables utilizado en las primeras versiones de Unix en general y de Linux en particular.

Existen diversas variantes de este formato (ZMAGIC, QMAGIC ...). QMAGIC es el formato de los ejecutables que se parece algo a los antiguos binarios a.out (también conocidos como ZMAGIC), pero deja la primera página del programa libre. Ello permite recuperar más fácilmente las direcciones no asignadas (como NULL) en el intervalo 0-4096.

Los enlazadores anticuados sólo gestionan el formato ZMAGIC, mientras que los más recientes gestionan los dos. Los dos tipos son, de todos modos, soportados por el núcleo.

Para saber si un programa es de formato a.out, basta con utilizar el mandato *file*:

```
gandalf# file /usr/local/bin/kermi
t
/usr/local/bin/kermi: Linux/i386 demand-paged executable (ZMAGIC)
```

Si la cadena *Linux/i386* aparece, el binario especificado está en formato a.out. Asimismo, se precisa que se trata de un binario en formato ZMAGIC.

El formato de los binarios de tipo a.out se implementa en el archivo *fs/binfmt_aout.c*. Sin embargo, la gestión de este formato sólo se conserva por razones históricas. Por ello no detallaremos su funcionamiento.

3.2 ELF

3.2.1 Bienvenida al mundo ELF

El nuevo formato de binarios para Linux es el fomato ELF (que no tiene nada que ver con el mundo de Tolkien y sus anillos, aunque a primera vista algunas características parezcan extrañas, enigmáticas o mágicas). ELF es la abreviatura de Executable and Linking Format. Este formato fue inicialmente diseñado y desarrollado por el USL (Unix System Laboratories), y es utilizado por los sistemas de tipo SVR4 y Solaris 2.x.

Dado que ELF es mucho más flexible y manejable que el formato a.out, los desarrolladores de Linux decidieron migrar hacia este formato en 1994. La técnica de migración se detalla en [Barlow 1995] y en [Dumas 1996].

3.2.2 ELF y su formato

ELF se ha convertido en un formato extendido y muy utilizado (en varios sistemas Unix) porque es de manipulación simple. Esta facilidad proviene de su estructura. El formato ELF se detalla completamente en [TIS 1993].

El formato ELF permite generar tres tipos de archivos:

- archivo relocizable (archivo objeto);
- archivo ejecutable;
- archivo compartido (bibliotecas).

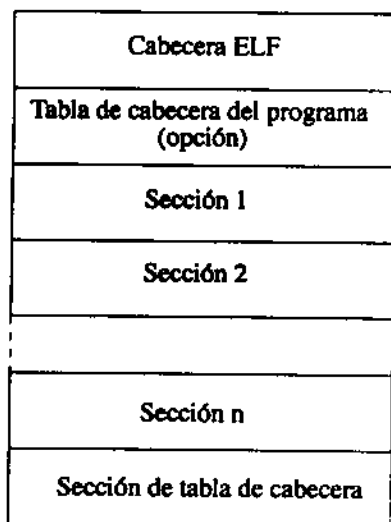


FIG 3.2 – Formato ELF: estructura

En todos los casos, el formato del archivo es el mismo, como lo ilustra la figura 3.2.

La cabecera del archivo ELF constituye una descripción del archivo y describe su organización. Es la única cosa que tiene una posición fija en el formato de un archivo ELF: las demás secciones pueden colocarse en cualquier lugar en el archivo. Pero es posible acceder directamente a estos datos. El archivo de cabecera `<elf.h>` (en realidad, el archivo `<linux/elf.h>` es quien contiene los datos) contiene la declaración de la estructura de cabecera de un archivo ELF. Se trata de la estructura `Elf32_Ehdr`.

<i>tipo</i>	<i>campo</i>	<i>descripción</i>
unsigned char [EI_NIDENT]	e_ident	Número mágico del archivo con ciertas indicaciones
Elf32_Half	e_type	Tipo del archivo
Elf32_Half	e_machine	Arquitectura del archivo
Elf32_Word	e_version	Versión del formato de archivo
Elf32_Addr	e_entry	Dirección virtual del punto de entrada del programa
Elf32_Off	e_phoff	Desplazamiento para encontrar la tabla de cabecera del programa
Elf32_Off	e_shoff	Desplazamiento para encontrar la sección de cabecera del programa
Elf32_Word	e_flags	Opciones particulares para el procesador
Elf32_Half	e_ehsize	Tamaño de la cabecera en bytes
Elf32_Half	e_phentsize	Tamaño de una entrada en la tabla de cabecera del programa
Elf32_Half	e_phnum	Número de entradas en la tabla de entrada del programa
Elf32_Half	e_shentsize	Tamaño de una cabecera de sección
Elf32_half	e_shnum	Número de entradas en la tabla de secciones
Elf32_half	e_shstrndx	Índice de la tabla de secciones

El campo de identificación, constituido por una tabla de bytes, es muy importante. Se puede descomponer de la forma siguiente:

<i>posición</i>	<i>función</i>	<i>significado</i>
EI_MAG0, EI_MAG1 EI_MAG2, EI_MAG3	número mágico	Constituido por 4 bytes: 0x7f E L F. Este número mágico está constituido en realidad por cuatro macrodefiniciones: ELFMAG0 , ELFMAG1 , ELFMAG2 y ELFMAG3 .
EI_CLASS	clase del archivo	ELF es un formato portable, por lo que las direcciones del archivo pueden estar en forma de 32 o 64 bits. Este byte puede tener los valores ELFCLASSNONE (inválido), ELFCLASS32 o ELFCLASS64 .
EI_DATA	tipo de codificación	Determina si se encuentra en un entorno de mayor o menor peso. Pueden darse los valores ELFDATANONE (inválido), ELFDATA2LSB y ELFDATA2MSB .
EI_VERSION EI_PAD	versión de la cabecera Marca	Debe ser EV_CURRENT . Da el inicio de los bytes no utilizados y asignados para extensiones futuras.

A partir de esta cabecera, es posible acceder a las diferentes secciones del archivo. Su número y posición son completamente variables, en función del compilador que ha generado el código, las opciones especificadas en la generación del archivo objeto, o simplemente del lenguaje en el que se haya escrito el código fuente.

Sin embargo, explorar un archivo binario ELF a mano es una operación que resulta en ocasiones oscura y compleja. Una biblioteca facilita los accesos a las informaciones de un archivo ELF, proporcionando funciones de análisis. Para utilizar estas últimas, hay que incluir el archivo `<libelf.h>` y añadir la opción `-lelf` en el enlazado.

4 Presentación de las bibliotecas

4.1 Utilidad

Las bibliotecas constituyen una manera simple de agrupar varios archivos objeto. Existen dos tipos de bibliotecas:

- estáticas: en el enlazado, el código de la biblioteca se integra al ejecutable;
- dinámicas: el código de la biblioteca se carga en la ejecución del programa.

Las bibliotecas dinámicas permiten una economía de espacio en disco pero más aún de ocupación de memoria porque una biblioteca dinámica se carga una sola vez, y el código puede ser compartido por todas las aplicaciones que la necesiten. Finalmente, cuando una biblioteca dinámica se actualiza, no es necesario recompilar las aplicaciones que la utilizan.

La biblioteca dinámica que se utiliza cotidianamente es la biblioteca en C. Todas las aplicaciones (salvo algunas excepciones) se enlazan dinámicamente con la biblioteca C. El programa `ldd` permite conocer la lista de bibliotecas enlazadas con una aplicación:

```
gandalf# ldd /usr/X386/bin/xv
        libXext.so.6 => /usr/X11/lib/libXext.so.6.0
        libX11.so.6 => /usr/X11/lib/libX11.so.6.0
        libm.so.5 => /lib/libm.so.5.0.5
        libc.so.5 => /lib/libc.so.5.3.9
```

4.2 Utilización de bibliotecas

La utilización de una biblioteca es muy simple: basta con especificar en el enlazado la biblioteca. Por ejemplo:

```
gandalf# gcc -o demo_compleja main.c -L/usr/local/compleja -  
compleja -lm  
gandalf# ldd demo_compleja  
      libm.so.5 => /lib/libm.so.5.0.5  
      libc.so.5 => /lib/libc.so.5.3.9
```

El resultado anterior corresponde al caso en que la biblioteca haya sido generada de forma estática, y no dinámica.

El apéndice D expone varias herramientas de gestión de bibliotecas.

5 Organización del código fuente del núcleo

5.1 Núcleo

Los programas fuente del núcleo Linux se instalan normalmente en el directorio */usr/src/linux*.

Se organizan de manera jerárquica, como se puede ver en la figura 3.3. Cada directorio o subdirectorio se dedica a ciertas funcionalidades del núcleo:

- *Documentación*: un cierto número de archivos de documentación respecto a la configuración del núcleo o el funcionamiento de ciertos módulos;
- *include*: todos los archivos de cabecera necesarios para la generación del núcleo pero también para la compilación de aplicaciones;
- *fs*: sistemas de archivos;
- *init*: el *main.c* de Linux;
- *ipc*: gestión de las comunicaciones interprocesos según la norma System V;
- *mm*: gestión de la memoria;
- *kernel*: principales llamadas al sistema;

- *lib*: módulos diversos;
- *drivers*: controladores de dispositivos;
- *net*: protocolos de red;
- *arch*: código dependiente de la plataforma;
- *scripts*: scripts utilizados para la configuración y para la generación de las dependencias del núcleo.

5.2 Archivos de cabecera

Los archivos de cabecera utilizados por el preprocesador del lenguaje C (como `<stdio.h>` por ejemplo) se sitúan en el directorio `/usr/include`. Definen la interfaz de las bibliotecas utilizadas por los programas que se compilan.

Estos archivos de cabecera se enlazan con la biblioteca C, que se distribuye independientemente del núcleo.

El núcleo Linux dispone también de archivos de cabecera que se utilizan en su compilación. Estos archivos se sitúan en el directorio `/usr/src/linux/include`. Encontramos principalmente dos subdirectorios:

- *linux*: este directorio contiene las declaraciones independientes de la arquitectura;
- *asm*: este directorio contiene las declaraciones dependientes de la arquitectura: se trata de un enlace con otro directorio (*asm-i386* para la arquitectura x86 por ejemplo).

Dos enlaces simbólicos permiten también acceder a estos archivos desde el directorio `/usr/include`. También es posible incluir en un programa archivos de cabecera que definen constantes y tipos utilizados por el núcleo utilizando las notaciones `<asm/file.h>` y `<linux/file.h>`. Hay que observar, sin embargo, que no es aconsejable la inclusión de tales archivos. Es preferible incluir los archivos estándar de la biblioteca C, porque están previstos para ello. Estos archivos de cabecera se encargan de importar las definiciones del núcleo.

En los capítulos siguientes, la descripción de llamadas al sistema de Linux presenta los archivos de cabecera estándar, que se pueden encontrar en todo sistema Unix. Ciertas definiciones de constantes y de tipos se efectúan en los archivos de cabecera del núcleo, pero estos últimos no se detallan en las primeras partes de cada capítulo, por razones de portabilidad.

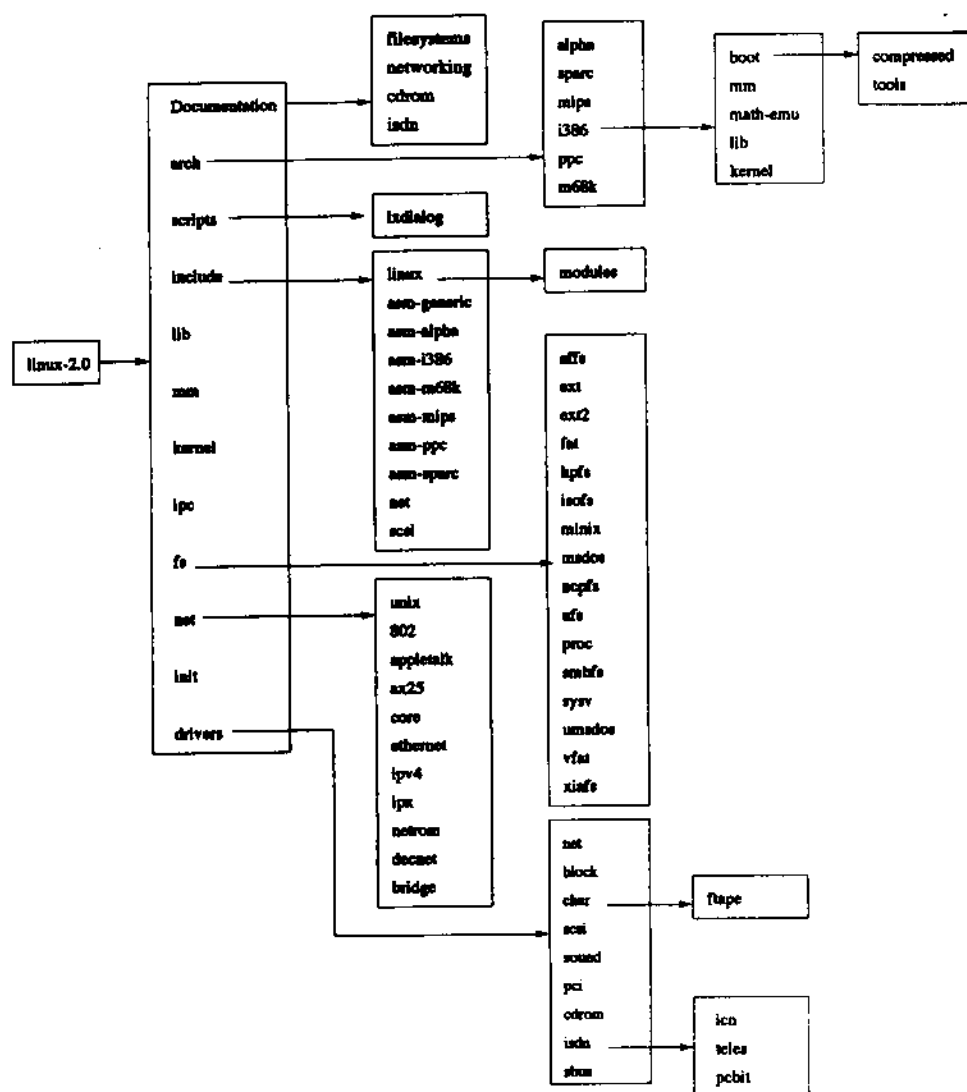


FIG. 3.3 – Árbol de fuentes

Por el contrario, el contenido de los archivos de cabecera del núcleo se describe en la explicación del funcionamiento de las diversas partes del núcleo.

6 Funcionamiento del núcleo

6.1 Llamadas al sistema

Como se explica en el capítulo 2, sección 4.1, un proceso se ejecuta normalmente en modo usuario y debe pasar al modo núcleo para ejecutar las llamadas al sistema.

6.1.1 Implementación de una llamada al sistema

Una llamada al sistema se caracteriza por un nombre y un número único que la identifica. Este número puede encontrarse en el archivo `<asm/unistd.h>`. Por ejemplo, la llamada al sistema *open* tiene el número 5. El nombre se define en el archivo ensamblador *arch/i386/kernel/entry.S* (aunque este archivo se encuentra en un directorio dedicado a la arquitectura x86, existe para las otras arquitecturas), y se llama *sys_open*.

La biblioteca C proporciona funciones que permiten ejecutar una llamada al sistema. Estas funciones afectan al nombre de las llamadas al sistema. Basta, pues, para ejecutar una llamada al sistema con llamar a la función correspondiente.

En los programas fuente del núcleo, una llamada al sistema posee el prototipo siguiente:

```
asmlinkage int sys_nomllamada(lista de argumentos)
{
/* Código de la llamada al sistema */
}
```

El número de argumentos debe estar comprendido entre 0 y 5. La palabra clave *asmlinkage* es una macroinstrucción definida en el archivo `<linux/linkage.h>`:

```
#ifdef __cplusplus
#define asmlinkage extern "C"
#else
#define asmlinkage
#endif
```

Cuando un proceso ejecuta una llamada al sistema, llama a la función correspondiente de la biblioteca C. Esta función trata los parámetros y hace pasar al proceso al modo núcleo.

En la arquitectura x86, este paso al modo núcleo se efectúa de la forma siguiente:

- los parámetros de la llamada al sistema se colocan en ciertos registros del procesador;
- se provoca un bloqueo desencadenando la interrupción lógica 0x80;
- este bloqueo provoca el paso del proceso al modo núcleo; el proceso ejecuta la función `system_call` definida en el archivo fuente `arch/i386/kernel/entry.S`;
- esta función utiliza el número de llamada al sistema (transmitido en el registro `eax`) como índice en la tabla `sys_call_table`, que contiene las direcciones de las llamadas al sistema, y llama a la función del núcleo correspondiente a la llamada al sistema;
- al volver de esta función, `system_call` vuelve a quien ha llamado; este retorno vuelve a pasar al proceso al modo usuario.

6.1.2 Creación de una llamada al sistema

Para comprender bien el funcionamiento de una llamada al sistema, nada mejor que crear una. La llamada al sistema que vamos a crear consiste en tomar tres argumentos, hacer la multiplicación de los dos primeros y colocar el resultado en el tercero.

En un primer momento, es necesario «declarar» la llamada al sistema, y por tanto asignarle un número. La versión 2.0 de Linux cuenta con 164 llamadas al sistema. La llamada al sistema `show_mult` tendrá pues como número el 164 (la primera llamada al sistema, `setup`, tiene por número 0). Para declarar nuestra llamada, hay que añadir en el archivo `include/asm/unistd.h`:

```
/* Llamadas al sistema del núcleo */
#define __NR_sched_get_priority_min 160
#define __NR_sched_rr_get_interval 161
#define __NR_nanosleep 162
#define __NR_mremap 163

/* Llamada al sistema añadida */
#define __NR_show_mult 164
```

Seguidamente, en el archivo `arch/i386/kernel/entry.S` (o bien en el archivo `entry.S` de la arquitectura):

```
/* Llamadas al sistema del núcleo */
.long SYMBOL_NAME(sys_sched_get_priority_min) /* 160 */
```



```
.long SYMBOL_NAME(sys_sched_rr_get_interval)
.long SYMBOL_NAME(sys_nanosleep)
.long SYMBOL_NAME(sys_mremap)

/* Valor inicial del número de llamadas */
/* .space (NR_syscalls-163)*4 */

/* Llamada al sistema añadida */
.long SYMBOL_NAME(sys_show_mult)
.space (NR_syscalls-164)*4
```

En este estado, se efectúan todos los enlaces que permiten la utilización de la llamada al sistema. No queda más que implementarlo. El código de la llamada al sistema puede estar integrado en el archivo *kernel/sys.c* que agrupa un cierto número de llamadas al sistema. El código a añadir es el siguiente:

```
asmlinkage int sys_show_mult(int x, int y, int *res)
{
    int error;
    int compute;

    /* Verificación de la validez de la variable res */
    error = verify_area(VERIFY_WRITE, res, sizeof(*res));
    if (error)
        return error;

    /* cálculo del valor */
    compute = x*y;

    /* Copia el resultado en la memoria de usuario */
    put_user(compute, res);

    printf("Value computed: %d x %d = %d \n", x, y, compute);

    /* Llamada terminada */
    return 0;
}
```

Una vez recompilado el núcleo y reiniciada la máquina, es posible probar la llamada al sistema recién implementada. Para poder utilizar esta llamada al sistema, es necesario declarar una función que ejecuta esta llamada al sistema. Esta función no existe en la biblioteca C, y hay que declararla explícitamente. Varias macroinstrucciones que permiten definir este tipo de funciones se declaran en el archivo de cabecera *<sys-call.h>*. En el caso de una llamada al sistema que acepta tres parámetros, hay que utilizar la macroinstrucción *_syscall3*:

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

_syscall3 (int, show_mult, int, x, int, y, int *resul);

main ()
{
    int ret = 0;

    show_mult (2, 5, &ret);
    printf ("Resultado %d %d = %d \n", 2, 5, ret);
}
```

La macroinstrucción `_syscall3` es expandida por el preprocesador y proporciona el código siguiente:

```
int show_mult(int x, int y, int *resul)
{
    long __res;
    __asm__ __volatile__ (*int $0x80"
        : "=a" (__res)
        : "0" (164 ),
          "b" ((long)( x )),
          "c" ((long)( y )),
          "d" ((long)( resul ));
    if (__res>=0)
        return ( int ) __res;
    errno=__res;
    return -1;
};
```

La función `show_mult` es generada por `_syscall3`, que inicializa los registros del procesador (el número de la llamada al sistema se coloca en el registro `eax` y los parámetros se colocan en los registros `ebx`, `ecx` y `edx`), y seguidamente desencadena la interrupción `0x80`. De vuelta de esta interrupción, es decir, al volver de la llamada al sistema, el valor de retorno se comprueba. Si es positivo o nulo, se devuelve a quien llama. En caso contrario, este valor contiene el código de error devuelto; entonces se guarda en la variable global `errno` y se devuelve el valor `-1`.

6.1.3 Códigos de retorno

El núcleo puede detectar errores en la ejecución de una llamada al sistema. En ese caso, se devuelve un código de error al proceso que hace la llamada.

Generalmente, se devuelve el valor `-1` en caso de error. Este valor indica únicamente que se ha producido un error. A fin de permitir al proceso que ha llamado determinar la causa del error, la biblioteca C proporciona una variable global llamada `errno`. Esta variable se actualiza tras cada llamada al sistema que cause un error, y contiene un código que indica la causa del error. No se actualiza tras una llamada al sistema con éxito, por lo que hay que comprobar el valor de retorno de cada llamada al sistema y utilizar el valor de `errno` sólo en caso de fallo.

El archivo de cabecera `<errno.h>` define numerosas constantes que representan los posibles errores. En los capítulos siguientes, se detalla en la presentación de las llamadas al sistema la lista de códigos de error que pueden ser devueltos.

La biblioteca C proporciona también las dos variables siguientes:

```
extern int _sys_nerr;  
extern char *_sys_errlist[];
```

La variable `_sys_nerr` contiene el número de códigos de error implementados. La tabla `_sys_errlist` contiene las cadenas de caracteres que describen todos los errores. El mensaje de error correspondiente al último error detectado puede obtenerse pues por medio de `_sys_errlist[errno]`.

Además, la biblioteca C proporciona dos funciones:

```
#include <stdio.h>  
#include <errno.h>  
  
void perror (const char *s);  
  
char *strerror (int errnum);
```

La función `perror` muestra un mensaje en la salida de errores del proceso que llama. Este mensaje contiene la cadena especificada por el parámetro `s` y el mensaje de error correspondiente al valor de `errno`. La función `strerror` devuelve la cadena de caracteres correspondiente al error cuyo código se especifica en el parámetro `errnum`.

6.2 Funciones de utilidad

El núcleo contiene numerosas funciones de utilidad que hay que presentar ya ahora. Aunque estas funciones sean descritas en capítulos posteriores, es necesario conocer su funcionamiento para comprender la descripción de las funciones internas.

6.2.1 Manipulación de espacio de direccionamiento

El espacio de direccionamiento de un proceso en modo núcleo es diferente de su espacio de direccionamiento en modo usuario. Le resulta por tanto imposible acceder directamente a los datos cuya dirección se ha pasado como parámetro a una llamada al sistema.

Linux proporciona varias funciones para acceder a estos datos:

```
int verify_area (int type, const void *addr, unsigned long size);

void put_user (unsigned long value, void *addr);
unsigned long get_user (void *addr);

void memcpy_toofs (void *to, const void *from, unsigned long size);
void memcpy_fromfs (void *to, const void *from, unsigned long size);
```

La función `verify_area` verifica que la dirección pasada (parámetro `addr`) es válida. El parámetro `size` representa el número de bytes de la zona, y `type` especifica el tipo de la verificación. Este parámetro puede tomar el valor `VERIFY_READ` para verificar que la zona de memoria es accesible para lectura o `VERIFY_WRITE` para escritura.

La función `put_user` escribe el valor especificado por el parámetro `value` en el espacio de direccionamiento del proceso que llama, en la dirección especificada por el parámetro `addr`. La función `get_user` devuelve el valor situado en la dirección pasada en el parámetro `addr`, en el espacio de direccionamiento del proceso que llama.

La función `memcpy_toofs` copia datos desde el espacio de direccionamiento del núcleo hacia el espacio de direccionamiento del proceso que llama. El parámetro `to` contiene la dirección en el espacio de direccionamiento del proceso, `from` contiene la dirección en el espacio de direccionamiento del núcleo y `size` especifica el número de bytes a copiar. La función `memcpy_fromfs` efectúa la copia de datos desde el espacio de direccionamiento del proceso que llama hacia el espacio de direccionamiento del núcleo.

Estas funciones se detallan en el capítulo 8, sección 6.5.

6.2.2 Asignaciones y liberaciones

Se proporcionan varias funciones de asignación dinámica y liberación de memoria para las necesidades internas del núcleo:

```
void *kmalloc (unsigned int size, int priority);  
void kfree (void *addr);  
  
void *vmalloc (unsigned long size);  
void vfree (void *addr);
```

La función `kmalloc` asigna una zona de memoria cuyo tamaño se especifica por el parámetro `size`, y devuelve la dirección de la zona asignada. En caso de fracaso, se devuelve el valor `NULL`. El parámetro `priority` especifica el tipo de la asignación:

<i>constante</i>	<i>significado</i>
GFP_BUFFER	Asignación de memoria para el búfer caché.
GFP_ATOMIC	Asignación de memoria para un gestor de interrupciones.
GFP_USER	Asignación de memoria para un proceso en modo usuario.
GFP_KERNEL	Asignación de memoria para el núcleo.
GFP_NFS	Asignación de memoria para el soporte del sistema de archivos NFS.

La función `kfree` libera una zona de memoria cuya dirección se pasa en el parámetro `addr`. Esta dirección debe corresponder con un valor devuelto por `kmalloc`.

Las funciones `vmalloc` y `vfree` implementan también la asignación y la liberación de zonas de memoria, que son más flexibles que `kmalloc` y `kfree`.

Estas funciones se detallan en el capítulo 8, sección 5.3.2.

Capítulo 4

Procesos

Primitivas detalladas

_exit, clone, exit, fork, getegid, geteuid, getgid, getgroups, getpgid, getpgrp, getpid, getppid, getpriority, getrlimit, getrusage, getsid, getuid, nice, ptrace, sched_yield, setegid, seteuid, setfsuid, setgid, setgroups, setpgid, setpgrp, setpriority, setregid, setreuid, setrlimit, setsid, setuid, times, wait, wait3, wait4, waitpid

1 Conceptos básicos

1.1 Noción de proceso

De manera informal, puede considerarse como un programa en curso de ejecución. Éste progresa de manera secuencial: en todo momento, se ejecuta una instrucción como máximo de la serie del proceso.

Sin embargo, un proceso no se limita al programa que ejecuta. También se caracteriza por su actividad puntual, representada por el valor del contador ordinal y los registros del procesador. Incluye una pila, que contiene datos temporales, y un segmento de datos que contienen variables globales.

Un programa en sí mismo no es un proceso: un programa es una entidad pasiva (archivo ejecutable residente en un disco), mientras que un proceso es una entidad activa con un contador ordinal que especifica la instrucción siguiente a ejecutar y un conjunto de recursos asociados.

1.2 Estado de un proceso

En el transcurso de su ejecución, un proceso cambia de estado. El estado de un proceso se define por su actividad actual. Los diferentes estados posibles de un proceso son los siguientes:

- en ejecución: el proceso es ejecutado por el procesador;
- a punto: el proceso podría ser ejecutado pero otro proceso se está ejecutando en ese momento;
- suspendido: el proceso está en espera de un recurso; por ejemplo, espera el fin de una entrada/salida;
- parado: el proceso ha sido suspendido por una intervención externa;
- zombi: el proceso ha terminado su ejecución pero sigue siendo referenciado en el sistema.

Los cambios de estado de un proceso se representan en el grafo de estados, figura 4.1.

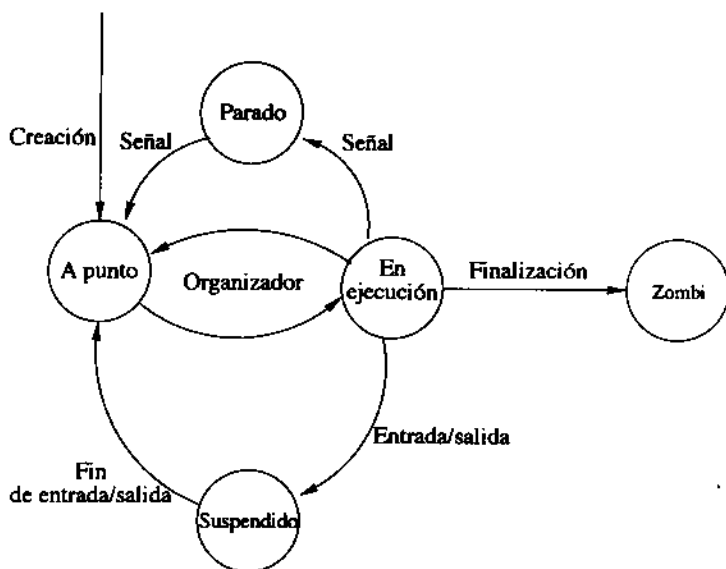


FIG. 4.1 – Grafo de estados de un proceso

1.3 Atributos de un proceso

Durante su ejecución, un proceso se caracteriza por varios atributos mantenidos por el sistema:

- su estado;
- su identificador (número único);
- el valor de los registros, incluyendo el contador ordinal;
- la identidad del usuario bajo cuyo nombre se ejecuta;
- las informaciones utilizadas por el núcleo para proceder al ordenamiento de los procesos (prioridad...);
- las informaciones respecto al espacio de direccionamiento del proceso (segmentos de código, de datos, de pila);
- las informaciones respecto a las entradas/salidas efectuadas por el proceso (descriptores de archivos abiertos, directorio actual...);
- informaciones de contabilidad que resumen los recursos utilizados por el proceso.

1.4 Identificadores de un proceso

A un proceso se le asocian varios identificadores de usuarios y de grupos:

- el identificador de usuario real: es el identificador del usuario que ha lanzado el proceso;
- el identificador de usuario efectivo: es el identificador que utiliza el sistema para los controles de acceso; puede ser distinto del identificador de usuario real, por ejemplo en el caso de programas que poseen el bit *setuid*;
- el identificador de grupo real: es el identificador de grupo primario del usuario que ha lanzado el proceso;
- el identificador de grupo efectivo: es el identificador que utiliza el sistema para los controles de acceso; puede ser diferente del identificador de grupo real, por ejemplo en el caso de programas que poseen el bit *setgid*;

- una lista de identificadores de grupos: todo usuario puede pertenecer a varios grupos de manera simultánea, y el núcleo mantiene una lista de grupos asociados a cada proceso a fin de proceder a los controles de acceso. Bajo Linux, un proceso puede poseer hasta 32 grupos.

Linux mantiene también otros dos identificadores: el *saved uid* y el *saved gid*. Cuando un proceso modifica su identificador de usuario o de grupo efectivo, por una llamada a las primitivas *setreuid* o *setregid*, el núcleo autoriza la modificación en los casos siguientes:

- el proceso posee los privilegios de superusuario;
- el proceso especifica el mismo valor para el nuevo identificador;
- el nuevo identificador es igual al identificador guardado.

Estos identificadores guardados son particularmente útiles para un proceso que ejecute un programa que posee el bit *setuid*, resp. *setgid*, es decir, un proceso que posee identificadores de usuario, res. grupo, real y efectivo diferentes. Este proceso puede utilizar las primitivas *setuid*, resp. *setgid*, y *setreuid*, resp. *setregid*, para anular sus privilegios, utilizando el identificador de usuario o de grupo real, efectuar un tratamiento que no necesita ningún privilegio particular, y luego restaurar su identificador de usuario o de grupo efectivo.

1.5 Filiación

La creación de un proceso se efectúa duplicando el proceso actual: la llamada al sistema *fork* permite a un proceso crear una copia conforme a sí mismo, exceptuando el identificador de proceso. El proceso que se duplica se llama el proceso padre, y el nuevo proceso se llama el proceso hijo.

Cuando Linux arranca, crea el proceso número 1 que ejecuta el programa *init*, encargado de inicializar el sistema. Este proceso crea a su vez procesos para efectuar diferentes tareas, pudiendo éstas a su vez crear nuevos procesos. El resultado es una jerarquía de procesos emparentados.

La figura 4.2 muestra una jerarquía típica de procesos, mediante el mandato *ps tree*.

1.6 Grupos de procesos

Linux mantiene grupos de procesos. Un grupo de procesos es un conjunto que contiene uno o más procesos. Todo proceso forma parte de un grupo, y sus descendientes pertenecen en principio al mismo grupo. Un proceso puede elegir crear un nuevo grupo y

```

init(1) auto
|-(agetty,9700)
|-(agetty,9699)
|-(agetty,9698)
|-(agetty,71)
|-cron(26)
|-innd(63) -p4 -i0 -c360
|   |-overchan(72)
|-(kflushd,2)
|-(kswapd,3)
|-selection(40) -t msc -c 1 -p m
|-syslogd(36)
|-(tcsh,9701)
|   |- (startx,9724)
|       |- (xinit,9725)
|           |-X(9726) :0
|               |-fvwm(9728)
|                   |-FvwmPager(9735) 9 4 /home/card/.fvwmrc 0 8 0 2
|                   |-GoodStuff(9733) 7 4 /home/card/.fvwmrc 0 8
|                   |- (xclock,9734)
|                   |-xterm(9731) -geometry 80x50+20+50
|                       |-tcsh(9737)
|                   |-xterm(9732) -geometry 80x20+530+70
|                       |- (tcsh,9736)
|                           |-bash(10771)
|-update(9)
|-xterm(10311)
|   |-tcsh(10316)
|-xterm(10321)
|   |-tcsh(10328)
|-xterm(10325)
|   |-tcsh(10330)
|       |-vi(30664) ../Processus/ConceptsBase.tex
|           |-tcsh(30716) -c pstree -a -c -p
|               |-pstree(30720) -a -c -p
|-xterm(17889)
|   |-tcsh(17893)
|       |-make(30696)
|           |-latex(30715) Livre.tex
|       |-xdvi(28449) Livre
|           |-gs(28531) -sDEVICE=x11 -dNOPAUSE -q -
|-xterm(10788)
|   |-tcsh(10791)
|-xterm(10790)
|   |-tcsh(10792)

```

FIG. 4.2 – Jerarquía típica de procesos

convertirse así en jefe (*leader*) del grupo. Un grupo se identifica por su número de grupo, que es igual al número de su proceso *leader*.

Esta noción de grupo permite enviar señales a todos los procesos miembros de un grupo (véase el capítulo 5, sección 2.1), a fin de implementar el *job control*. Los intérpretes de mandatos (*bash*, *csh*, *ksh*...) utilizan estos grupos para permitir al usuario suspender la ejecución de procesos y proseguirla, en primer plano o en segundo plano.

Los grupos de procesos se utilizan asimismo en la gestión de terminales (véase el capítulo 9, sección 2.6).

1.7 Sesiones

Una sesión es un conjunto que contiene uno o más grupos de procesos. A cada sesión se le asocia un terminal de control único. Este terminal de control es o bien un dispositivo terminal (en una conexión a la consola), o bien un dispositivo pseudoterminal (en una conexión remota). Cuando un proceso crea una nueva sesión:

- se convierte en el proceso *leader* de la sesión;
- se crea un nuevo grupo de procesos y el proceso que llama se convierte en su *leader*;
- el proceso que llama no tiene terminal de control.

Una sesión se identifica por su número, que es igual al número de su proceso *leader*.

Generalmente, se crea una nueva sesión por parte del programa *login* en la conexión de un usuario. Todos los procesos creados forman parte de la misma sesión. En el caso de encadenamiento de mandatos (por medio de tuberías, por ejemplo), el intérprete de mandatos (*shell*) crea nuevos grupos de procesos, para implementar el *job control*.

1.8 Multiprogramación

En un instante dado, hay un solo proceso en ejecución (al menos en un ordenador mono-procesador). Este proceso se llama el proceso actual. El sistema mantiene una lista de procesos a punto que podría ejecutar y procede periódicamente a un ordenamiento.

A cada proceso se le atribuye un lapso de tiempo. Linux elige un proceso a ejecutar, y le deja ejecutarse durante ese lapso. Cuando ha transcurrido, el sistema hace pasar al proceso actual al estado a punto, y elige otro proceso que ejecuta durante otro lapso. El lapso de tiempo es muy corto y el usuario tiene la impresión que varios procesos se ejecutan simultáneamente, aunque sólo un proceso se ejecuta en un instante dado. Se dice que los procesos se ejecutan en pseudoparalelismo.

2 Llamadas al sistema de base

2.1 Creación de procesos

El proceso actual puede crear un proceso hijo utilizando la llamada al sistema *fork*. Esta primitiva provoca la duplicación del proceso actual. Su prototipo es el siguiente:

```
#include <unistd.h>

pid_t fork (void);
```

En la llamada a *fork*, el proceso actual se duplica: se crea una copia que le es idéntica, excepto en su identificador y el de su padre. Al volver *fork*, dos procesos, el padre y el hijo, ejecutan el mismo código.

La primitiva *fork* devuelve el valor 0 al proceso hijo, y devuelve el identificador del proceso creado al proceso padre. Es pues imperativo comprobar su valor de retorno para distinguir el código que debe ser ejecutado por el proceso padre y el que debe ser ejecutado por el proceso hijo. En caso de fallo, *fork* devuelve el valor -1, y la variable *errno* puede tomar los valores siguientes:

error	significado
EAGAIN	Se ha llegado al número máximo de procesos del usuario actual o del sistema.
ENOMEM	El núcleo no ha podido asignar suficiente memoria para crear un nuevo proceso.

El programa siguiente ilustra la llamada al sistema *fork* para crear un proceso hijo.

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

void main (void)
{
    pid_t      pid;

    pid = fork ();
    if (pid == -1)
        perror ("fork");
    else if (pid == 0)
        printf ("soy el hijo: pid = %d\n",pid);
    else
        printf ("soy el padre: pid = %d\n",pid);
}
```

2.2 Finalización del proceso actual

El proceso actual termina automáticamente cuando deja de ejecutar la función `main`, utilizando la instrucción `return`. También dispone de llamadas al sistema que le permiten parar explícitamente su ejecución:

```
#include <unistd.h>

void _exit (int status);

void exit (int status);
```

Las primitivas `_exit` y `exit` provocan la finalización del proceso actual. El parámetro `status` especifica un código de retorno, comprendido entre 0 y 255, que hay que comunicar al proceso padre. Por convención, un proceso debe devolver el valor 0 en caso de finalización normal, y un valor no nulo en caso de finalización debida a un error.

Antes de terminar la ejecución del proceso, `exit` ejecuta las funciones eventualmente registradas por llamadas a las funciones de biblioteca `atexit` y `on_exit`.

Si el proceso actual posee procesos hijos, éstos se vinculan al proceso número 1, que ejecuta el programa `init`. La señal `SIGCHLD` se envía al proceso padre para prevenirle de la finalización de uno de sus procesos hijos.

2.3 Espera de la finalización de un proceso hijo

Un proceso puede alcanzar la finalización de un proceso hijo mediante las primitivas `wait` y `waitpid`. Su sintaxis es la siguiente:

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

pid_t wait (int *statusp);

pid_t waitpid (pid_t pid, int *statusp, int options);
```

La primitiva `wait` suspende la ejecución del proceso actual hasta que acabe un proceso hijo. Si un proceso hijo ya ha terminado, `wait` devuelve el resultado inmediatamente.

La primitiva `waitpid` suspende la ejecución del proceso actual hasta que un proceso hijo especificado por el parámetro `pid` termine. Si un proceso hijo correspondiente a `pid` ha terminado ya, `waitpid` devuelve el resultado inmediatamente.

El resultado de *waitpid* depende del valor del parámetro *pid*:

- si *pid* es positivo, especifica el número del proceso hijo a esperar;
- si *pid* es nulo, especifica todo proceso hijo cuyo número de grupo de procesos sea igual al del proceso que llama;
- si *pid* es igual a -1, especifica la espera de la finalización del primer proceso hijo; en este caso, *waitpid* ofrece la misma semántica que *wait*;
- si *pid* es inferior a -1, especifica todo proceso hijo cuyo número de grupo de procesos es igual al valor absoluto de *pid*.

Dos constantes, declaradas en `<sys/wait>`, pueden utilizarse para inicializar el parámetro *options*, a fin de modificar el comportamiento de *waitpid*:

constante	significado
WNOHANG	Provoca un error inmediato si no ha terminado aún ningún proceso hijo.
WUNTRACED	Provoca la consideración de los hijos cuyo estado cambia, es decir, los procesos hijos cuyo estado ha pasado de a punto a suspendido.

Las dos primitivas devuelven el número del proceso hijo que ha terminado, o el valor -1 en caso de fallo. El estado del proceso hijo se devuelve en la variable cuya dirección se pasa en el parámetro *statusp*.

La interpretación de este estado se efectúa gracias a macroinstrucciones definidas en el archivo de cabecera `<sys/wait.h>`:

macroinstrucción	significado
WIFEXITED	No nulo si el proceso hijo ha terminado por una llamada a <code>_exit</code> o <code>exit</code> .
WEXITSTATUS	Código de retorno transmitido por el proceso hijo en su finalización.
WIFSIGNALED	No nulo si el proceso hijo ha sido interrumpido por la recepción de una señal.
WTERMSIG	Número de señal que ha provocado la finalización del proceso hijo.
WIFSTOPPED	No nulo si el proceso hijo ha pasado del estado a punto a suspendido.
WSTOPSIG	Número de señal que ha causado la suspensión del proceso hijo.

Linux implementa también otras dos llamadas al sistema que aseguran una compatibilidad con los sistemas BSD Unix.

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

```
pid_t wait3 (int *statusp, int options, struct rusage *rusage);
```

```
pid_t wait4 (pid_t pid, int *statusp, int options, struct rusage *rusage);
```

Estas llamadas son relativamente parecidas a *wait* y *waitpid*. Aceptan un parámetro suplementario, *rusage*, que apunta a una variable en la que se colocan las informaciones de compatibilidad del proceso. La estructura *rusage*, definida en el archivo de cabecera `<sys/resource.h>`, contiene los campos siguientes:

<i>tipo</i>	<i>campo</i>	<i>descripción</i>
struct timeval	ru_utime	Tiempo de procesador consumido por el proceso en modo usuario.
struct timeval	ru_stime	Tiempo de procesador consumido por el proceso en modo núcleo.
long	ru_maxrss	Tamaño máximo del espacio de direccionamiento del proceso residente en memoria, en kilobytes.
long	ru_ixrss	Número de kilobytes del espacio de direccionamiento del proceso, compartidos con otros procesos.
long	ru_idrss	Número de kilobytes del segmento de datos del proceso, no compartidos con otros procesos.
long	ru_isrss	Número de kilobytes del segmento de pila del proceso.
long	ru_minflt	Número de fallos de página provocados por el proceso que se han resuelto sin lanzar una entrada/salida.
long	ru_majflt	Número de fallos de página provocados por el proceso que se han resuelto lanzando una entrada/salida.
long	ru_nswap	Número de veces que el proceso ha sido descartado de la memoria para ubicarlo en memoria secundaria.
long	ru_inblock	Número de veces que el sistema de archivos ha efectuado una lectura de datos para el proceso.
long	ru_oublock	Número de veces que el sistema de archivos ha efectuado una escritura de datos para el proceso.
long	ru_msgsnd	Número de mensajes enviados por el proceso.
long	ru_msgrcv	Número de mensajes recibidos por el proceso.
long	ru_nsignals	Número de señales recibidas por el proceso.
long	ru_nvcsw	Número de veces que el proceso ha abandonado voluntariamente el procesador (generalmente para esperar la llegada de un evento, como el fin de una entrada/salida).
long	ru_nivcsw	Número de veces que el proceso ha abandonado el procesador porque había agotado su porción de tiempo o porque un proceso más prioritario entró en el estado a punto.

La estructura `timeval`, utilizada para definir los campos `ru_utime` y `ru_stime`, es la siguiente:

<i>tipo</i>	<i>campo</i>	<i>descripción</i>
long	tv_sec	Segundos
long	tv_usec	Microsegundos

La función `wait3` corresponde a la llamada al sistema `wait4`, a la que se pasa el valor `-1` para el parámetro `pid`.

En caso de fallo de estas primitivas, la variable `errno` puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
ECHILD	El proceso especificado por <code>pid</code> no existe.
EFAULT	<code>statusp</code> o <code>rusage</code> contiene una dirección inválida.
EINTR	La llamada al sistema ha sido interrumpida por la recepción de una señal.
EPERM	El proceso que llama no es privilegiado y su identificador de usuario efectivo no es igual al del proceso especificado por <code>pid</code> .

2.4 Lectura de los atributos del proceso actual

Un proceso cuenta con varias llamadas al sistema para obtener los atributos que le caracterizan:

```
#include <unistd.h>

pid_t getpid (void);
pid_t getppid (void);
uid_t getuid (void);
uid_t geteuid (void);
gid_t getgid (void);
gid_t getegid (void);
int getgroups (int size, gid_t list[]);
```

`getpid` devuelve el identificador único del proceso actual, `getppid` devuelve el identificador único del padre del proceso actual.

getuid devuelve el identificador del usuario real del proceso, *geteuid* devuelve el identificador de usuario efectivo del proceso actual.

getgid devuelve el identificador de grupo real del proceso, *getegid* devuelve el identificador de grupo efectivo del proceso actual.

Para terminar, *getgroups* permite obtener la lista de grupos asociados al proceso actual. Los identificadores de estos grupos se devuelven en el parámetro *list*, y el parámetro *size* especifica el tamaño de la tabla *list*. *getgroups* devuelve el número total de grupos asociados al proceso actual.

2.5 Modificación de atributos

Un proceso puede, en ciertas circunstancias, modificar sus atributos. Linux proporciona varias llamadas al sistema para ello:

```
#include <unistd.h>

int setuid (uid_t uid);

int setreuid (uid_t ruid, uid_t euid);

int seteuid (uid_t euid);

int setgid (gid_t gid);

int setregid (gid_t rgid, gid_t egid);

int setegid (gid_t egid);

#define __USE_BSD
#include <grp.h>

int setgroups (size_t size, const gid_t *list);
```

La llamada *setuid* permite que un proceso modifique su identificador de usuario efectivo. El proceso debe ser privilegiado, es decir, debe poseer los derechos del superusuario, o el nuevo identificador debe ser igual al anterior o al identificador del usuario guardado. En el caso que el proceso sea privilegiado, el identificador de usuario real y el identificador de usuario guardado también se modifican, lo que significa que un proceso que posea privilegios del superusuario que modifica su identificador de usuario por *setuid* no puede restaurar ya sus privilegios.

Los identificadores de usuario real y efectivo del proceso actual pueden modificarse por *setreuid*. Los parámetros *ruid* y *euid* representan estos identificadores. Si se utiliza el valor -1, el identificador correspondiente no se modifica. Un proceso no privi-

legiado puede invertir estos dos identificadores. La llamada *seteuid* corresponde a una llamada de *setreuid* donde el parámetro *ruid* tiene el valor -1.

La llamada *setgid* permite a un proceso modificar su identificador de grupo efectivo. El proceso debe ser privilegiado o el nuevo identificador debe ser igual al anterior o al identificador de grupo guardado. En el caso en que el proceso sea privilegiado, el identificador de grupo real y el identificador de grupo guardado también se modifican.

Los identificadores de grupo real y efectivo del proceso actual pueden modificarse por *setregid*. Los parámetros *rgid* y *egid* representan estos identificadores. Si se utiliza el valor -1, el identificador correspondiente no se modifica. Un proceso no privilegiado puede invertir estos dos identificadores. La llamada *setegid* corresponde a una llamada de *setregid* donde el parámetro *rgid* tiene el valor -1.

La llamada al sistema *setgroups* permite modificar los identificadores de grupo asociados al proceso actual. El parámetro *list* especifica los grupos a posicionar, con el tamaño de la tabla indicado por el parámetro *size*. Sólo un proceso privilegiado puede modificar los grupos que le están asociados.

Además de estas llamadas al sistema, Linux ofrece también dos primitivas especializadas respecto a los controles de acceso a los archivos:

```
int setfsuid (uid_t fsuid);
```

```
int setfsgid (gid_t fsgid);
```

La primitiva *setfsuid* modifica el identificador de usuario del cual se sirve el núcleo en los controles de acceso a los archivos.

La primitiva *setfsgid* modifica el identificador de grupo del que se sirve el núcleo en los controles de acceso a los archivos.

Estas dos llamadas al sistema normalmente no son utilizadas por un proceso «clásico». De modo predeterminado, el núcleo utiliza los identificadores de usuario efectivo y de grupo efectivo para los controles de acceso a los archivos, lo que corresponde a su semántica clásica. Estas dos primitivas están sin embargo disponibles para permitir a los servidores acceder a los archivos utilizando los derechos de un usuario y de un grupo particulares, cuando tratan una petición por cuenta de dicho usuario. El servidor NFS, por ejemplo, utiliza estas dos primitivas para modificar sus derechos a cada petición de acceso a los archivos.

Todas estas llamadas al sistema devuelven el valor 0 en caso de éxito, o el valor -1 en caso de fallo. La variable *errno* puede tomar el valor *EPERM* que indica que el proceso no posee los privilegios necesarios para modificar sus identificadores.

2.6 Información de contabilidad

La llamada al sistema *getrusage* permite que un proceso conozca los recursos que ha consumido. Su sintaxis es la siguiente:

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
```

```
int getrusage (int who, struct rusage *rusage);
```

El parámetro *who* especifica a qué proceso debe aplicarse la operación: puede tomar el valor *RUSAGE_SELF* para obtener los recursos consumidos por el proceso actual, o *RUSAGE_CHILDREN* para obtener los consumidos por los procesos hijos. El resultado se devuelve en una variable apuntada por el parámetro *rusage* (véase la sección 2.3 para la definición de la estructura *rusage*). En caso de éxito, se devuelve el valor 0, y en caso de fallo *getrusage* devuelve el valor -1 y la variable *errno* puede tomar el valor *EFAULT*, que indica que el parámetro *rusage* contiene una dirección inválida.

Linux proporciona también la primitiva *times* que permite a un proceso obtener el tiempo de procesador que ha consumido. Su sintaxis es la siguiente:

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

clock_t times (struct tms *buf);
```

El parámetro *buf* especifica la dirección de una variable en la que se devolverá el resultado. La estructura *tms*, definida en el archivo de cabecera *<sys/times.h>*, contiene los campos siguientes:

tipo	campo	descripción
time_t	tms_utime	Tiempo de procesador consumido por el proceso en modo usuario, expresado en segundos.
time_t	tms_stime	Tiempo de procesador consumido por el proceso en modo núcleo, expresado en segundos.
time_t	tms_cutime	Tiempo de procesador consumido por los procesos hijos en modo usuario, expresado en segundos.
time_t	tms_cstime	Tiempo de procesador consumido por los procesos hijos en modo núcleo, expresado en segundos.

times devuelve el número de ciclos de reloj transcurridos desde el arranque del sistema. Si el parámetro *buf* contiene una dirección inválida, devuelve el valor -1 y la variable *errno* toma el valor *EFAULT*.

2.7 Límites

Un proceso puede imponer límites sobre los recursos que puede consumir. Generalmente, estos límites se posicionan en la conexión al sistema, y son heredados por los procesos creados, pero un proceso puede modificarlos mediante las llamadas al sistema:

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
```

```
int setrlimit (int resource, const struct rlimit *rlim);
```

```
int getrlimit (int resource, struct rlimit *rlim);
```

La primitiva *setrlimit* permite posicionar un límite. El parámetro *resource* indica el recurso a limitar, y el límite se especifica por el parámetro *rlim*. En caso de éxito, se devuelve el valor 0, si no *setrlimit* devuelve el valor -1.

La estructura *rlimit*, definida en el archivo *<sys/resource.h>*, contiene los campos siguientes:

tipo	campo	descripción
int	<i>rlim_cur</i>	Límite «suave»
int	<i>rlim_max</i>	Límite absoluto

Un proceso no privilegiado puede aumentar su límite «flexible», que no puede exceder el límite absoluto, o disminuir éste. Sólo un proceso privilegiado puede aumentar este límite.

La llamada al sistema *getrlimit* permite a un proceso conocer el límite asociado a un recurso. En caso de éxito, se devuelve el valor 0, si no *setrlimit* devuelve el valor -1.

Para estas dos llamadas al sistema, se definen constantes en el archivo de cabecera *<sys/resource.h>* para representar los diferentes recursos:

constante	significado
RLIMIT_CPU	Tiempo de procesador expresado en milisegundos
RLIMIT_FSIZE	Tamaño máximo de archivo, expresado en bytes
RLIMIT_DATA	Tamaño máximo del segmento de datos, expresado en bytes
RLIMIT_STACK	Tamaño máximo del segmento de pila, expresado en bytes
RLIMIT_CORE	Tamaño máximo del archivo <i>core</i> a crear en caso de error fatal del programa, expresado en bytes
RLIMIT_RSS	Tamaño máximo de los datos residentes en memoria central, expresado en bytes

RLIMIT_NPROC	Número máximo de procesos por usuario
RLIMIT_NOFILE	Número máximo de archivos abiertos
RLIMIT_MEMLOCK	Tamaño máximo de datos bloqueados en memoria central, expresado en bytes
RLIMIT_AS	Tamaño máximo del espacio de direccionamiento, expresado en bytes

En caso de fallo de estas primitivas, la variable `errno` puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EFAULT	<code>rlim</code> contiene una dirección inválida
EINVAL	<code>resource</code> contiene un valor inválido
EPERM	El proceso que llama no posee los privilegios que le permitan aumentar su límite absoluto

2.8 Grupos de procesos

Linux proporciona varias llamadas al sistema que permiten gestionar los grupos de procesos:

```
#include <unistd.h>

int setpgid (pid_t pid, pid_t pgid);

pid_t getpgid (pid_t pid);

int setpgrp (void);

pid_t getpgrp (void);
```

La primitiva `setpgid` modifica el grupo asociado al proceso especificado por el parámetro `pid`. El parámetro `pgid` especifica el número del grupo. Si `pid` es nulo, la modificación se aplica al proceso actual. Si `pgid` es nulo, el número del proceso actual se utiliza como número de grupo.

La llamada al sistema `getpgid` devuelve el número del grupo al que pertenece el proceso especificado por el parámetro `pid`. Si `pid` es nulo, se devuelve el número del grupo del proceso actual. La llamada al sistema `getpgrp` devuelve el número del grupo del proceso actual.

La función `setpgrp` modifica el número del grupo del proceso actual y corresponde a la llamada `setpgid (0, 0)`.

En caso de fallo de estas primitivas, la variable `errno` puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EINVAL	<code>pgid</code> contiene un valor negativo
EPERM	El proceso no está autorizado a modificar el grupo al que pertenece el proceso especificado por <code>pid</code>
ESRCH	El proceso especificado por <code>pid</code> no existe

2.9 Sesiones

Dos llamadas al sistema permiten manipular las sesiones de procesos:

```
pid_t setsid (void);
```

```
pid_t getsid (pid_t pid);
```

La llamada al sistema *setsid* crea una nueva sesión. El proceso actual no debe ser un *leader* de grupo de procesos. Como consecuencia de esta llamada, el proceso actual es a la vez el *leader* de una nueva sesión y el *leader* de un nuevo grupo de procesos, y además no tiene asociado ningún terminal. El número del proceso actual se utiliza como identificador de la nueva sesión y del nuevo grupo de procesos. Este identificador es devuelto por *setsid*. En caso de fallo, se devuelve el valor `-1`, y la variable `errno` toma el valor `EPERM`, que indica que el proceso actual es ya un *leader* de grupo de procesos.

La asociación de un terminal de control a una sesión se efectúa automáticamente cuando el proceso *leader* de la sesión abre un dispositivo terminal o pseudoterminal aún no asociado.

La primitiva *getsid* devuelve el número de sesión asociada al proceso especificado por el parámetro `pid`. Si `pid` es nulo, se devuelve el número de la sesión del proceso actual. En caso de fallo, se devuelve el valor `-1`, y la variable `errno` toma el valor `ESRCH`, que indica que el proceso especificado por el parámetro `pid` no existe.

2.10 Ejecución de programa

Un nuevo proceso, creado por una llamada a la primitiva *fork*, es una copia conforme de su proceso padre, y por tanto ejecuta el mismo programa. Una llamada al sistema permite que un proceso ejecute un nuevo programa:

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

```
int execve (const char *pathname, const char *argv [],const char *envp[]);
```

La primitiva *execve* provoca la ejecución de un nuevo programa. *pathname* especifica el nombre del archivo a ejecutar, que debe ser un programa binario o un archivo de mandatos que empiece por la línea `#!nombre_de_intérprete`. El parámetro *argv* indica los argumentos del programa a ejecutar: cada elemento de la matriz *argv* debe apuntar a una cadena de caracteres que representa un argumento. El primer elemento de la matriz debe contener el nombre del programa, los elementos siguientes contienen los argumentos, y el último elemento debe contener el valor `NULL`. El parámetro *envp* especifica las variables de entorno del programa. Cada uno de los elementos debe contener la dirección de una cadena de caracteres de la forma `nombre_de_vari-ble=valor`, y el último elemento de la matriz debe contener el valor `NULL`.

La llamada al sistema *execve* provoca el recubrimiento de los segmentos de código, de datos y de pila por los del programa especificado. En caso de éxito no hay pues retorno, porque el proceso que llama ejecuta ahora un nuevo programa. En caso de fallo, *execve* devuelve el valor `-1`, y la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EINVAL	La lista de argumentos o variables de entorno es de tamaño excesivo
EINVAL	El proceso no tiene acceso en ejecución al archivo especificado por <i>pathname</i>
EFAULT	<i>pathname</i> contiene una dirección inválida
ENAMETOOLONG	<i>pathname</i> especifica un nombre de archivo demasiado largo
ENOENT	<i>pathname</i> se refiere a un nombre de archivo inexistente
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENOEXEC	El archivo especificado por <i>pathname</i> no es un programa ejecutable
ENOMEM	La memoria disponible es insuficiente para ejecutar el programa
ENOTDIR	Uno de los componentes de <i>pathname</i> , utilizado como nombre de directorio, no es un directorio
EPERM	El sistema de archivos que contiene el archivo especificado por <i>pathname</i> se ha montado con opciones que impiden la ejecución de programas

Linux proporciona diversas funciones de biblioteca que ofrecen variantes de *execve*:

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

```
int execl (const char *pathname, const char *arg, ...);
```

```
int execl (const char *pathname, const char *arg, ..., char *const envp[]);
```

```
int execlp (const char *pathname, const char *arg, ...);
```

```
int execl(const char *pathname, char *const argv[]);
```

```
int execvp(const char *pathname, char *const argv[]);
```

Para todas las funciones, el parámetro *pathname* especifica el programa a ejecutar. El parámetro *argv* de *execl* y *execvp* indica los parámetros a transmitir al programa, de la misma manera que para *execve*. El parámetro *envp* indica las variables de entorno a transmitir al programa. Finalmente, en el caso de las funciones *execl*, *execle* y *execlp*, los argumentos del programa se citan por parámetros explícitos: *arg* debe contener la dirección de una cadena de caracteres que representa el nombre del programa y debe ir seguido de una lista de punteros con la dirección de los argumentos, y un último parámetro de la lista que debe valer *NULL*.

Las dos funciones *execlp* y *execvp* consideran *pathname* como un nombre simple de mandato, contrariamente a las otras funciones y a *execve*: el camino de búsqueda asociado al proceso actual (variable de entorno *PATH*) se utiliza para buscar el programa ejecutable especificado.

3 Conceptos avanzados

3.1 Coordinador

El coordinador es la función del núcleo que decide qué proceso debe ser ejecutado por el procesador: el coordinador explora la lista de procesos a punto y utiliza varios criterios para elegir el proceso a ejecutar.

El coordinador de Linux proporciona tres políticas de coordinación diferentes: una para los procesos «normales», y dos para los procesos de «tiempo real».

A cada proceso se le asocia un tipo de proceso, una prioridad fija y una prioridad variable. El tipo de proceso puede ser:

- *SCHED_FIFO* para un proceso de «tiempo real» no preemtivo;
- *SCHED_RR* para un proceso de «tiempo real» preemtivo;
- *SCHED_OTHER* para un proceso clásico.

La política de coordinación depende del tipo de procesos contenidos en la lista de procesos a punto de ejecutar:

- Cuando un proceso de tipo `SCHED_FIFO` está a punto, se ejecuta inmediatamente. El coordinador prioriza la elección del proceso de tipo `SCHED_FIFO` que posea la más alta prioridad, y lo ejecuta. Este proceso no es normalmente preemptible, es decir, que posee el procesador, y el sistema sólo interrumpirá su ejecución en tres casos:
 1. otro proceso de tipo `SCHED_FIFO` que posea una prioridad más elevada pasa al estado a punto: se ejecuta inmediatamente;
 2. el proceso se suspende en espera de un evento, como por ejemplo el fin de una entrada/salida;
 3. el proceso abandona voluntariamente el procesador por una llamada a la primitiva `sched_yield`. El proceso pasa al estado a punto y se ejecutan otros procesos.
- Cuando un proceso de tipo `SCHED_RR` está a punto, se ejecuta inmediatamente. Su comportamiento es similar al de un proceso de tipo `SCHED_FIFO`, con una excepción: cuando el proceso se ejecuta, se le atribuye un lapso de tiempo. Cuando este lapso expira, puede elegirse y ejecutarse un proceso de tipo `SCHED_FIFO` o `SCHED_RR` de prioridad superior o igual a la del proceso actual.
- Los procesos de tipo `SCHED_OTHER` únicamente pueden ejecutarse cuando no existe ningún proceso de «tiempo real» en estado a punto. El proceso a ejecutar se elige tras examinar las prioridades dinámicas. La prioridad dinámica de un proceso se basa por una parte en el nivel especificado por el usuario por las llamadas al sistema `nice` y `setpriority`, y por otra parte en una variación calculada por el sistema. Todo proceso que se ejecute durante varios ciclos de reloj disminuye en prioridad y puede así llegar a ser menos prioritario que los procesos que no se ejecutan, cuya prioridad no se ha modificado.

Puede encontrarse una definición más completa de estas diferentes políticas de coordinación en [Gallmeister 1995].

3.2 Personalidades

A fin de permitir la ejecución de programas provenientes de otros sistemas operativos, Linux soporta la noción de personalidad. A cada proceso se le asocia un ámbito de ejecución. Este ámbito especifica la forma como el proceso efectúa llamadas al sistema, y la forma como se tratan las señales.

Una personalidad define cómo se tratan:

- las llamadas al sistema: Linux utiliza una interrupción lógica para pasar al modo núcleo, mientras que otros sistemas Unix utilizan un salto intersegmento;
- los números de señales especificados por los procesos: cuando un proceso especifica un número de señal, por ejemplo llamando a las primitivas *sigaction* o *kill*, el número de la señal se convierte utilizando una tabla de correspondencias;
- los números de señales enviadas a los procesos: cuando una señal debe enviarse a un proceso, su número se convierte utilizando una tabla de correspondencias.

De modo predeterminado, los procesos utilizan el ámbito de ejecución nativo de Linux, que especifica que las llamadas al sistema se efectúan por interrupción lógica y los números de señales no se convierten. Un sistema de emulación puede utilizar sin embargo ámbitos de ejecución diferentes para ejecutar programas binarios provenientes de otros sistemas operativos.

3.3 Clonado

Bajo Linux, de la misma manera que bajo Unix, los procesos poseen un espacio de direccionamiento diferente, y deben utilizar medios de comunicación especializados como las tuberías (véase capítulo 10) o los IPC System V (véase capítulo 11), para intercambiar datos.

Pero Linux ofrece una extensión que permite crear «clones» de procesos. Un proceso clon se crea, por la primitiva *clone*, por duplicación de su proceso padre. Pero, contrariamente a un proceso clásico, puede compartir una parte de su contexto con su padre.

Según las opciones especificadas en la llamada a *clone*, se puede compartir una o más partes de su contexto:

- El espacio de direccionamiento: los dos procesos comparten los mismos segmentos de código y de datos. Toda modificación efectuada por uno es visible por parte del otro.
- Las informaciones de control del sistema de archivos: los dos procesos comparten los mismos directorios raíz y actual. Si uno de estos directorios es modificado por uno de los procesos (por las primitivas *chdir* o *chroot*), la modificación es efectiva para el otro.
- Los descriptores de archivos abiertos: los dos procesos comparten los mismos descriptores de archivos abiertos. Si uno de ellos cierra un archivo, el otro ya no puede acceder a él.

- Los gestores de señales: los dos procesos comparten la tabla de funciones llamadas en la recepción de una señal. Toda modificación por parte de uno de los procesos, mediante la primitiva *sigaction* por ejemplo, provoca el cambio de la rutina de tratamiento de la señal por el otro proceso.
- El identificador de procesos: los dos procesos pueden compartir el mismo número.

En el caso extremo en que los dos procesos compartan el máximo de cosas, se diferencian únicamente por el valor de los registros del procesador y por su segmento de pila.

Esta posibilidad de clonado permite, entre otras cosas, implementar servidores en los que se ejecuten varias actividades (*threads*). Estas actividades pueden compartir simplemente datos, sin emplear mecanismos de comunicación interprocesos.

4 Llamadas al sistema complementarias

4.1 Cambio de personalidad

Linux permite a un proceso que modifique su personalidad, a fin de poder ejecutar un programa proveniente de otro sistema operativo. Esta posibilidad es utilizada por los emuladores, por ejemplo el emulador iBCS2 que permite ejecutar programas binarios prvistos para SCO Unix o System V Release 4.

La llamada al sistema *personality* permite que un proceso modifique su ámbito de ejecución, a fin de que Linux emule el comportamiento de otro sistema operativo. No se incluye en la biblioteca C estándar y debe declararse pues explícitamente:

```
#include <syscall.h>

_syscall1 (int personality, int pers);
```

Esta declaración corresponde al prototipo siguiente:

```
int personality (int pers);
```

El parámetro *pers* especifica el sistema operativo a emular. En caso de éxito, la antigua personalidad se devuelve en *personality*. En caso de error, se devuelve el valor -1, y la variable *errno* toma el valor *EINVAL*, que indica que el parámetro *pers* contiene un valor inválido.

El archivo de cabecera `<linux/personality.h>` define varias constantes que especifican los sistemas operativos a emular:

constante	significado
PER_LINUX	Linux, es decir, ninguna emulación
PER_SVR4	System V Release 4
PER_SVR3	System V Release 3
PER_SCO3VR3	SCO Unix versión 3.2
PER_WISeV386	UNIX System V/386 Release 3.2.1
PER_ISCR4	Interactive Unix
PER_BSD	BSD Unix
PER_XENIX	Xenix

4.2 Modificación de la coordinación

Varias llamadas al sistema permiten modificar la política y los parámetros de coordinación asociados a un proceso:

```
#include <sched.h>

int sched_setscheduler (pid_t pid, int policy,
                       const struct sched_param *param);

int sched_getscheduler (pid_t pid);

int sched_setparam (pid_t pid, const struct sched_param *param);

int sched_getparam (pid_t pid, struct sched_param *param);
```

La llamada al sistema *sched_setscheduler* permite modificar la política y los parámetros de coordinación asociados a un proceso. El parámetro *pid* especifica el proceso sobre el que conviene actuar. Puede ser nulo para indicar el proceso actual. El parámetro *policy* especifica la política de coordinación a aplicar al proceso, y debe tener uno de los valores *SCHED_FIFO*, *SCHED_RR*, o *SCHED_OTHER*. El parámetro *param* especifica los parámetros de coordinación a utilizar. La primitiva *sched_getscheduler* permite obtener la política de coordinación asociada al proceso caracterizado por *pid*, o al proceso actual si *pid* es nulo.

La llamada al sistema *sched_setparam* permite modificar los parámetros de coordinación asociados a un proceso. El parámetro *pid* especifica el proceso sobre el cual conviene actuar. Puede ser nulo para indicar el proceso actual. El parámetro *param* indica los parámetros de coordinación a utilizar. La primitiva *sched_getparam* permite obtener los parámetros de coordinación asociados al proceso caracterizado por *pid*, o

al proceso actual si `pid` es nulo. Estos parámetros se envían en la variable cuya dirección se pasa en el parámetro `param`.

Linux 2.0 sólo permite modificar la prioridad estática asociada a un proceso. La estructura `sched_param`, definida en el archivo de cabecera `<sched.h>`, contiene un solo campo, `sched_priority`:

tipo	campo	descripción
int	<code>sched_priority</code>	Prioridad estática asociada al proceso

En caso de fallo, esas primitivas devuelven el valor `-1`, y la variable `errno` puede tomar los valores siguientes:

error	significado
EFAULT	<code>param</code> contiene una dirección inválida
EINVAL	<code>policy</code> o la variable apuntada por <code>param</code> contiene un valor inválido
EPERM	El proceso no posee los privilegios necesarios para modificar sus parámetros de ordenación
ESRCH	El proceso especificado por <code>pid</code> no existe

Las primitivas `sched_get_priority_min` y `sched_get_priority_max` permiten obtener las prioridades estáticas mínima y máxima asociadas a una política de coordinación. Su sintaxis es la siguiente:

```
#include <sched.h>

int sched_get_priority_min (int policy);

int sched_get_priority_max (int policy);
```

La primitiva `sched_rr_get_interval` permite obtener el lapso de tiempo atribuido a un proceso de tipo `SCHED_RR`. Su sintaxis es la siguiente:

```
#include <sched.h>

int sched_rr_get_interval (pid_t pid, struct timespec *interval);
```

El parámetro `pid` especifica el proceso. Puede ser nulo para indicar el proceso actual. El proceso en cuestión debe ser de tipo `SCHED_RR`. El parámetro `interval` debe contener la dirección de una variable en la que se colocará el lapso de tiempo asociado al proceso especificado. En caso de éxito, se devuelve el valor `0`, si no `sched_rr_get_interval` devuelve el valor `-1` y la variable `errno` puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EFAULT	<i>interval</i> contiene una dirección inválida
ENOSYS	La llamada al sistema no está implementada (ocurre en Linux 2.0)
ESRCH	El proceso especificado por el parámetro <i>pid</i> no existe

La estructura `timespec` comporta los campos siguientes:

<i>tipo</i>	<i>campo</i>	<i>descripción</i>
long	<code>tv_sec</code>	Número de segundos
long	<code>tv_nsec</code>	Número de nanosegundos

La primitiva `sched_yield` permite al proceso actual liberar al procesador. Su sintaxis es la siguiente:

```
#include <sched.h>

int sched_yield (void);
```

Al ejecutar esta primitiva, el proceso actual se coloca al final de la lista de procesos a punto, y se ejecuta el coordinador. Si existen otros procesos a punto en el sistema, cambia el proceso actual.

4.3 Prioridades de los procesos

Linux proporciona varias llamadas al sistema que permiten manipular las prioridades de los procesos:

```
#include <unistd.h>

int nice (int inc);

#include <sys/resource.h>

int setpriority (int which, int who, int prio);

int getpriority (int which, int who);
```

La primitiva `nice` permite modificar la prioridad dinámica del proceso actual. El parámetro `inc` se añade a la prioridad actual. Sólo un proceso privilegiado puede especificar un valor negativo a fin de aumentar su prioridad. En caso de éxito, se devuelve el valor 0, si no `nice` devuelve el valor -1 y la variable `errno` toma el valor `EPERM`, que indica que el proceso que llama no posee los privilegios necesarios para aumentar su prioridad.

La llamada al sistema *setpriority* modifica la prioridad de un proceso, de un grupo de procesos o de todos los procesos de un usuario. El parámetro *which* puede tomar como valor `PRIO_PROCESS`, `PRIO_PGRP` o `PRIO_USER`. El parámetro *who* especifica un número de procesos, de grupo de procesos o de usuario según el valor del *which*. El parámetro *prio* indica la prioridad a colocar. Sólo un proceso privilegiado puede dar a uno o más procesos mayor prioridad. La primitiva *setpriority* devuelve el valor 0 en caso de éxito, o el valor -1 en caso de error.

La llamada al sistema *getpriority* permite obtener la prioridad de un proceso, de un grupo de procesos o de todos los procesos de un usuario. Los parámetros *which* y *who* tienen el mismo significado que en *setpriority*. El valor devuelto es la prioridad más importante atribuida a los procesos especificados. En la medida en que *getpriority* puede devolver el valor -1 sin que ello indique un error, es necesario volver a poner a 0 la variable *errno*, y comprobar su valor tras el retorno de la llamada al sistema.

En caso de fallo de estas dos primitivas, la variable *errno* puede tomar los valores siguientes:

error	significado
EACCES	El proceso que llama no posee los privilegios necesarios para aumentar la prioridad de otros procesos
EINVAL	<i>which</i> contiene un valor inválido
EPERM	El proceso actual no posee los mismos identificadores de usuarios reales y efectivos que el proceso o procesos especificados por <i>who</i>
ESRCH	Ningún proceso corresponde a la combinación especificada por los parámetros <i>which</i> y <i>who</i>

4.4 Control de la ejecución de un proceso

La llamada al sistema *ptrace* permite que un proceso controle la ejecución de otro proceso. Su sintaxis es la siguiente:

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

```
int ptrace (long request, pid_t pid, long addr, long data);
```

El parámetro *request* especifica la operación a efectuar sobre el proceso cuyo número se pasa en el parámetro *pid*. El significado de los parámetros *addr* y *data* depende del valor de *request*.

Las operaciones disponibles se definen en el archivo de cabecera `<sys/ptrace.h>`. Las constantes son las siguientes:

constante	significado
PTRACE_TRACEME	El proceso actual indica que será controlado por otro proceso
PTRACE_ATTACH	El proceso actual indica que controlará el proceso identificado por <i>pid</i> . La señal SIGSTOP se envía al proceso bajo control para suspender su ejecución
PTRACE_PEEKDATA	El contenido de la palabra situada en la dirección <i>addr</i> , en el segmento de datos del espacio de direccionamiento controlado, se devuelve en la variable direccionada por la variable <i>data</i>
PTRACE_PEEKTEXT	El contenido de la palabra situada en la dirección <i>addr</i> , en el segmento de código del espacio de direccionamiento del proceso controlado, se devuelve en la variable direccionada por la variable <i>data</i>
PTRACE_PEEKUSR	El contenido de la palabra situada en la dirección <i>addr</i> en la estructura <i>user</i> del proceso controlado (véase la sección 5.2) se devuelve en la variable direccionada por <i>data</i>
PTRACE_POKEDATA	El valor contenido en <i>data</i> se escribe en la palabra situada en la dirección <i>addr</i> , en el segmento de datos del espacio de direccionamiento del proceso controlado
PTRACE_POKETEXT	El valor contenido en <i>data</i> se escribe en la palabra situada en la dirección <i>addr</i> , en el segmento de código del espacio de direccionamiento del proceso controlado
PTRACE_POKEUSR	El valor contenido en <i>data</i> se escribe en la palabra situada en la dirección <i>addr</i> en la estructura <i>user</i> del proceso controlado
PTRACE_SYSCALL	La ejecución del proceso controlado se prosigue hasta la ejecución de la próxima llamada al sistema. El mandato <i>strace</i> utiliza esta opción para mostrar las llamadas al sistema ejecutadas por un proceso
PTRACE_CONT	La ejecución del proceso controlado se prosigue
PTRACE_KILL	La ejecución del proceso controlado se finaliza
PTRACE_SINGLESTEP	La ejecución del proceso controlado se prosigue para una sola instrucción máquina
PTRACE_DETACH	El proceso ya no está bajo control

En caso de error, *ptrace* devuelve el valor -1 y la variable *errno* puede tomar los valores siguientes:

error	significado
EFAULT	<i>addr</i> contiene una dirección inválida
EIO	<i>request</i> o <i>data</i> contiene un valor inválido
EPERM	El proceso que llama no posee los privilegios necesarios para controlar el proceso especificado por <i>pid</i> , o este último está ya bajo control
ESRCH	El proceso especificado por <i>pid</i> no existe

Los depuradores, como *gdb*, utilizan *ptrace* para ejecutar un programa instrucción por instrucción, y para permitir al usuario visualizar las variables del programa.

4.5 Clonado

La llamada al sistema *clone* crea un «clon» del proceso actual. Esta primitiva no se incluye en la biblioteca estándar y es necesario declararla explícitamente.

Antes de llamar a *clone*, debe asignarse un segmento de pila (llamando a la función *malloc*, por ejemplo), y los parámetros de la función a ejecutar deben apilarse en esta zona de memoria. Seguidamente, los registros del procesador deben inicializarse de la forma siguiente (en un procesador x86):

- *eax* debe contener el código de la llamada *clone*, representado por la constante `__NR_clone`;
- *ebx* debe contener una combinación de constantes presentadas más adelante;
- *ecx* debe contener el puntero de pila para el proceso hijo.

El programa *clone.S* siguiente, derivado de un archivo fuente de la biblioteca C del proyecto GNU, implementa la llamada de la primitiva *clone* en lenguaje ensamblador x86. El prototipo de la función *clone* proporcionado es el siguiente:

```
int clone (int (*fn)(), void *child_stack, int flags, int nargs, ...);
```

El parámetro *fn* es un puntero a la función a ejecutar por el proceso hijo. El parámetro *child_stack* es un puntero a la zona de memoria asignada para la pila del proceso hijo. El parámetro *flags* define las modalidades de «clonado». Finalmente, el parámetro *nargs* define el número de argumentos a pasar a la función apuntada por *fn* y va seguido por estos argumentos.

Varias constantes, definidas en el archivo de cabecera `<linux/sched.h>`, definen las modalidades del «clonado»:

constante	significado
CLONE_VM	El proceso hijo comparte el espacio de direccionamiento del proceso padre
CLONE_FS	El proceso hijo comparte los directorios raíz y actual del proceso padre
CLONE_FILES	El proceso hijo comparte los descriptores de archivos abiertos del proceso padre
CLONE_SIGHAND	El proceso hijo comparte los gestores de señales del proceso padre
CLONE_PID	El proceso hijo posee el mismo número que el proceso padre

Además, puede especificarse también la señal a enviar al proceso padre en la terminación del proceso hijo.

```
#define __ASSEMBLY__

#include <linux/linkage.h>
#include <asm/errno.h>
#include <asm/unistd.h>

.text
ENTRY(clone)
/* Verificación de los argumentos */
movl    $-EINVAL,%eax
movl    4(%esp),%ecx /*El puntero de función debe ser no nulo*/
testl   %ecx,%ecx
jz      syscall_error
movl    8(%esp),%ecx /*La dirección de pila debe ser no nula*/
testl   %ecx,%ecx
jz      syscall_error
movl    16(%esp),%edx /*El número de argumentos debe ser positivo*/
testl   %edx,%edx
js      syscall_error

/* Asignación de espacio en la pila y copia de argumentos */
movl    %edx,%eax
negl    %eax
leal    -4(%ecx,%eax,4),%ecx
jz      2f
1:      movl    16(%esp,%edx,4),%eax
        movl    %eax,0(%ecx,%edx,4)
        decl    %ecx
        jnz     1b
2:

/* Guardado del puntero a la función
   Se retira de la pila tras llamar al clon */
movl    4(%esp),%eax
movl    %eax,0(%ecx)

/* Llamada a la primitiva clone */
pushl   %ebx
movl    16(%esp),%ebx
movl    $__NR_clone,%eax
int     $0x80
popl    %ebx

/* Comprobación del código de retorno */
test    %eax,%eax
jl      syscall_error
jz      thread_start
```

```

ret

syscall_error:
/* Error, se devuelve -1 */
movl    $-1,%eax
ret

thread_start:
/* Código del proceso clon */
subl    %ebp,%ebp
call    %%ebx          /* Llamada a la función */
movl    $__NR_exit,%eax /* Finalización del proceso */
int     $0x80

```

El programa *EjemploClone.c* es un ejemplo simple (y poco útil) del uso de la función *clone*. Crea un proceso hijo que comparte el espacio de direccionamiento, los descriptores de archivos y la gestión de las señales con su proceso padre. El proceso hijo modifica una variable y cierra un archivo. El proceso padre muestra el contenido de la variable y comprueba si el archivo sigue abierto.

```

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

#include <linux/sched.h>
#include <linux/unistd.h>

#define STACKSIZE 16384

/*
 * Esta función es similar a thr_create() de la biblioteca
 * pthreads, pero es menos evolucionada
 */
int start_clone (void (*fn) (void *), void *data)
{
    long retval;
    void *newstack;

    /*
     * Asignación de la pila de la nueva tarea
     */
    newstack = (void **) malloc (STACKSIZE);
    if (!newstack)
        return -1;

    /*

```

```

    * Inicialización de la pila
    */
newstack = (void *) (STACKSIZE + (char *) newstack);

/*
 * Creación del clon
 */
retval = clone (fn, newstack, CLONE_VM | CLONE_FS | CLONE_FILES
               | CLONE_SIGHAND | SIGCHLD, 1, data);
if (retval < 0) {
    errno = -retval;
    retval = -1;
}
return retval;
}

int show_same_vm;

void cloned_process_starts_here (void *data)
{
    printf ("child:\t got argument %d as fd\n", (int) data);
    show_same_vm = 5;
    printf ("child:\t vm = %d\n", show_same_vm);
    close ((int) data);
}

int main (void)
{
    int fd, pid;

    fd = open ("/dev/null", O_RDONLY);
    if (fd < 0) {
        perror ("/dev/null");
        exit (1);
    }
    printf ("mother:\t fd = %d\n", fd);
    show_same_vm = 10;
    printf ("mother:\t vm = %d\n", show_same_vm);
    pid = start_clone (cloned_process_starts_here, (void *) fd);
    if (pid < 0) {
        perror ("start_clone");
        exit (1);
    }
    sleep (1);
    printf ("mother:\t vm = %d\n", show_same_vm);
    if (write (fd, "c", 1) < 0)
        printf ("mother:\t child closed our file descriptor\n");
    exit (0);
}

```

La ejecución del programa provoca la visualización siguiente:

```
mother: fd = 3
mother: vm = 10
child:  got argument 3 as fd
child:  vm = 5
mother: vm = 5
mother: child closed our file descriptor
```

Hay que observar que la primitiva *clone* es una llamada al sistema de muy bajo nivel, y es probablemente bastante compleja de usar directamente. Xavier Leroy está desarrollando actualmente una biblioteca de *threads* utilizando *clone* y está disponible en el URL <http://pauillac.inria.fr/~xleroy/linuxthreads/>.

5 Presentación general de la implementación

5.1 La tabla de procesos

5.1.1 Descriptor de proceso

Cada proceso se referencia por un descriptor. Este descriptor contiene los atributos del proceso, así como las informaciones que permiten gestionar el proceso.

La estructura `task_struct`, definida en el archivo `<linux/sched.h>`, caracteriza un proceso. Contiene los campos siguientes:

tipo	campo	descripción
volatile long	state	Estado del proceso
long	counter	Número de ciclos de reloj durante los que el proceso actual está autorizado a ejecutarse
long	priority	Prioridad del proceso
unsigned long	signal	Señales en espera (véase el capítulo 5, sección 5.1)
unsigned long	blocked	Señales ocultas (véase el capítulo 5, sección 5.1)
unsigned long	flags	Véase más adelante
int	errno	Código de error originado por las llamadas al sistema
long [8]	debugreg	Copia de los registros de hardware de depuración

<code>struct exec_domain *</code>	<code>exec_domain</code>	Ámbito de ejecución del proceso
<code>struct linux_binfmt *</code>	<code>binfmt</code>	Puntero a las operaciones relacionadas con el formato del programa ejecutado por el proceso
<code>struct task_struct *</code>	<code>next_task</code>	Puntero al proceso siguiente en la lista
<code>struct task_struct *</code>	<code>prev_task</code>	Puntero al proceso anterior en la lista
<code>struct task_struct *</code>	<code>next_run</code>	Puntero al proceso siguiente en la lista de procesos a punto
<code>struct task_struct *</code>	<code>prev_run</code>	Puntero al proceso anterior en la lista de procesos a punto
<code>unsigned long</code>	<code>saved_kernel_stack</code>	Puntero de pila usado en modo núcleo
<code>unsigned long</code>	<code>kernel_stack_page</code>	Dirección de la página de memoria que contiene la pila utilizada en modo núcleo
<code>int</code>	<code>exit_code</code>	Código de retorno a devolver al proceso padre: los 8 bits de menor peso (bits 0 a 7) contienen el número de la señal que ha causado la finalización del proceso, o bien los 8 bits siguientes (bits 8 a 15) contienen el código de retorno devuelto por el proceso tras llamar a la primitiva <code>_exit</code>
<code>int</code>	<code>exit_signal</code>	Número de la señal a enviar al proceso padre en la finalización
<code>unsigned long</code>	<code>personality</code>	Personalidad asociada al proceso
<code>int:1</code>	<code>dumpable</code>	Booleano que indica si debe crearse un archivo <i>core</i> en caso de error fatal
<code>int:1</code>	<code>did_exec</code>	Booleano que indica si el proceso ha usado <i>execve</i> para ejecutar un programa
<code>int</code>	<code>pid</code>	Número del proceso
<code>int</code>	<code>pgrp</code>	Número del grupo que contiene el proceso
<code>int</code>	<code>session</code>	Número de la sesión que contiene el proceso
<code>int</code>	<code>leader</code>	Booleano que indica si el proceso es el líder de su sesión
<code>int [NGROUPS]</code>	<code>groups</code>	Grupos asociados al proceso
<code>struct task_struct *</code>	<code>p_opptr</code>	Puntero al descriptor del proceso padre original
<code>struct task_struct *</code>	<code>p_pptr</code>	Puntero al descriptor del proceso padre
<code>struct task_struct *</code>	<code>p_cptr</code>	Puntero al descriptor del proceso hijo creado más recientemente
<code>struct task_struct *</code>	<code>p_ysptr</code>	Puntero al proceso «hermano» siguiente, que ha sido creado por el mismo proceso padre
<code>struct task_struct *</code>	<code>p_osptr</code>	Puntero al proceso «hermano» anterior
<code>struct wait_queue *</code>	<code>wait_chldexit</code>	Variable utilizada para esperar la finalización de un proceso hijo
<code>unsigned short</code>	<code>uid</code>	Identificador de usuario real asociado al proceso
<code>unsigned short</code>	<code>euid</code>	Identificador de usuario efectivo asociado al proceso
<code>unsigned short</code>	<code>suid</code>	Identificador de usuario guardado asociado al proceso
<code>unsigned short</code>	<code>fsuid</code>	Identificador de usuario asociado al proceso para los controles de acceso a los archivos

unsigned short	gid	Identificador de grupo real asociado al proceso
unsigned short	egid	Identificador de grupo efectivo asociado al proceso
unsigned short	sgid	Identificador de grupo guardado asociado al proceso
unsigned short	fsuid	Identificador de grupo asociado al proceso para los controles de acceso a los archivos
unsigned short	timeout	Lapso de espera máximo durante el que el proceso debe estar suspendido
unsigned short	policy	Política de coordinación asociada al proceso
unsigned short	rt_priority	Prioridad estática asociada al proceso
long	utime	Tiempo de procesador consumido en modo usuario
long	stime	Tiempo de procesador consumido en modo núcleo
long	cutime	Tiempo de procesador consumido por los procesos hijos en modo usuario
long	cstime	Tiempo de procesador consumido por los procesos hijos en modo núcleo
long	start_time	Fecha de creación del proceso
unsigned long	minflt	Número de excepciones de memoria tratadas por el proceso sin cargar páginas
unsigned long	majflt	Número de excepciones de memoria tratadas por el proceso cargando una página desde el disco
unsigned long	nswap	Número de páginas del proceso que se han guardado en la zona de <i>swap</i>
unsigned long	cminflt	Número de excepciones de memoria tratadas por los procesos hijos sin cargar páginas
unsigned long	cmajflt	Número de excepciones de memoria tratadas por los procesos hijos cargando una página desde el disco
unsigned long	cnsnap	Número de páginas de los procesos hijos guardadas en la zona de <i>swap</i>
int:1	swappable	Booleano que indica si el proceso puede guardarse en memoria secundaria
unsigned long	swap_cnt	Número de páginas de memoria a descartar
struct rlimit	rlim	Límites asociados al proceso
[RLIM_NLIMITS]		
unsigned short	used_math	Booleano que indica si el proceso ha utilizado el coprocesador matemático
char [16]	comm	Nombre del programa ejecutado por el proceso
int	link_count	Número de enlaces simbólicos explorados en la resolución de un nombre de archivo (véase el capítulo 6)
struct tty_struct *	tty	Puntero al descriptor del terminal asociado al proceso (véase el capítulo 9)
struct sem_undo *	semundo	Puntero a una lista de semáforos System V a liberar (véase el capítulo 11)
struct sem_queue *	semsleeping	Puntero a la cola de espera del semáforo System V en la que el proceso está suspendido

<code>struct desc_struct *</code>	<code>ldt</code>	Puntero al descriptor de la tabla de segmentos local a este proceso. Esta tabla se usa y modifica por el emulador Wine.
<code>struct thread_struct</code>	<code>tss</code>	Valor de los registros del procesador
<code>struct fs_struct *</code>	<code>fs</code>	Informaciones de control utilizadas en los accesos a los archivos (véase el capítulo 6, sección 5.2.5)
<code>struct files_struct *</code>	<code>files</code>	Puntero a los descriptors de archivos abiertos por el proceso (véase el capítulo 6, sección 5.2.5)
<code>struct mm_struct *</code>	<code>mm</code>	Informaciones de control utilizadas para la gestión de memoria (véase el capítulo 8, sección 5.4.1)
<code>struct signal_struct *</code>	<code>sig</code>	Puntero a los descriptors de acciones asociadas a las señales (véase el capítulo 5, sección 5.1)
<code>int</code>	<code>processor</code>	Identificador del procesador sobre el que se ejecuta el proceso
<code>int</code>	<code>last_processor</code>	Identificador del último procesador sobre el que se ha ejecutado el proceso

El estado del proceso (campo `state`) puede expresarse por varias constantes:

<i>constante</i>	<i>significado</i>
TASK_RUNNING	El proceso está a punto o en curso de ejecución
TASK_INTERRUPTIBLE	El proceso está suspendido pero puede ser despertado por una señal
TASK_UNINTERRUPTIBLE	El proceso está suspendido y no puede ser despertado por una señal
TASK_ZOMBIE	El proceso ha terminado su ejecución
TASK_STOPPED	El proceso ha sido suspendido por el usuario

También se definen varias constantes para el campo `flags`:

<i>constante</i>	<i>significado</i>
PF_PTRACED	El proceso está controlado por otro
PF_TRACESYS	El proceso está controlado por otro y debe ejecutarse hasta la próxima llamada al sistema
PF_FORKNOEXEC	El proceso no ha ejecutado la llamada al sistema <code>execve</code> para ejecutar otro programa
PF_SUPERPRIV	El proceso ha utilizado los privilegios del superusuario
PF_DUMPCORE	El proceso ha terminado produciendo un archivo <i>core</i>
PF_SIGNALED	El proceso ha terminado por la llegada de una señal
PF_STARTING	El proceso se está creando
PF_EXITING	El proceso está terminando
PF_USEDFPU	El proceso ha utilizado el coprocesador matemático durante su último lapso de tiempo (este estado es utilizado por el código de gestión de los multiprocesadores)

5.1.2 Organización de la tabla de procesos

Los descriptors de proceso los asigna dinámicamente el núcleo, llamando a la función `kmalloc`. La matriz `task`, definida en el archivo fuente `kernel/sched.c`, contiene punteros a estos descriptors. La matriz `current_set` contiene punteros a los descriptors de procesos en curso de ejecución en cada procesador. Una macroinstrucción, llamada `current`, corresponde a la dirección del descriptor del proceso ejecutado por el procesador actual.

La variable `init_task` contiene el descriptor del primer proceso creado en el arranque del sistema. Tras el arranque, este proceso sólo se ejecuta cuando ninguno más esté a punto, y su descriptor sirve para recuperar el inicio de la tabla de procesos.

Los descriptors de procesos se organizan en forma de una lista doblemente encadenada, por los campos `next_task` y `prev_task`. Los descriptors de los procesos a punto o en curso de ejecución se colocan en otra lista doblemente encadenada, mediante los campos `next_run` y `prev_run`.

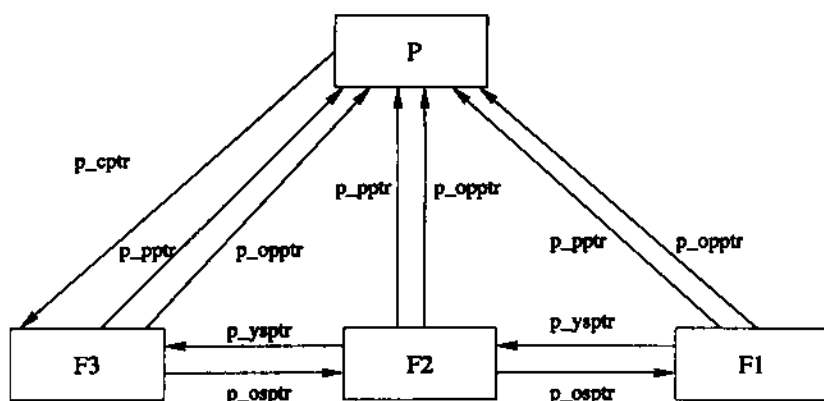


FIG. 4.3 – Relaciones entre descriptors de procesos

Los campos `p_opptr`, `p_pptr`, `p_cptr`, `p_ysptr` y `p_osptr` se utilizan para gestionar las filiaciones entre procesos. Cuando un proceso se duplica, llamando a la primitiva `fork`:

- los punteros `p_opptr` y `p_pptr` del descriptor del proceso hijo contienen la dirección del proceso padre;
- el puntero `p_osptr` del descriptor del proceso hijo toma el valor del puntero `p_cptr` del descriptor del proceso padre;

- el puntero `p_ysptr` del descriptor del proceso hijo se inicializa al valor nulo;
- el puntero `p_ysptr` del proceso «hermano» más reciente (referenciado por el puntero `p_cpctr` del descriptor del proceso padre) contiene la dirección del descriptor del nuevo proceso hijo;
- el puntero `p_cpctr` del descriptor del proceso padre contiene la dirección del descriptor del proceso hijo.

La figura 4.3 representa estos punteros en el caso de un proceso P que ha creado sucesivamente tres procesos hijos F1, F2 y F3.

5.1.3 Manipulación de la tabla de procesos

En el archivo `<linux/sched.h>` se definen varias macroinstrucciones que permiten gestionar las listas de descriptores de procesos:

macroinstrucción	significado
REMOVE_LINKS	Esta macroinstrucción suprime un descriptor de todas las listas a las que pertenece
SET_LINKS	Esta macroinstrucción inserta un descriptor en todas las listas: actualiza los punteros del descriptor para insertarlo en la lista de sus «hermanos»
for_each_task	Esta macroinstrucción permite explorar todos los descriptores de procesos, haciendo variar su parámetro

5.2 Registros del procesador

El contexto de un proceso incluye el contenido de los registros del procesador. En un cambio de contexto, o en la llamada a una primitiva del sistema, estos registros son guardados en memoria por el núcleo. La estructura `pt_regs`, definida en el archivo de cabecera `<asm/ptrace.h>`, contiene los campos siguientes para la arquitectura x86:

tipo	campo	descripción
long	ebx	Valor del registro ebx
long	ecx	Valor del registro ecx
long	edx	Valor del registro edx
long	esi	Valor del registro esi
long	edi	Valor del registro edi
long	ebp	Valor del registro ebp
long	eax	Valor del registro eax a restaurar al volver de la llamada al sistema
unsigned short	ds	Valor del registro de segmento ds (descriptor del segmento de datos)

unsigned short	es	Valor del registro de segmento es
unsigned short	fs	Valor del registro de segmento fs
unsigned short	gs	Valor del registro de segmento gs
long	orig_eax	Valor del registro eax en la llamada al sistema (eax contiene el número de la primitiva a ejecutar)
long	eip	Valor del registro eip (contador ordinal)
unsigned short	cs	Valor del registro de segmento cs (descriptor del segmento de código)
long	eflags	Valor de los indicadores del procesador
long	esp	Valor del registro esp (puntero de pila)
unsigned short	ss	Valor del registro de segmento ss (descriptor del segmento de pila)

La estructura `user`, definida en el archivo de cabecera `<asm/user.h>`, es utilizada por el núcleo cuando debe crearse un archivo *core*: esta estructura se coloca al principio del archivo para que un depurador pueda acceder al contexto del proceso cuando ha terminado. La llamada al sistema *ptrace* utiliza también esta estructura permitiendo a un proceso leer y modificar su contenido.

Los campos contenidos en la estructura `user` son los siguientes:

tipo	campo	descripción
struct pt_regs	regs	Valor de los registros del procesador
int	u_fpvalid	Booleano que indica si el proceso utiliza el coprocesador matemático
struct user_i387_struct	i387	Valor de los registros del coprocesador matemático
unsigned long int	u_tsize	Tamaño del segmento de código, expresado en páginas de memoria
unsigned long int	u_dsize	Tamaño del segmento de datos, expresado en páginas de memoria
unsigned long int	u_ssize	Tamaño del segmento de pila, expresado en páginas de memoria
unsigned long	start_code	Dirección virtual del inicio del segmento de código
unsigned long	start_stack	Dirección virtual del inicio de la pila
long int	signal	Número de la señal que ha causado el fin del proceso
int	reserved	No utilizado
struct pt_regs *	u_ar0	Dirección del valor de los registros en la estructura <code>user</code> (utilizado por <i>gdb</i>)
struct user_i387_struct *	u_fpstate	Dirección del valor de los registros del coprocesador en la estructura <code>user</code> (utilizado por <i>gdb</i>)
unsigned long	magic	Firma que identifica un archivo <i>core</i>
char[32]	u_comm	Nombre del programa ejecutado por el proceso
int[8]	u_debugreg	Valor de los registros de hardware de depuración

5.3 Sincronización de procesos

5.3.1 Principio

En un instante dado, un solo proceso puede ejecutarse en modo núcleo. Aunque es posible que se apliquen interrupciones de hardware y software a este proceso, Linux no provoca la coordinación si el proceso actual está activo en modo núcleo. Un proceso que se ejecuta en modo núcleo puede provocar sin embargo un cambio de proceso actual suspendiendo su ejecución. Esta suspensión voluntaria se debe generalmente a la espera de un evento, tal como el fin de una entrada/salida o la terminación de un proceso hijo.

Linux proporciona dos herramientas que permiten a los procesos sincronizarse en modo núcleo: las colas de espera (*wait queues*) y los semáforos. Las estructuras correspondientes se definen en el archivo de cabecera `<linux/wait.h>`.

5.3.2 Colas de espera

Una cola de espera es una lista encadenada y circular de descriptores. Cada descriptor contiene la dirección de un descriptor de proceso así como el puntero al elemento siguiente de la cola. La estructura `wait_queue` caracteriza los elementos de la cola de espera:

<i>tipo</i>	<i>campo</i>	<i>descripción</i>
<code>struct task_struct *</code>	<code>task</code>	Puntero al descriptor del proceso en espera
<code>struct wait_queue *</code>	<code>next</code>	Puntero al elemento siguiente de la cola

5.3.3 Semáforos

Los semáforos constituyen un mecanismo general de sincronización entre procesos.

Un semáforo no es realmente un mecanismo de comunicación, sino más bien un mecanismo de sincronización de procesos.

En informática, y más particularmente en el caso de los sistemas operativos, un semáforo se utiliza para controlar el acceso a un recurso.

El semáforo y las operaciones asociadas fueron definidos en 1965 por E. W. Dijkstra [Dijkstra 1965]. Un semáforo comprende un valor entero (un contador) y dos operaciones:

- **P** (del holandés *proberen*, probar): esta operación se utiliza cuando un proceso quiere entrar en una sección crítica. Esta operación realiza las etapas siguientes:
 1. probar el valor del contador del semáforo que controla el recurso;
 2. si este valor es positivo, el proceso puede servirse del recurso, y decrementa el valor en 1 para indicar que utiliza una unidad del recurso;
 3. si este valor es nulo, el proceso se duerme hasta que el valor sea de nuevo positivo. Cuando el proceso se despierta, vuelve a la etapa 1.
- **V** (del holandés *verhogen*, incrementar): esta operación es simétrica a la anterior, y se utiliza cuando un proceso abandona la sección crítica. El valor del contador del semáforo se incrementa y los procesos en espera son despertados.

Las operaciones **P** y **V** deben realizarse de manera atómica, es decir, que no deben ser interrumpidas.

En la literatura inglesa, las operaciones **P** y **V** se denominan frecuentemente **down** y **up**.

Los semáforos se utilizan en el ámbito de los sistemas informáticos principalmente para resolver dos problemas:

- **La exclusión mutua:** se trata de impedir a dos procesos el acceso a un mismo recurso en un mismo instante. Si esto se produjera, el recurso podría encontrarse en un estado inconsistente.
- **El problema de los productores/consumidores:** se trata de permitir la cooperación de dos procesos: uno produce informaciones que el otro utilizará. El semáforo se utiliza para prevenir al consumidor que los datos están a punto.

Estos problemas se resuelven mediante semáforos binarios, pero el contador del semáforo puede tomar otros valores positivos. Es el caso cuando varias unidades de un mismo recurso están disponibles; el contador toma entonces el valor del número de unidades accesibles simultáneamente.

Bajo Linux, la estructura `semaphore`, definida en el archivo `<asm/semaphore.h>`, contiene los campos siguientes:

tipo	campo	descripción
int	count	Contador del semáforo
int	waiting	Número de procesos en espera
struct wait_queue *	wait	Lista de procesos en espera en el semáforo

5.4 Los *timers*

Linux gestiona una lista de *timers* a fin de poder poner procesos en espera durante un lapso especificado. Estos *timers* se organizan en forma de una lista circular doblemente encadenada. Los *timers* a desencadenar en un futuro próximo se colocan al principio de la lista, mientras que los elementos a desencadenar en un futuro lejano se sitúan al final de la lista.

La estructura `timer_list`, definida en el archivo `<linux/timer.h>`, define el formato de los elementos de la lista:

tipo	campo	descripción
<code>struct timer_list *</code>	<code>next</code>	Puntero al elemento siguiente en la lista
<code>struct timer_list *</code>	<code>prev</code>	Puntero al elemento anterior de la lista
<code>unsigned long</code>	<code>expires</code>	Fecha de expiración del <i>timer</i>
<code>unsigned long</code>	<code>data</code>	Parámetro a transmitir a la función asociada
<code>void (*) (unsigned long)</code>	<code>function</code>	Dirección de la función a llamar cuando el <i>timer</i> expire

La variable `timer_head`, definida en el archivo fuente `kernel/sched.c`, contiene la dirección del primer elemento de la lista.

La fecha de expiración (campo `expires`) se expresa en número de ciclos de reloj desde el arranque del sistema. La variable global `jiffies` es mantenida por el núcleo (su valor se incrementa en cada interrupción del reloj) y contiene siempre el número de ciclos de reloj transcurridos desde el arranque.

5.5 Ámbitos de ejecución

El núcleo mantiene los ámbitos de ejecución soportados. Para ello, una lista encadenada simple contiene los descriptores de ámbitos. La estructura `exec_domain`, definida en el archivo `<linux/personality.h>`, describe los elementos de esta lista:

tipo	campo	descripción
<code>const char *</code>	<code>name</code>	Nombre del ámbito
<code>lcall7_func</code>	<code>handler</code>	Dirección de la función llamada si el proceso efectúa una llamada al sistema por un salto intrasegmento al segmento número 7 (método de llamada utilizado por System V y BSD Unix)
<code>unsigned char</code>	<code>pers_low</code>	Código de la personalidad asociada al ámbito
<code>unsigned char</code>	<code>pers_high</code>	Código de la personalidad asociada al ámbito

<code>unsigned long *</code>	<code>signal_map</code>	Tabla de correspondencias que permite convertir los números de señales proporcionados por el proceso que llama
<code>unsigned long *</code>	<code>signal_invmap</code>	Tabla de correspondencias que permite convertir los números de señales enviados al proceso que llama
<code>long *</code>	<code>use_count</code>	Número de procesos que utilizan este ámbito
<code>struct exec_domain *</code>	<code>next</code>	Puntero al descriptor del ámbito siguiente en la lista

La variable `exec_domains`, definida en el archivo fuente `kernel/exec_domain.c`, contiene la dirección del primer descriptor de la lista. En el arranque del sistema, la lista se inicializa y sólo contiene un elemento, llamado `default_exec.domain`, que define la personalidad de Linux.

5.6 Formatos de los archivos ejecutables

Linux soporta varios formatos de programas ejecutables, como los binarios `a.out` y `ELF`. A cada formato se le asocian varias funciones de manipulación de los programas ejecutables.

La estructura `linux_binfmt`, definida en el archivo de cabecera `<linux/binfmts.h>`, contiene los punteros que son necesarios:

- `int (*load_binary)(struct linux_binprm *binprm, struct pt_regs *regs)`

Esta función se llama para cargar un programa ejecutable en memoria en la ejecución de la llamada al sistema `execve`. El primer parámetro especifica el programa a ejecutar así como su contexto; el segundo parámetro indica el valor de los registros a posicionar.

- `int (*load_shlib)(int fd)`

Esta función se llama para cargar una biblioteca compartida en memoria. El parámetro `fd` especifica el descriptor de entradas/salidas asociado al archivo que contiene la biblioteca compartida.

- `int (*core_dump)(long signr, struct pt_regs *regs)`

Esta función se llama para crear un archivo *core*. El parámetro `signr` indica el número de la señal que ha causado la terminación del proceso, y `regs` especifica el contexto del proceso en la recepción de la señal.

La función de carga de un ejecutable, `load_binary`, utiliza su primer parámetro para conocer el contexto del proceso. La estructura `linux_binprm`, definida en el archivo `<linux/binfmts.h>`, se utiliza a este efecto y contiene los campos siguientes:

tipo	campo	descripción
char[128]	buf	Primeros 128 bytes del archivo ejecutable
unsigned long [MAX_ARG_PAGES]	page	Direcciones de las páginas de memoria que contienen los argumentos
int	sh_bang	Booleano que indica si se ha interpretado ya una línea del tipo <code>#!nombre_intérprete</code>
struct inode *	inode	Descriptor del i-nodo del archivo a ejecutar
int	e_uid	Identificador de usuario efectivo
int	e_gid	Identificador de grupo efectivo
int	argc	Número de argumentos
int	envc	Número de variables de entorno
char *	filename	Nombre del archivo a ejecutar
unsigned long	loader	Campo utilizado solamente en la arquitectura Alpha
int	dont_iput	Booleano que indica si debe llamarse a la función <code>iput</code> para liberar el descriptor del i-nodo

6 Presentación detallada de la implementación

6.1 Funciones internas

6.1.1 Sincronización de procesos

El archivo fuente `kernel/sched.c` contiene funciones de servicio que permiten la sincronización de procesos en modo núcleo. Estas funciones se utilizan en todas las partes del núcleo cuando un proceso debe suspenderse en espera de un evento, y cuando debe ser «despertado».

La función `add_to_runqueue` inserta un descriptor de proceso en la lista de procesos a punto. La función `del_from_runqueue` permite suprimir un descriptor de esta lista. Un descriptor de proceso puede colocarse al fin de la lista por la función `move_last_runqueue`.

La función `wake_up_process` despierta un proceso suspendido: pone su estado a `TASK_RUNNING` y lo inserta en la lista de procesos a punto.

Las colas de espera (*wait queues*) son manipuladas por las funciones `__add_wait_queue` y `__remove_wait_queue`, definidas en el archivo de cabecera `<linux/sched.h>`. Se añade un elemento a la cola de espera por `__add_wait_queue`, y se suprime de dicha cola por `__remove_wait_queue`. Las funciones `add_wait_queue` y `remove_wait_queue` llaman respectivamente a `__add_wait_queue` y `__remove_wait_queue` ocultando previamente todas las interrupciones.

La función `wake_up` permite despertar todos los procesos en espera de un evento en una cola de espera. Explora la cola, y llama a `wake_up_process` para cada proceso cuyo estado sea `TASK_UNINTERRUPTIBLE` o `TASK_INTERRUPTIBLE`. Un tratamiento similar se realiza por la función `wake_up_interruptible`, pero esta última sólo despierta los procesos cuyo estado es `TASK_INTERRUPTIBLE`.

La función `__sleep_on` suspende el proceso actual, y lo coloca en una cola de espera. Modifica el estado del proceso, guarda su descriptor en la cola por una llamada a `add_wait_queue`, y provoca un cambio de proceso actual por una llamada a la función `schedule`. Cuando el proceso se despierta, el resto de la función se ejecuta, y el descriptor del proceso se suprime de la cola de espera por una llamada a `remove_wait_queue`. Las funciones `interruptible_sleep_on` y `sleep_on` llaman a `__sleep_on` especificándole que ponga el estado del proceso actual respectivamente a `TASK_INTERRUPTIBLE` y `TASK_UNINTERRUPTIBLE`.

La función `__down` permite que el proceso actual suspenda en espera de un evento sobre un semáforo. Se añade el descriptor del proceso actual a la cola de espera del semáforo, su estado cambia a `TASK_UNINTERRUPTIBLE`, y se comprueba el contador del semáforo. Mientras este contador es inferior o igual a cero, la función `schedule` se llama para cambiar de proceso actual, y el estado del proceso se pone a `TASK_UNINTERRUPTIBLE`. Cuando el contador del semáforo pasa a estrictamente positivo, el estado del proceso actual se pone a `TASK_RUNNING` y el descriptor del proceso se suprime de la cola de espera del semáforo por una llamada a `remove_wait_queue`.

El uso de semáforos se efectúa mediante las funciones `down` y `up` definidas en el archivo de cabecera `<asm/semaphore.h>`.

6.1.2 Coordinación

El coordinador de procesos lo implementa la función `schedule`, situada en el archivo fuente `kernel/sched.c`.

La función `goodness` se utiliza para seleccionar un proceso. Devuelve un valor que indica hasta qué punto el proceso necesita el procesador. En el caso de un proceso en curso de ejecución en otro procesador, devuelve `-1000` a fin de indicar que el proceso no debe seleccionarse. En el caso de un proceso en tiempo real, devuelve la prioridad estática del proceso (campo `rt_priority`) aumentada en `1000`. En el caso de un proceso normal, devuelve el valor del campo `counter` que representa el número de ciclos de reloj durante el cual el proceso debe ejecutarse.

La función `schedule` implementa la coordinación propiamente dicha. Empieza por desplazar al proceso actual al final de la lista de procesos a punto, llamando a la función `move_last_runqueue`, si este proceso ha agotado todos sus ciclos.

Si el proceso actual se encuentra en el estado `TASK_INTERRUPTIBLE`, `schedule` comprueba si ha recibido una señal no oculta; si es éste el caso, el estado del proceso actual se coloca de nuevo a `TASK_RUNNING` a fin de despertarlo. Se efectúa seguidamente una exploración de la tabla de procesos: para cada proceso, la función `goodness` se llama a fin de determinar si el proceso debe elegirse. Al volver de esta búsqueda, el proceso elegido se convierte en el proceso actual: la función `get_mmu_context` se llama para restaurar el contexto de memoria del proceso, y el cambio de contexto se provoca mediante una llamada a `switch_to`.

El criterio utilizado por el coordinador es el valor del campo `counter` del descriptor de proceso. Este campo contiene el número de ciclos de reloj durante los cuales el proceso debe ejecutarse. Este campo es modificado por varias funciones:

- `update_process_times`: esta función es llamada periódicamente por `timer_bh`, que forma parte de la cola de tareas (*task queue*) `tq_timer`, activada por el gestor de interrupción de reloj. El campo `counter` es decrementado por `update_process_times`, que posiciona la variable `need_resched` a `1`, si el proceso actual ha agotado su lapso.
- `schedule`: cuando el campo `counter` es nulo para todos los procesos, el coordinador efectúa un bucle sobre todos los descriptores de la tabla de procesos a fin de reinicializar el campo `counter`. El campo `priority` se utiliza entonces como valor de base.
- `add_to_runqueue`: esta función, que añade un proceso a la lista de procesos a punto, comprueba el campo `counter`: si es superior al del proceso actual, la variable `need_resched` se pone a `1`, a fin de provocar un cambio de proceso actual.

6.1.3 Los *timers*

El núcleo, de manera interna, implementa *timers* a fin de provocar esperas cronometradas. Se definen varias funciones de gestión en el archivo fuente *kernel/sched.c*.

La función `add_timer` añade un *timer* a la lista: explora la lista de *timers* registrados para insertar su argumento de modo que la lista quede ordenada. De este modo, los primeros elementos de la lista corresponden a los *timers* a desencadenar en primer lugar.

La función `del_timer` suprime un *timer* de la lista: modifica simplemente los encañamientos para suprimir el elemento de la lista.

La función `run_timer_list` es llamada por la lista de tareas (*task queue*) `tq_timer`. Explora la lista de *timers* y ejecuta cada *timer* expirado, es decir, aquel cuyo campo `expires` es inferior o igual al valor de la variable global `jiffies`, llamando a la función asociada (campo `function`) con el parámetro especificado (campo `data`). Cada uno de los *timers* ejecutados se suprime de la lista.

6.1.4 Espera con demora

Un proceso en modo núcleo puede suspender su ejecución especificando una demora destinada a esperar un evento. Para ello, un proceso utiliza el campo `timeout` de su descriptor: pone el valor de este campo a la demora deseada, expresada en ciclos de reloj, modifica su estado a `TASK_INTERRUPTIBLE` y llama a la función `schedule` para provocar una ordenación como sigue:

```
current->timeout = delay;
current->state = TASK_INTERRUPTIBLE;
schedule ();
```

Al cambiar de proceso, `schedule` verifica si el campo `timeout` del proceso es no nulo, es decir, si el proceso ha especificado un tiempo de espera. Si es el caso, `schedule` activa un *timer*, por una llamada a las funciones `init_timer` y `add_timer`. Este *timer* llamará a la función `process_timeout`, pasándole la dirección del descriptor de proceso como parámetro, tras la demora especificada.

Cuando se llama a la función `process_timeout`, pone a cero el campo `timeout` del descriptor correspondiente al proceso, y despierta al proceso llamando a la función `wake_up_process`.

Si el proceso se despierta antes de la expiración de la demora, por la llegada del evento esperado o la recepción de una señal, sigue la ejecución de la función `schedule` que suprime el *timer* correspondiente llamando a `del_timer`.

6.2 Implementación de las llamadas al sistema

6.2.1 Creación de procesos

La implementación de la llamada al sistema *fork* se sitúa en el archivo fuente *kernel/fork.c*. Este archivo define varias variables globales:

- *nr_tasks*: número de procesos existentes en el sistema;
- *nr_running*: número de procesos en estado a punto existentes en el sistema;
- *total_forks*: número total de llamadas al sistema *fork* ejecutadas desde el arranque del sistema;
- *last_pid*: número atribuido al último proceso creado.

También se definen varias funciones internas. La función *find_empty_process* busca un puesto libre en la tabla *task*. Verifica que el usuario que ejecuta el proceso actual está autorizado para crear un nuevo proceso, es decir, que el número máximo de procesos no se ha alcanzado y que el usuario no lo ha sobrepasado. Efectúa seguidamente un bucle de búsqueda de un puesto libre en la tabla *task* y devuelve su índice.

La función *get_pid* se llama para obtener un nuevo número de proceso. Si el nuevo proceso comparte el identificador del proceso padre (mediante la opción *CLONE_PID* de la primitiva *clone*), se devuelve este identificador; si no, *get_pid* incrementa el valor de *last_pid* y efectúa una exploración de la tabla *task* para verificar que el número no se usa ya como identificador de proceso, de grupo de procesos o de sesión. Si el número está en uso, *last_pid* se incrementa de nuevo y se reemprende la búsqueda.

Las funciones *dup_mmap* y *copy_mm* duplican el espacio de direccionamiento del proceso actual. Se detallan en el capítulo 8, sección 6.7.

Las funciones *copy_fs* y *copy_files* duplican los descriptores de archivos abiertos por el proceso actual. Se detallan en el capítulo 6, sección 6.3.10.

La función *copy_sighand* duplica los descriptores de señales del proceso actual. Se detalla en el capítulo 5, sección 5.1.1.

La función *copy_thread*, definida en el archivo fuente *arch/i386/kernel/process.c*, inicializa los valores de los registros del procesador que se asocian al proceso creado. Modifica el contenido del campo *tss* del descriptor de proceso.

La función `do_fork` procede a la duplicación del proceso actual, en la ejecución de las primitivas *clone* y *fork*. Asigna un nuevo descriptor de proceso llamando a `kma-llloc`, asigna una página de memoria para la pila del proceso en modo núcleo por una llamada a `alloc_kernel_stack`, y obtiene una entrada en la tabla de procesos por una llamada a `find_empty_process`. El descriptor del nuevo proceso se inicializa seguidamente copiando las informaciones que caracterizan el proceso actual, cuyo descriptor es apuntado por la variable `current`, y modificando los atributos diferentes entre los procesos padre e hijo. El nuevo descriptor se inserta en la tabla de procesos, y las funciones `copy_files`, `copy_fs`, `copy_sighand`, `copy_mm` y `copy_thread` se llaman para inicializar el contexto del proceso creado. Finalmente, se devuelve el número atribuido al nuevo proceso.

Para la arquitectura x86, la implementación de las primitivas *fork* y *clone* está contenida en el archivo fuente *arch/i386/kernel/process.c*. Las funciones `sys_fork` y `sys_clone` llaman ambas a `do_fork` pasándole parámetros diferentes.

Hay que observar que `sys_fork` y `sys_clone` se llaman sin parámetros. Sin embargo, acceden al valor de los registros del procesador en su llamada. Estos registros se guardan en la pila por la macroinstrucción `SAVE_ALL` al activar la llamada al sistema (función `system_call` definida en el archivo fuente *arch/i386/kernel/entry.S*), y `sys_fork` y `sys_clone` acceden a los valores de los registros mediante un parámetro ficticio de estructura `pt_regs`.

6.2.2 Terminación de proceso

El archivo fuente *kernel/exit.c* contiene la implementación de las llamadas al sistema vinculadas a la terminación de proceso: *exit* y *wait4*.

La función `notify_parent` previene a un proceso sobre la terminación de uno de sus hijos. Se envía una señal al padre del proceso actual, utilizando el campo `exit_signal`, por una llamada a `send_sig`, y luego utiliza la función `wake_up_interruptible` para despertar al proceso padre si está en espera de la terminación de uno de sus hijos, es decir, si ha sido puesto en espera en la cola `wait_chldexit` por una llamada a la primitiva *wait4*.

Las funciones `close_files`, `__exit_files` y `__exit_fs` liberan los descriptors de archivos abiertos de un proceso. Su funcionamiento se detalla en el capítulo 6, sección 6.3.10.

La función `__exit_sighand` libera los descriptors de gestores de señales de un proceso. Su funcionamiento se detalla en el capítulo 5, sección 5.1.1.

La función `__exit_mm` libera los descriptores de zonas de memoria asociadas a un proceso. Su funcionamiento se detalla en el capítulo 8, sección 6.7.

La función `exit_notify` se llama al terminar un proceso, a fin de advertir a los procesos emparentados. Llama primero a `forget_original_parent`, que explora la tabla de procesos, modificando los punteros `p_opptr` para hacerlos apuntar al proceso `init` si apuntaban al proceso en curso de finalización. Si el proceso actual es *leader* de un grupo que posee procesos suspendidos (en estado `TASK_STOPPED`), las señales `SIGHUP` y `SIGCONT` se envían al proceso del grupo llamando a la función `kill_pg`. El proceso padre es advertido seguidamente de la terminación llamando a `notify_parent`. Finalmente, se efectúa un bucle de exploración de procesos hijos: cada uno de los procesos hijos se enlaza al proceso `init`, y se envía una señal a `init` por la llamada de `notify_parent` si el proceso hijo está en el estado `TASK_ZOMBIE`.

La función `do_exit` procede a la terminación del proceso actual. Libera los descriptores asociados al proceso llamando a las funciones `__exit_mm`, `__exit_files`, `__exit_fs` y `__exit_sighand`. Luego pone el estado del proceso actual a `TASK_ZOMBIE`, guarda el código de retorno del proceso, y llama a `exit_notify` para advertir al proceso padre de la finalización de uno de sus hijos. Finalmente, se llama a la función `schedule` para proceder a una coordinación.

La primitiva `exit` viene implementada por la función `sys_exit`. Esta función llama simplemente a `do_exit` transmitiéndole el código de retorno, proporcionado por el proceso y multiplicado por 256. De este modo, el código de retorno se coloca en los bits 8 a 15 del campo `exit_code` del descriptor de proceso actual.

La función `sys_wait4` provoca la espera de la terminación de un proceso hijo. Verifica en primer lugar la validez de sus argumentos, y registra el descriptor del proceso actual en la cola de espera referenciada por el campo `wait_chldexit`. Seguidamente, efectúa una exploración de la lista de procesos hijos del proceso actual, comprobando si cada proceso hijo corresponde al parámetro `pid`. Para cada proceso hijo correspondiente al criterio especificado en la llamada al sistema, se tienen en cuenta dos casos:

1. El proceso hijo está en estado `TASK_STOPPED`, `WUNTRACED` está incluido entre las opciones de espera, y el proceso padre aún no ha sido advertido de su cambio de estado.
2. El proceso hijo está en el estado `TASK_ZOMBIE`.

Tras esta búsqueda, si el proceso actual posee procesos hijos de los cuales ninguno ha terminado aún su ejecución, y si las opciones de espera no incluyen `WNOHANG`, el proceso actual se suspende en espera de la terminación de un hijo modificando su estado a `TASK_INTERRUPTIBLE`, y llamando a la función `schedule` para provocar una

coordinación; seguidamente, vuelve a empezar el tratamiento cuando el proceso actual es despertado por la finalización de un proceso hijo.

Si no existe ningún proceso hijo, el código de retorno de la función se pone a `ECHILD`. Finalmente, el proceso actual se suprime de la cola de espera referenciada por el campo `wait_chldexit`.

Las otras llamadas al sistema de espera de terminación de proceso hijo se implementan llamando a `wait4`:

- `wait (&status)` corresponde a `wait4 (-1, &status, 0, NULL)`
- `wait3 (&status, options, &rusage)` corresponde a `wait4 (-1, &status, options, &rusage)`
- `waitpid (pid, &status, options)` corresponde a `wait4 (pid, &status, options, NULL)`

6.2.3 Obtención de los atributos

El archivo fuente *kernel/sched.c* contiene la implementación de las llamadas al sistema que permiten al proceso actual obtener sus atributos: *getpid*, *getppid*, *getuid*, *geteuid*, *getgid* y *getegid*. Las funciones de tratamiento de estas primitivas son muy simples, porque se limitan a devolver un campo del descriptor del proceso actual, apuntado por la variable `current`.

La llamada al sistema *getgroups* se implementa en el archivo fuente *kernel/sys.c*. La función `sys_getgroups` comprueba la validez de sus argumentos, cuenta el número de grupos asociados al proceso actual, y los copia desde el descriptor del proceso actual a la memoria intermedia proporcionada por quien llama. La función devuelve el número de grupos.

6.2.4 Modificación de atributos

El archivo fuente *kernel/sys.c* contiene la implementación de las llamadas al sistema que permiten al proceso actual modificar sus atributos: *setregid*, *setgid*, *setreuid*, *setuid*, *setfsuid*, *setfsgid* y *setgroups*. Las funciones que implementan estas primitivas son muy simples: verifican que el proceso actual está autorizado a modificar sus atributos, y cambian el valor de los campos correspondientes en su descriptor.

6.2.5 Grupos de procesos y sesiones

Las llamadas al sistema que manipulan los grupos de procesos y las sesiones se implementan en el archivo fuente *kernel/sys.c*.

La función *sys_getpgrp* devuelve el campo *pgrp* del descriptor del proceso actual, apuntado por la variable *current*. La función *sys_getpgid* efectúa una búsqueda del descriptor del proceso cuyo número se le pasa como parámetro, y devuelve el campo *pgrp* de este descriptor. La función *get_sid* efectúa un tratamiento similar, pero devuelve el campo *session* del descriptor encontrado.

La función *sys_setpgid* implementa la primitiva *setpgid*. Efectúa una búsqueda del proceso a modificar en la tabla *task*. Una vez encontrado el descriptor correspondiente, procede a varios controles:

- si un proceso modifica el grupo de uno de sus procesos hijos, ambos procesos deben pertenecer a la misma sesión y el proceso hijo no debe haber utilizado la primitiva *execve* para ejecutar un nuevo programa; si no, el proceso a modificar debe ser el proceso actual;
- el proceso a modificar no debe ser el *leader* de una sesión;
- el identificador de grupo a poner debe ser el identificador del proceso actual, o debe estar incluido en la sesión a la que pertenece el proceso actual.

Tras estas verificaciones, el grupo al que pertenece el proceso actual se modifica, cambiando el valor del campo *pgrp* de su descriptor.

La llamada al sistema *setsid* se implementa por la función *sys_setsid*. Esta función explora primero la tabla *task* para verificar que el proceso actual no es un *leader* de grupo de procesos. Una vez hecha esta verificación, modifica los identificadores de grupo de procesos y de sesión del proceso actual (campos *pgrp* y *session*), indica que el proceso actual es el *leader* de su sesión poniendo el campo *leader* a 1, disocia el terminal de control del proceso modificando los campos *tty* y *tty_old_pgrp*, y devuelve el número de la sesión creada.

6.2.6 Control de procesos

La llamada al sistema *ptrace* se implementa en el archivo fuente *arch/i386/kernel/ptrace.c*, para la arquitectura x86, porque esta primitiva es dependiente del procesador.

La función `get_task` devuelve el descriptor de proceso correspondiente al número de proceso especificado. `get_stack_long` devuelve una palabra situada en la pila núcleo del proceso especificado, y `put_stack_long` modifica el contenido de una palabra en la pila núcleo.

Varias funciones manipulan el espacio de direccionamiento de un proceso: `get_long`, `put_long`, `find_extend_vma`, `read_long` y `write_long`. En razón de sus interacciones con los mecanismos de gestión de la memoria, se detallan en el capítulo 8.

La función `sys_ptrace` implementa la llamada al sistema *ptrace*. Ésta trata primero la petición `PTRACE_TRACEME`: si el indicador `PF_PTRACED` está ya puesto en el descriptor del proceso actual, se devuelve el error `EPERM`; si no, se activa el indicador. Tras algunos controles de validez, se trata la petición `PTRACE_ATTACH`: si el proceso especificado está ya bajo control, se devuelve el error `EPERM`, si no se activa el indicador `PF_PTRACED` en el campo `flags` de su descriptor, el proceso se vincula a quien lo controla modificando el puntero `p_pptr` y utilizando las macroinstrucciones `REMOVE_LINKS` y `SET_LINKS`; finalmente, se envía la señal `SIGSTOP` al proceso, llamando a la función `send_sig`, a fin de suspender la ejecución del proceso.

Antes de tratar las otras peticiones, `sys_ptrace` efectúa varios controles: el proceso especificado debe estar bajo control (el indicador `PF_PTRACED` del campo `flags` de su descriptor debe estar activado), debe estar en el estado `TASK_STOPPED`, y debe estar vinculado al proceso actual (el puntero `p_pptr` debe contener la dirección del descriptor del proceso actual, contenido en la variable `current`). Tras estos controles, se tratan las diferentes peticiones:

- `PTRACE_PEEKTEXT`, `PTRACE_PEEKDATA`: la palabra especificada se lee en el espacio de direccionamiento del proceso por `read_long`, y se devuelve al proceso que llama.
- `PTRACE_PEEKUSR`: la palabra especificada se lee en la estructura `user` del proceso por `get_stack_long`, y se devuelve al proceso que llama.
- `PTRACE_POKETEXT`, `PTRACE_POKEDATA`: la palabra especificada se escribe en el espacio de direccionamiento del proceso por `write_long`.
- `PTRACE_POKEUSR`: la palabra especificada se escribe en la estructura `user` del proceso por `put_stack_long`, y se devuelve al proceso que llama.
- `PTRACE_SYSCALL`, `PTRACE_CONT`: el campo `flag` del descriptor del proceso se modifica a fin de poner o borrar el indicador `PF_TRACESYS`, el proceso se des-

pierta por una llamada a `wake_up_process`. Finalmente, el bit de seguimiento contenido en los indicadores del procesador se pone a cero.

- **PTRACE_KILL**: el proceso se despierta por una llamada a `wake_up_process`, y el campo `exit_code` se rellena para indicar que el proceso debe recibir la señal `SIGKILL`. Finalmente, el bit de seguimiento contenido en los indicadores del procesador se pone a cero.
- **PTRACE_SINGLESTEP**: el bit de seguimiento contenido en los indicadores del procesador se activa, y el proceso se despierta con una llamada a `wake_up_process`.
- **PTRACE_DETACH**: el campo `flag` del descriptor del proceso se modifica para borrar los indicadores `PF_TRACED` y `PF_TRACESYS`, y el proceso se despierta por una llamada a `wake_up_process`. El proceso se vincula seguidamente a su padre original, modificando el puntero `p_pptr`, y llamando a las macroinstrucciones `REMOVE_LINKS` y `SET_LINKS`. Finalmente, el bit de seguimiento contenido en los indicadores del procesador se pone a cero.

Capítulo 5

Señales

Primitivas detalladas

alarm, getitimer, kill, killpg, pause, psignal, raise, setitimer, sigaction, sigaddset, sigblock, sigdelset, sigemptyset, sigfillset, siggetmask, siginterrupt, sigismember, sigmask, sigpause, sigpending, sigprocmask, sigsetmask, sigsuspend, strsignal

1 Conceptos básicos

1.1 Introducción

La gestión de señales es un mecanismo existente desde las primeras versiones de Unix. Permite a los procesos reaccionar a los eventos provocados por ellos mismos o por otros procesos. Este mecanismo puede asimilarse a la gestión de interrupciones lógicas.

1.2 Definición de señales

Cada señal corresponde a un evento particular. Una señal se representa en el sistema por un nombre de la forma SIGXXX. El número total de señales existentes en el sistema se indica por la constante NSIG definida en el archivo de cabecera <signal.h>.

Una señal puede generarse de varias maneras, por ejemplo:

- Es el resultado de una excepción de hardware. Cuando un proceso escribe en una zona de memoria no asignada, se da un acceso inválido a una página de memoria

que provoca una excepción. Esta excepción es atrapada por el núcleo que genera la señal SIGSEGV hacia el proceso transgresor.

- Es el resultado de la pulsación de `Ctrl-C` por parte del usuario del terminal. Esto genera la señal SIGINT cuya acción predeterminada es terminar el proceso de primer plano de la sesión.
- Es el resultado de la llamada al sistema *kill* o del mandato del mismo nombre, que permite enviar una señal a un proceso determinado (véase la sección 2.1).
- Es el resultado de un evento gestionado por el núcleo. Por ejemplo, la señal SIGALRM es emitida por el sistema cuando expira una alarma hacia el proceso que la ha solicitado (véase la sección 4.2.1).

Una señal generada no se recibe de manera síncrona en el proceso destinatario. Las señales son atendidas por un proceso cuando éste pasa del modo sistema al modo usuario. Puede existir, pues, una demora importante entre la emisión y la recepción de una señal.

Cada señal produce una acción predeterminada para el proceso que la recibe. Hay tres acciones predeterminadas distintas:

- ignorar la señal;
- terminar el programa;
- terminar el programa y generar un archivo *core*.

En el último caso, el sistema fabrica una imagen de la memoria del proceso y la guarda en un archivo llamado *core*. Este archivo se utiliza en la depuración del programa con herramientas como *gdb*.

La acción predeterminada asociada a una señal puede ser modificada por el proceso receptor de la señal. El proceso utiliza entonces una función de desviación de la señal llamada habitualmente *handler*.

Existen dos señales cuyo comportamiento no puede modificarse. Son SIGKILL y SIGSTOP. Esta característica permite al superusuario poder interrumpir o suspender la ejecución de todo proceso.

Dos señales no corresponden a ningún evento en particular: se trata de SIGUSR1 y SIGUSR2, que están a disposición del programador.

1.3 Lista de señales

Esta sección presenta la lista de las diferentes señales definidas bajo Linux. Las señales se clasifican en función de las acciones que las generan. Para cada señal, las tablas describen su causa y la acción predeterminada que provocan. Además, un asterisco indica que la señal es conforme a POSIX. 1. El valor numérico de las señales no se indica porque es, en algunos casos, dependiente de la arquitectura sobre la que se instala el sistema.

La primera tabla indica las señales de terminación o interrupción de procesos.

señal	causa	acción predeterminada
SIGHUP *	Terminación del proceso líder de sesión o desconexión de un módem	Terminación
SIGINT *	Emisión del carácter de interrupción del terminal (Ctrl-C)	Terminación
SIGQUIT *	Emisión del carácter de terminación del terminal (Ctrl-\)	Terminación con <i>core</i>
SIGABRT *, SIGIOT	Terminación anormal (<i>abort</i>)	Terminación con <i>core</i>
SIGKILL *	Señal de terminación sin desviación posible	Terminación
SIGTERM *	Señal emitida en principio por el mandato <i>kill</i>	Terminación

La tabla siguiente contiene las señales provocadas por una excepción de hardware.

señal	causa	acción predeterminada
SIGILL *	Instrucción ilegal	Terminación
SIGTRAP	Punto de parada en un programa (llamada al sistema <i>ptrace</i>)	Terminación con <i>core</i>
SIGBUS	Error de bus	Terminación
SIGFPE*	Error aritmético	Terminación
SIGSEGV*	Direccionamiento de memoria no válido	Terminación con <i>core</i>
SIGSTKFLT	Desbordamiento de pila del coprocesador matemático (únicamente en arquitecturas Intel)	Terminación

Las señales de usuario se dejan a disposición del programador.

señal	causa	acción predeterminada
SIGUSR1*	Definida por el programador	Terminación
SIGUSR2*	Definida por el programador	Terminación

La señal siguiente puede ser generada por el sistema en el cierre de una tubería (véase el capítulo 10).

señal	causa	acción predeterminada
SIGPIPE*	Tubería sin lector	Terminación

Las señales relacionadas con el control de actividad son peculiares, porque permiten suspender o proseguir la ejecución de los procesos.

señal	causa	acción predeterminada
SIGCHLD* , SIGCLD	Terminación de un hijo	Ninguna
SIGCONT*	El proceso se lleva a primer o segundo plano	Prosigue la ejecución del proceso si estaba parado
SIGSTOP*	Suspensión del proceso	Suspensión (no modificable)
SIGTSTP*	Emisión al terminal del carácter de suspensión (Ctrl-Z)	Suspensión
SIGTTIN*	Lectura del terminal para un proceso en segundo plano	Suspensión
SIGTTOU*	Escritura en el terminal para un proceso en segundo plano	Suspensión

Las señales siguientes están relacionadas con los recursos modificables del proceso (véase el capítulo 4, sección 2.7).

señal	causa	acción predeterminada
SIGXCPU	Límite de tiempo de CPU sobrepasado	Terminación
SIGXFSZ	Límite de tamaño de archivo sobrepasado	Terminación

Las señales siguientes están relacionadas con la gestión de las alarmas. Se generan después que el proceso ha utilizado la llamada al sistema *setitimer* o *alarm* en el caso de **SIGALRM** (véase la sección 4.2.1, y la sección 4.2.2).

señal	causa	acción predeterminada
SIGALRM*	Fin del timer ITIMER_REAL	Terminación
SIGVTALRM	Fin del timer ITIMER_VIRTUAL	Terminación
SIGPROF	Fin del timer ITIMER_PROF	Terminación

Las señales para la gestión de entradas/salidas asíncronas son las siguientes:

señal	causa	acción predeterminada
SIGWINCH	Cambio de tamaño de una ventana (utilizado por X11)	Ninguna
SIGIO , SIGPOLL	Datos disponibles para una entrada/salida	Terminación
SIGURG	Datos urgentes para los <i>sockets</i>	Ninguna

Una señal de alerta indica un fallo del sistema de alimentación. Esta señal únicamente se emite evidentemente si existe un sistema de alimentación ininterrumpida (SAI).

señal	causa	acción predeterminada
SIGPWR , SIGINFO	Fallo de alimentación	Terminación

1.4 Visualización de las señales

Existe una tabla que contiene la cadena de caracteres asociada a cada señal. Se define en el archivo `<signal.h>`:

```
extern const char * const sys_siglist[];
```

Esta tabla no se utiliza directamente, sino por medio de dos funciones de la biblioteca C: *strsignal* y *psignal*.

El prototipo de estas funciones es el siguiente:

```
#include <string.h>
#include <signal.h>

char *strsignal (int sig);

void psignal (int sig, const char *s);
```

La primera función toma como parámetro el valor numérico de la señal y devuelve la cadena de caracteres asociada que se almacena en *sys_siglist*.

La segunda muestra en la salida de error un mensaje compuesto de la cadena especificada por el parámetro *s* y del mensaje asociado a la señal de la cual *sig* contiene el número. Su principio es similar al de la función *perror*.

2 Llamadas al sistema de base

La semántica de las llamadas al sistema que gestionan las señales en Linux es la de la norma POSIX. Sin embargo, la gestión de las señales existía en los sistemas Unix mucho antes de la definición de una norma, por lo que existe un cierto número de llamadas al sistema no POSIX definidos a este efecto. Estas llamadas al sistema aparecieron en las dos clases de sistemas Unix, System V y BSD. En ocasiones tienen los mismos nombres pero semánticas diferentes. A fin de ser lo más completo posible y permitir la portabilidad de un mayor número de programas, Linux implementa las diferentes semánticas. En principio, se utiliza la semántica de System V pero la inclusión del archivo de cabecera *<bsd/signal.h>* en lugar de *<signal.h>* en el momento de la compilación de los programas y eventualmente la inclusión de la biblioteca *libbsd.a* en el momento del enlazado les permite respetar la semántica de los sistemas BSD.

En la descripción de las llamadas al sistema, se presentan las dos semánticas cuando es necesario.

2.1 Emisión de una señal

La llamada al sistema *kill* se utiliza para enviar una señal a un proceso o a un grupo de procesos.

```
#include <signal.h>

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

El segundo argumento de la llamada indica la señal a emitir. El primer argumento representa la identidad del destinatario. Son posibles varios casos:

- si *pid* es positivo, se envía la señal *sig* al proceso identificado por *pid*;
- si *pid* es nulo, la señal *sig* se envía a los procesos del grupo de procesos emisor;
- si *pid* es igual a *-1*, la señal *sig* se envía a todos los procesos, salvo al primero (el programa *init*);
- si *pid* es inferior a *-1*, la señal *sig* se envía al grupo de procesos identificado por el valor *-pid*.

Si la señal enviada es nula, la llamada no genera ninguna señal, pero controla la existencia del proceso destinatario (atención: si el proceso está en el estado ZOMBIE, se considera como existente.).

Un proceso puede enviar una señal a otro si su identificador real o efectivo es igual al identificador real o efectivo del otro proceso.

Si la llamada se desarrolla correctamente, se devuelve el valor cero; si no, se produce un error, se devuelve *-1* y la variable *errno* contiene uno de los valores siguientes:

<i>error</i>	<i>significado</i>
EINVAL	La señal especificada no es válida
ESRCH	El proceso o grupo de procesos destino no existe
EPERM	El valor del identificador de usuario efectivo del proceso que llama es diferente del proceso o procesos destino

Un proceso perteneciente al superusuario tiene el derecho de enviar una señal a todos los demás procesos (salvo *init*: a fin de proteger al sistema, ningún proceso puede enviar una señal que provocaría la finalización del proceso *init*).

Para enviar una señal a un grupo de procesos, existe también la llamada *killpg*. Su prototipo es el siguiente:


```
#include <signal.h>

int killpg (int pgrp, int sig);
```

El primer argumento de la llamada representa el grupo hacia el cual se emite la señal. Si su valor es nulo, entonces la señal se envía a los procesos del grupo del proceso emisor. Si no, el funcionamiento es similar al de *kill*.

Existe también una función ANSI-C que permite a un proceso enviar una señal, es la función *raise*. Su prototipo es el siguiente:

```
#include <signal.h>

int raise (int sig);
```

Esta función es equivalente a `kill (getpid() , sig)`. En caso de error, se devuelve un valor no nulo.

2.2 Desviación de una señal

El proceso que desea modificar el comportamiento predeterminado de una señal, la desvía hacia una nueva función.

La llamada al sistema *signal* permite realizar esta acción. Su prototipo es el siguiente:

```
#include <signal.h>

void (*signal (int signum, void (*handler) (int))) (int);
```

El primer argumento es la señal afectada por la operación. El segundo argumento es un puntero a la función de desvío o bien una de las dos constantes siguientes: `SIG_IGN` para ignorar la señal, `SIG_DFL` para reposicionar el comportamiento predeterminado. La función de desviación toma como parámetro un entero correspondiente a la señal que la ha activado y no devuelve ningún argumento.

Al llegar una señal, el sistema interrumpe la ejecución normal del proceso para ejecutar la función de desviación. Pasa el valor de la señal emitida a esta función.

El valor de retorno de la llamada al sistema *signal* es un puntero a la función de desviación presente ante la llamada, o bien la constante `SIG_ERR` si se produce un error.

El programa siguiente presenta la manera de usar la llamada *signal*.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
void    desvia (int sig);
int     main ()
{
    if (signal (SIGUSR1, desvia) == SIG_ERR {
        perror ("señal no desviable");
        exit (1);
    }
    for (;;) ;

    return 0;
}

void    desvia (int sig)
{
    printf ("Señal recibida: %s\n", strsignal (sig));
}
```

La señal SIGUSR1 se desvía hacia la función *desvia*. El mandato del shell *kill* permitirá probar el programa.

```
scylla (2)>./ejemplo1 &
[1] 4031
scylla (2)>kill -USR1 4031
scylla (2)>Señal recibida: User defined signal 1
```

Cuando el programa recibe la señal SIGUSR1, ejecuta la función *desvia* para mostrar el mensaje en la pantalla. Seguidamente prosigue su ejecución normal.

Si el programa recibe de nuevo la misma señal, su ejecución termina.

```
scylla (2)>kill -USR1 4031
scylla (2)>
[1] User signal 1      ./ejemplo1
scylla (2)>
```

Al ejecutar la función de desviación, el sistema reinstala automáticamente la función predeterminada. En este ejemplo, termina el programa.

Para evitar este problema, la función *desvia* puede reescribirse de la manera siguiente:

```
void desvia (int sig)
{
    signal (SIGUSR1, desvia);
```

```
    printf ("Señal recibida: %s\n", strsignal(sig));  
}
```

Esta vez, al principio de la ejecución de la función de desvío, se reposiciona la señal. Sin embargo, este método puede no funcionar en el caso en que la máquina esté muy cargada. Entonces es posible que el proceso reciba una segunda señal antes de haberla desviado y por tanto termina su ejecución.

Además, durante la ejecución de la función de desvío, el proceso puede recibir otras señales.

La sección 4.1.3 presenta un método más seguro para encargarse de las señales.

La semántica presentada aquí es conforme a la de System V. La semántica BSD para esta llamada difiere ligeramente. La función de desvío predeterminada se reinstala sólo tras recibir la señal. En el ejemplo, es inútil utilizar la llamada *signal* al principio de la función *handler*.

2.3 Espera de una señal

Un proceso puede ponerse en espera de la llegada de una señal mediante la llamada al sistema *pause* cuyo prototipo es el siguiente:

```
#include <unistd.h>  
  
int pause (void);
```

Esta llamada suspende el proceso que llama hasta la llegada de una señal cualquiera. El valor de retorno de la llamada es siempre -1, y la variable *errno* contiene el valor *EINTR*.

El programa siguiente vuelve al ejemplo de la sección anterior añadiendo en el bucle infinito la llamada a *pause*.

```
#include <signal.h>  
#include <stdio.h>  
#include <string.h>  
#include <unistd.h>  
  
void desvia (int sig);  
  
int main ()  
{
```

```
if (signal (SIGUSR1, desvia) == SIG_ERR) {
    perror ("señal no desviable");
    exit (1);
}
for (;;)
    pause ();
return 0;
}

void desvia (int sig)
{
    printf ("Señal recibida: %s\n", strsignal (sig));
}
```

Este programa permite pues evitar la espera activa por la suspensión del proceso.

La llamada al sistema *pause* presenta un grave inconveniente: no permite especificar la espera de una señal particular. El proceso será despertado por la llegada de cualquier señal. La sección 4.1.4 presenta otra llamada que permite definir las señales a esperar y bloquear las demás.

3 Conceptos avanzados

3.1 Las llamadas al sistema interrumpibles

Muchas llamadas al sistema, que necesitan en su mayor parte entradas/salidas, pueden ser interrumpidas por la llegada de una señal. Cuando el proceso que ejecuta la llamada al sistema recibe una señal, la llamada al sistema se interrumpe y se trata la señal. El valor de retorno de la llamada al sistema es generalmente `-1`, y la variable `errno` contiene el valor `EINTR`. En general, la llamada al sistema no se vuelve a ejecutar, salvo por declaración explícita del programador. La sección 4.1.3 presenta un mecanismo que permite volver a ejecutar automáticamente ciertas llamadas al sistema.

3.2 Las funciones reentrantes

La llegada de una señal puede provocar errores en el desarrollo del proceso si está mal gestionado. Ciertas funciones de bibliotecas manipulan datos estáticos y no son reen-

trantes. Si estas funciones se llaman en la función de desviación de la señal, el comportamiento del proceso puede resultar alterado. Por ejemplo, la función *malloc* gestiona las zonas de memoria en forma de listas encadenadas; si una señal interviene durante la llamada a esta función y la función de desviación llama también a *malloc*, las listas corren el riesgo de quedar en un estado incoherente y el funcionamiento del proceso puede resultar erróneo.

Es pues preferible utilizar únicamente funciones reentrantes en las funciones de desviación.

3.3 Los grupos de señales

Es posible suspender o bloquear varias señales simultáneamente. La sección siguiente presenta en detalle las llamadas al sistema que permiten realizar estas operaciones. En este caso, las señales son gestionadas por grupo. Existe un cierto número de funciones definidas para gestionar estos grupos.

Estos son sus prototipos:

```
#include <signal.h>

int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, const int signum);
int sigdelset (sigset_t *set, const int signum);
int sigismember (const sigset_t *set, const int signum);
```

Las dos primeras funciones inicializan un conjunto:

- la primera crea un conjunto vacío;
- la segunda crea un conjunto con todas las señales.

Las dos funciones siguientes añaden una señal a un conjunto o realizan la operación inversa. La última permite comprobar la pertenencia de una señal a un conjunto, y devuelve 1 si la señal pertenece al conjunto, o 0 en caso contrario.

El único caso posible de error para todas estas funciones consiste en proporcionar una señal inválida. El valor de retorno es entonces -1 y la variable *errno* contiene el valor *EINVAL*.

4 Llamadas al sistema complementarias

4.1 La gestión avanzada de las señales

4.1.1 La interrupción de las llamadas al sistema

A fin de gestionar el comportamiento de un programa en caso de interrupción por una señal, los sistemas BSD proporcionan la función `siginterrupt`. Su prototipo es:

```
#include <signal.h>

int siginterrupt (int sig, int flag);
```

El comportamiento de la función es el siguiente:

- Si el valor del parámetro `flag` es nulo, se relanzará una llamada al sistema si es interrumpido por la señal `sig`.
- Si el valor del parámetro `flag` es 1 y no se ha transferido ningún dato, entonces una llamada al sistema interrumpida por la señal `sig` devuelve -1 y la variable `errno` contiene el valor `EINTR`.
- Si el valor del parámetro `flag` es 1 y los datos han sido transferidos, entonces una llamada al sistema interrumpida por la señal `sig` devuelve la cantidad de datos transferidos.

4.1.2 El bloqueo de las señales

Un proceso puede controlar la llegada de las diferentes señales. Puede retrasar su llegada bloqueándolas, y se dice entonces que están pendientes. Cada proceso posee un conjunto llamado máscara de señales, que contiene la lista de las señales bloqueadas. Este conjunto puede modificarse mediante la llamada al sistema `sigprocmask`. El prototipo de esta llamada es el siguiente:

```
#include <signal.h>

int sigprocmask (int how, const sigset_t *set, sigset_t *oldset);
```

El primer argumento indica la forma como el conjunto de señales pasado como segundo argumento debe ser tratado. Toma uno de los valores siguientes:

<i>opción</i>	<i>significado</i>
SIG_BLOCK	Las señales bloqueadas son la unión del grupo actual y el conjunto set
SIG_UNBLOCK	Las señales del conjunto set se retiran de la lista de señales bloqueadas
SIG_SETMASK	Las señales bloqueadas son las del conjunto set

El tercer argumento de la llamada contiene el conjunto de señales definido antes de la llamada a *sigprocmask*.

El valor de retorno de la llamada es 0 en caso de éxito y -1 en caso contrario. En este caso, la variable *errno* toma uno de los valores siguientes:

<i>error</i>	<i>significado</i>
EINVAL	El valor de <i>how</i> es incorrecto
EFAULT	set u oldset contiene una dirección inválida
EINTR	La llamada al sistema se ha interrumpido

Si la nueva máscara de señales desbloquea ciertas señales pendientes, al menos una de ellas se entregará antes de volver de la llamada a *sigprocmask*.

El proceso puede solicitar al sistema su lista de señales pendientes por la llamada al sistema *sigpending* cuyo prototipo es el siguiente:

```
#include <signal.h>

int sigpending (sigset_t *set);
```

El parámetro *set* contiene tras la llamada la lista de señales pendientes del proceso que llama.

El único caso posible de error para esta llamada es aquel en el que la dirección del parámetro *set* es inválida. La variable *errno* contiene entonces el valor **EFAULT**.

4.1.2.1 Ejemplo: el programa siguiente manipula la máscara de señales para bloquear la señal **SIGINT**, comprobar su llegada y desbloquearla.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

void desvia (int sig);
```

```
int main ()
{
    sigset_t  mascara,
              pendientes;

    if (signal (SIGINT, desvia) == SIG_ERR) {
        perror ("señal SIGINT no desviable");
        exit (1);
    }
    sigemptyset (&mascara);
    sigaddset (&mascara, SIGINT);
    if (sigprocmask (SIG_BLOCK, &mascara, NULL) < 0) {
        perror ("SIG_BLOCK en sigprocamsk");
        exit (1);
    }
    sleep (5);

    if (sigpending (&pendientes) < 0) {
        perror ("sigpending");
        exit (1);
    }
    if (sigismember (&pendientes, SIGINT))
        printf ("Señal SIGINT pendiente\n");

    if (sigprocmask (SIG_UNBLOCK, &mascara, NULL) < 0) {
        perror ("SIG_UNBLOCK en sigprocamsk");
        exit (1);
    }
    printf ("Señal SIGINT desbloqueada\n");

    return 0;
}

void desvia (int sig)
{
    printf ("Señal recibida: %s\n", strsignal (sig));
}
```

El programa se ejecuta, recibe la señal SIGINT por la pulsación de las teclas Ctrl-C.

```
scylla (2)>./BloqueoSenyal
```

<- Pulsación de Ctrl-C

Señal SIGINT pendiente

Señal recibida: Interrupt

Señal SIGINT desbloqueada

La señal SIGINT está primero bloqueada, y aparece en la lista de señales pendientes. Luego se trata en el momento de desbloquearla.

Una segunda emisión de esta señal termina el proceso porque la función de desvío predeterminada se ha reinstalado automáticamente. Además, si el proceso recibe varias veces la señal SIGINT cuando está bloqueada, sólo se tendrá en cuenta un solo ejemplar de la señal.

Existen también cuatro llamadas provenientes de los sistemas BSD para gestionar las máscaras de señales. Son *siggetmask*, *sigsetmask*, *sigmask* y *sigblock*. Han quedado obsoletas por la llamada a *sigprocmask* y se implementan en forma de funciones de biblioteca que utilizan esta llamada.

4.1.3 El desvío de señales

La sección 2.2 ha mostrado que la llamada al sistema *signal* tenía algunas limitaciones. Éstas afectan al control del comportamiento del proceso en la recepción de una señal. La llamada al sistema *sigaction*, introducida por la norma POSIX, permite también desviar una señal y resuelve estos problemas. El prototipo de la llamada es el siguiente:

```
#include <signal.h>

int sigaction (int signum, const struct sigaction *act,
               struct sigaction *oldact);
```

El primer parámetro representa la señal a desviar. El segundo y tercer parámetros son respectivamente el nuevo y el antiguo comportamiento a adoptar al llegar la señal.

La definición de la estructura *sigaction* es la siguiente:

tipo	campo	descripción
void (*) (int)	sa_handler	Función de desvío
sigset_t	sa_mask	Máscara de señales durante la ejecución de la función de desvío (en principio, la señal desviada se añade automáticamente a la máscara de señales)
unsigned long	sa_flags	Opciones que definen el comportamiento del proceso al recibir la señal
void (*) (void)	sa_restorer	Inutilizado, contendrá el puntero de pila de la función de desvío (aún no implementado)

Las opciones posibles para el valor del campo *sa_flags* se indican en la tabla siguiente.

<i>opción</i>	<i>significado</i>
SA_NOCLDSTOP	Si la señal es SIGCHLD , no recibir señales cuando el proceso hijo se suspenda (cuando recibe una de estas señales: SIGSTOP , SIGSTP , SIGTTIN , SIGTTOU)
SA_ONESHOT , SA_RESETHAND	Reinstalar la función de desvío predeterminada tras la recepción de la señal
SA_RESTART	Relanzar automáticamente la llamada al sistema interrumpida por la recepción de la señal
SA_NOMASK , SA_NODEFER	No bloquear la recepción de la señal durante la ejecución de la función de desvío

Los errores posibles para esta llamada al sistema son los siguientes:

<i>error</i>	<i>significado</i>
EINVAL	sig es una señal no válida o SIGKILL o SIGSTOP
EFAULT	act u oldact contiene una dirección no válida

Si el segundo argumento de *sigaction* contiene el valor **NULL**, la llamada al sistema devuelve la función actual de desvío para la señal **sig**.

Si el segundo y tercer argumentos contienen el valor **NULL**, la llamada al sistema comprueba únicamente la validez de la señal (esto es, verifica que la señal existe para esta máquina).

4.1.4 La espera de señales

La llamada al sistema *pause* no permite esperar una señal particular, el proceso se suspende hasta que llegue cualquier señal.

La norma POSIX ha regulado este problema con la introducción de la llamada al sistema *sigsuspend*. Esta llamada reemplaza temporalmente la máscara de señales del proceso por una máscara de espera y suspende el proceso hasta la llegada de una señal que no pertenezca a la máscara de espera. Estas dos operaciones se realizan de manera atómica.

El prototipo de la llamada es el siguiente:

```
#include <signal.h>

int sigsuspend (const sigset_t *mask);
```

Al volver de la llamada, la máscara de señales de origen del proceso se restaura.

Existe una llamada similar proveniente de los sistemas BSD: es la llamada *sigpause*. Su implementación se realiza en la biblioteca C por una simple llamada a *sigsuspend*.

4.2 La gestión de las alarmas

4.2.1 La llamada al sistema *alarm*

Un proceso puede solicitar al sistema que le envíe una señal en un tiempo dado. Utiliza para ello la llamada al sistema *alarm*. El prototipo de esta llamada es el siguiente:

```
#include <unistd.h>

long alarm (long seconds);
```

El parámetro *seconds* representa el período tras el cual el sistema enviará la señal *SIGALARM* al proceso. El valor de retorno de la llamada al sistema corresponde al tiempo que quedaba antes de que una alarma anteriormente solicitada se desencadene. La nueva alarma anula la anterior. El hecho de llamar a *alarm* con un tiempo nulo corresponde a la anulación de la alarma.

Esta llamada al sistema se utiliza generalmente para gestionar las temporizaciones de procesos. El ejemplo siguiente muestra la utilización de esta llamada para limitar el tiempo de espera de entrada de datos por el teclado.

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

void gestiona_alarma (int sig);

int tiempo;

int main ()
{
    signal (SIGALRM, gestiona_alarma);
    while (1) {
        tiempo = 8;
        printf ("Pulse sobre 'Intro' o el programa terminará");
        printf ("en 10 segundos\n");
        alarm (2);
        getchar ();
        alarm (0);
    }
}

void gestiona_alarma (int sig)
```

```
{
    signal (SIGALRM, gestiona_alarma);
    alarm (2);
    if (tiempo) {
        printf ("Le quedan %d segundos \n", tiempo);
        tiempo -= 2;
    } else {
        printf ("Demasiado tarde, el programa termina\n");
        exit (1);
    }
}
```

En la ejecución se obtiene el resultado siguiente:

```
bash$ ./TempoAlarm
Pulse sobre 'Intro' o el programa terminará en 10 segundos
Le quedan 8 segundos
Le quedan 6 segundos
                                     <-Pulsación de la tecla 'Intro'
Pulse sobre 'Intro' o el programa terminará en 10 segundos
Le quedan 8 segundos
Le quedan 6 segundos
Le quedan 4 segundos
Le quedan 2 segundos
Demasiado tarde, el programa termina
bash$
```

Este programa realiza un bucle con la pulsación de la tecla Intro. El tiempo restante antes del fin del programa se muestra cada dos segundos. Si el usuario pulsa esta tecla antes de 10 segundos, el contador de tiempo se vuelve a poner a cero y el bucle vuelve a empezar, si no el programa termina.

El tiempo indicado en la posición de la alarma es el tiempo mínimo al final del que el proceso tratará la señal. El proceso sólo tratará la señal si el coordinador le asigna el procesador. No habrá pues límite máximo para el tratamiento de la señal.

4.2.2 Una gestión más precisa del tiempo

La llamada al sistema anterior es limitada. No permite gestionar automáticamente una alarma. La gestión del tiempo es únicamente relativa al tiempo real, pero puede ser interesante gestionar alarmas relativas al tiempo de ejecución del proceso. Los sistemas BSD y System V han resuelto estos problemas introduciendo dos nuevas llamadas al sistema que permiten gestionar automáticamente alarmas periódicas con referencias de tiempo diferentes.

El sistema permite que cada proceso controle tres alarmas periódicas correspondientes a tres referencias de tiempo diferentes. Cuando una de las alarmas expira, se envía una señal al proceso y la alarma se reinicia.

Las alarmas se representan por tres contadores descritos en la tabla siguiente:

<i>opción</i>	<i>significado</i>
ITIMER_REAL	Contador decrementado en tiempo real; al expirar, se envía la señal SIGALRM al proceso
ITIMER_VIRTUAL	Contador decrementado cuando el proceso se ejecuta; al expirar, se envía la señal SIGVTALRM al proceso
ITIMER_PROF	Contador decrementado cuando el proceso se ejecuta y cuando el sistema se ejecuta por cuenta del proceso; al expirar, se envía la señal SIGPROF al proceso

Los prototipos de estas llamadas son los siguientes:

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

int getitimer (int which, struct itimerval *value);

int setitimer (int which, const struct itimerval *value,
               struct itimerval *ovalue);
```

Estas llamadas sirven respectivamente para recibir o escribir informaciones relativas a una alarma.

El primer parámetro de las llamadas indica de qué contador se trata. El segundo parámetro apunta a una estructura que contiene el valor de la próxima alarma y el valor actual del contador; cada uno de estos campos se describe por la estructura `timeval`, que se describe en el capítulo 4, sección 2.3.

La estructura `itimerval` es la siguiente:

<i>tipo</i>	<i>campo</i>	<i>descripción</i>
<code>struct timeval</code>	<code>it_interval</code>	Próxima alarma
<code>struct timeval</code>	<code>it_value</code>	Valor actual del contador

El contador se decrementa en `it_value` a cero; se genera una señal y el campo `it_value` se reinicializa en el valor `it_interval` y se reinicia el descuento. La alarma se para si llega a cero. Este valor se alcanza cuando se pone `it_value` a cero, o cuando el contador expira y el valor de `it_interval` es nulo.

En caso de error, la llamada al sistema devuelve `-1` y la variable `errno` contiene uno de los valores siguientes:

<i>error</i>	<i>significado</i>
EFAULT	value u o value contiene una dirección no válida
EINVAL	El primer parámetro de la llamada al sistema no es uno de los valores ITIMER_REAL , ITIMER_VIRTUAL o ITIMER_PROF

La precisión de las alarmas está vinculada a la precisión del reloj del sistema (actualmente 10 ms). Las alarmas pueden expirar pues ligeramente tras su tiempo real. La entrega de la señal se realizará inmediatamente si el proceso está activo, o será ligeramente diferida en función de la carga del sistema.

En ciertos casos en los que la carga del sistema es muy importante, porque la entrega y la recepción de una señal se separan en el sistema, y es posible que la alarma periódica **ITIMER_REAL** expire dos veces antes que el proceso haya podido tenerla en cuenta una sola vez; entonces se genera una sola señal y el proceso sólo tiene en cuenta una alarma.

La llamada al sistema *alarm* utiliza el contador **ITIMER_REAL**; es necesario, pues, evitar utilizar simultáneamente esta llamada al sistema con la llamada *setitimer*.

Para ilustrar estas llamadas al sistema, el ejemplo anterior se reescribe con la llamada *setitimer*:

```
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>
#include <unistd.h>

void gestiona_alarma (int sig);

int tiempo;

int main ()
{
    struct itimerval mi_alarma;

    mi_alarma.it_interval.tv_sec = 2;
    mi_alarma.it_interval.tv_usec = 0;

    signal (SIGALRM, gestiona_alarma);
    while (1) {
        tiempo = 8;
        printf ("Pulse sobre 'Intro' o el programa terminará");
        printf ("en 10 segundos\n");
        mi_alarma.it_value.tv_sec = 2;
        mi_alarma.it_value.tv_usec = 0;
```

```
    setitimer (ITIMER_REAL, &mi_alarma, NULL);
    getchar ();
    mi_alarma.it_value.tv_sec = 0;
    mi_alarma.it_value.tv_usec = 0;
    setitimer (ITIMER_REAL, &mi_alarma, NULL);
}
)

void gestiona_alarma (int sig)
{
    signal (SIGALRM, gestiona_alarma);
    if (tiempo) {
        printf ("Le quedan %d segundos \n", tiempo);
        tiempo -= 2;
    } else {
        printf ("Demasiado tarde, el programa termina\n");
        exit (1);
    }
}
```

Es innecesario recordar aquí la función de alarma al recibir la señal porque es periódica.

4.3 La señal SIGCHLD

Cuando un proceso termina, queda en estado ZOMBIE hasta que su proceso padre tenga conocimiento de su estado de finalización por una de las llamadas al sistema *wait* o *wait4*. El proceso padre es avisado de la terminación de uno de sus hijos por el sistema, que le envía la señal SIGCHLD. La señal se desarma únicamente cuando el proceso padre realiza una de las llamadas al sistema citadas anteriormente.

En principio, bajo Linux, la señal SIGCHLD se pone a SIG_IGN y el sistema no previene al padre de la terminación de un hijo; el sistema realiza automáticamente la terminación del hijo y éste desaparece de la tabla de procesos.

La consideración de la señal SIGCHLD es importante en la escritura de aplicaciones cliente/servidor. En general, un servidor maestro controla la llegada de peticiones y crea un servidor esclavo que responde a ellas. Tras la petición, el servidor esclavo se termina y el servidor maestro verifica que no haya habido problemas por la lectura de su código de terminación. Es necesario leer el estado de terminación, si no existe un riesgo de saturación de la tabla de procesos por procesos en estado ZOMBIE.

El ejemplo siguiente muestra un medio de gestionar la señal SIGCHLD.

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

void desvia (int sig)
{
    printf ("Terminación del proceso hijo %d\n", wait (NULL));
}

int main (void)
{
    struct sigaction action;
    int i;

    action.sa_handler = desvia;
    action.sa_flags = 0;
    action.sa_mask = 0;

    sigaction (SIGCHLD, &action, NULL);

    for (i = 0; i < 5; i++) {
        switch (fork ()) {
            case 0:
                sleep (1);
                exit (0);
                break;
            default:
                pause ();
        }
    }
    return 0;
}
```

La variable `action` se inicializa con la función de desvío. El proceso padre crea un hijo y espera su terminación cinco veces sucesivas. En un caso real, la llegada de la señal sería asíncrona respecto a la espera del padre, pero el hecho de interceptar `SIGCHLD` permitiría recuperar el estado de terminación de los procesos hijos.

5 Presentación general de la implementación

5.1 Las estructuras de datos

Las informaciones respecto a la gestión de las señales para un proceso se almacenan en la estructura `task_struct` ya presentada en el capítulo 4, sección 5.1.1. Los campos de esta estructura relacionados con las señales son los siguientes:

- `signal`: señales en espera, expresadas en forma de una cadena de bits;
- `blocked`: señales ocultas, expresadas en forma de una cadena de bits;
- `exit_signal`: número de la señal que ha causado la terminación del proceso;
- `sig`: tabla que contiene la dirección de las funciones de desvío de las señales.

La estructura `signal_struct`, que define el tipo de `sig`, se declara en el archivo `<linux/sched.h>`. Contiene los dos campos siguientes:

tipo	campo	descripción
int	count	Contador utilizado por la llamada al sistema <i>clone</i> para referenciar el número de procesos que apuntan a la estructura
struct sigaction[32]	action	Tabla de funciones de desvío

5.1.1 La creación y la terminación de procesos

En cada creación de proceso se efectúa un tratamiento respecto a las señales. En principio, un proceso hereda las acciones definidas sobre las señales por su padre; por ello, es necesario copiar estas acciones del padre al hijo. Este es el papel de la función `copy_sighand` definida en el archivo fuente `kernel/proc.c`. Esta función es llamada por la función principal de creación de procesos `do_fork` (véase el capítulo 4, sección 6.2.1). Si se trata de una llamada *fork*, la función asigna el espacio de memoria para una nueva estructura `signal_struct` e inicializa el campo `sig` del descriptor de procesos con la dirección de éste. Si, por el contrario, se ha hecho una llamada a *clone* con la opción `CLONE_SIGHAND`, el nuevo proceso comparte las acciones vinculadas a las señales con su padre, y por tanto sólo el campo `sig->count` del descriptor del padre se incrementa (esto puede corresponder a la creación de un *thread* en el proceso).

Al terminar un proceso, debe aplicarse un tratamiento inverso al anterior. Es el papel de la función `__exit_sighand`, que se define en el archivo fuente `kernel/exit.c` y es

llamada por la función de terminación `do_exit` (véase el capítulo 4, sección 6.2.2). La función decrementa el contador de referencias del campo `sig` del descriptor de procesos, y si es nulo (es decir, el proceso actual es el único *thread* activo para esta estructura), la función libera el espacio de memoria reservado al almacenamiento de las acciones asociadas a las señales.

Esta función es llamada también por `do_fork` cuando la creación no se ha podido realizar completamente.

5.1.2 La emisión de señales

La emisión de señales se efectúa en el núcleo por la función `send_sig` que controla si puede realizarse verificando, entre otras cosas, que el proceso emisor forma parte del grupo adecuado. Esta función prepara la nueva máscara a posicionar en el caso de señales especiales o de un estado particular del proceso. Finalmente, llama a la función `generate` que modificará el campo `signal` de la estructura `task_struct` del proceso destino y lo despertará si está dormido de manera interrumpible.

Una señal puede ser emitida también directamente por el núcleo hacia un proceso. En general, es el signo de un problema y el núcleo utiliza entonces la función `force_sig` que pasa por encima de todas las verificaciones. Esta función quita la señal de la lista de señales ocultas si lo está, y reinstala las opciones predeterminadas para las señales ignoradas, para forzar la recepción de la señal emitida.

5.1.3 La recepción de las señales

La recepción de una señal y el tratamiento asociado para un proceso la realiza el proceso destinatario antes de pasar del modo núcleo al modo usuario, al volver de una llamada al sistema o bien después de que el coordinador le atribuya el procesador. Esta etapa se efectúa por una rutina escrita en lenguaje ensamblador. Para la arquitectura x86, se trata de la rutina `ret_from_sys_call`, definida en el archivo fuente `arch/i386/kernel/entry.S` que controla si existen señales a tratar y llama a la función `do_signal`, definida en el archivo fuente `arch/i386/kernel/signal.c`, para asegurar la recepción de ésta. Esta función analiza el estado del proceso y la señal a recibir para realizar la acción adecuada:

- Si el proceso es *rastreado* (opción `PF_TRACED` en el campo `flags` del descriptor), pasa al estado parado y su padre es prevenido por la función `notify_parent`.
- Si la acción es `SIG_IGN` y la señal `SIGCHLD`, se realiza una llamada a la función `sys_waitpid` para provocar la desaparición del proceso hijo de la tabla de procesos.

- Si la acción es `SIG_DFL`, la función `do_exit` se llama directamente, o bien se crea un archivo *core* mediante la función `current->binfmt->core_bump`, y seguidamente se llama a la función `do_exit` para terminar el proceso.

La función `handle_signal` se invoca a continuación. Esta función controla si el proceso realizaba una llamada al sistema o no. Si este es el caso, puede ser necesario volver a ejecutarla o indicar la interrupción. Si debe ser reejecutado, el contador de programa del proceso en modo usuario se modifica para apuntar al código de llamada al sistema. Si no, se devuelve el error `EINTR`.

`handle_signal` llama luego a la función `setup_frame`. Esta es la función más sensible porque es quien modifica la pila del proceso en modo usuario para guardar en ella el contexto del proceso. Este contexto se describe en la estructura `sigcontext` del archivo `<asm/sigcontext.h>`. Esta estructura contiene todos los registros del procesador y del coprocesador matemático, el número de interrupción de la llamada al sistema en curso y el código de error asociado, así como la máscara de señales. Este contexto se coloca en la pila, pero antes la función apila el valor de la señal convertida por el ámbito de ejecución del proceso. En el caso de un binario Linux, el valor de la señal simplemente se apila, pero en otras emulaciones se llama al campo `exec_domain->signal_invmap[signr]` para apilar el valor convertido de la señal. Tras estas modificaciones de la pila, la función apila una llamada al sistema que se ejecutará tras la función de tratamiento de la señal a fin de recuperar los datos de la pila y reinstalar la máscara de señales guardada (esta llamada *sigreturn* se describe a continuación). La función modifica finalmente el registro de instrucción en modo usuario para que ejecute la función de desvío.

La llamada al sistema *sigreturn* es utilizada únicamente por la función anterior para limpiar la pila y realizar las acciones vistas anteriormente. En las próximas versiones del núcleo, esta llamada servirá también para reinstalar la pila de usuario original si se ha realizado un cambio de pila durante el tratamiento de una señal.

6 Presentación detallada de la implementación

6.1 Funciones de biblioteca

6.1.1 La gestión de los grupos de señales

Estas funciones se implementan en la biblioteca C donde se realiza una prueba para verificar que las señales pasadas como parámetro son válidas, y luego se invocan las

macros definidas en el archivo `<signal.h>`. Por ejemplo, la implementación de la función *sigaddset* definida en el archivo *libc/signal/sigaddset.c* es la siguiente:

```
/* Add SIGNO to SET. */
int DEFUN(sigaddset, (set, signo), sigset_t *set AND int signo)
{
    if (set == NULL || signo <= 0 || signo >= NSIG)
    {
        errno = EINVAL;
        return -1;
    }
    return __sigaddset (set, signo);
}
```

La macroinstrucción `__sigaddset` se define en el archivo de cabecera `<signal.h>` de la manera siguiente:

```
#define __sigmask(sig) (1 << ((sig) - 1))
#define __sigaddset(set, sig) ((*set) |= __sigmask (sig)), 0)
```

6.1.2 La función *raise*

Esta función se implementa en el archivo fuente *posix/raise.c* de la biblioteca C. Consiste en una simple llamada a *kill*.

6.2 Las llamadas al sistema

6.2.1 El bloqueo de las señales

La llamada al sistema *sigprocmask* se implementa en el archivo *kernel/signal.c* para la mayoría de arquitecturas. Sólo la versión para procesadores Alpha implementa la llamada *sigprocmask* de una manera particular para ser compatible con la llamada al sistema correspondiente de OSF/1. Para esta arquitectura, el código correspondiente se encuentra en el archivo *arch/alpha/kernel/signal.c*.

La función `sys_sigprocmask`, que corresponde a la llamada *sigprocmask*, modifica el campo `blocked` de la estructura `task_struct` del proceso actual en función de los parámetros de la llamada al sistema. Esta función verifica que el proceso no intenta bloquear una señal no bloqueable realizando un *y* binario entre la máscara aportada y la constante `_BLOCKABLE`. Ésta define el conjunto de las señales ocultables:

```
#define _BLOCKABLE (~(S(SIGKILL) | S(SIGSTOP)))
```

En función del tipo de modificación deseada, el conjunto de las señales se añade a la máscara (SIG_BLOCK) o se retira de la máscara (SIG_UNBLOCK), o bien se convierte en la propia máscara (SIG_SETMASK). Finalmente, si el parámetro `oset` es no nulo, la función escribe la máscara anterior en el espacio de usuario (función `put_user`) tras haber verificado que el parámetro apunta a una zona de memoria válida en el espacio del usuario.

6.2.2 El desvío de señales

La llamada al sistema *sigaction* está implementada en el archivo *kernel/signal.c*.

La función `sys_sigaction` corresponde a la llamada *sigaction*. Esta función verifica en primer lugar que la señal a modificar es válida (debe estar comprendida entre 1 y 32). Seguidamente la función recupera la dirección de la estructura *sigaction* correspondiente a la señal por medio del campo `sig` de la estructura *task_struct* del proceso actual.

Si el parámetro *action* no es nulo, la función verifica que no se trata de la señal SIGKILL o SIGSTOP para las que el comportamiento no puede modificarse, lee luego del espacio de usuario la estructura *sigaction* y la coloca en la variable `new_sa`. Una vez más, se hace una verificación para controlar que la dirección de la nueva función de desvío es válida.

Si el parámetro *oldaction* no es nulo, la función verifica que apunta a una zona de memoria válida en escritura en el espacio de usuario y coloca el contenido de la estructura *sigaction*, que se ha recuperado anteriormente, y que corresponde a la señal antes de la llamada.

Si hay que efectuar una modificación, la estructura *task_struct* se actualiza con la nueva estructura *sigaction*. La función `check_pending` se llama seguidamente. Esta función controla si la acción sobre la señal es SIG_IGN o SIG_DFL y en este caso elimina la señal si está en espera de recepción. Esta operación no es posible si la señal es SIGCONT, SIGCHLD o SIGWINCH y si la acción es SIG_DFL. En este caso, la función `check_pending` no modifica las señales en espera.

La llamada al sistema *signal* no es implementada por la función `sys_signal`, presente en el código fuente del núcleo, sino por una función de la biblioteca C. Esta función de biblioteca se encuentra en el archivo *libc/sysdeps/linux/signal.c*. De hecho, la función llama a *sigaction* construyendo, a partir de la dirección de la función de desvío pasada como parámetro, una variable de estructura *sigaction*. El campo `sa_flags` de esta variable toma el valor (SA_ONESHOT | SA_NOMASK | SA_INTERRUPT) & SA_RES-

TART, a fin de respetar la semántica de System V de la llamada *signal*. Al volver de la llamada, se devuelve la dirección de la anterior función de desvío al proceso.

En el caso en que se defina la constante `__USE_BSD_SIGNAL` en la compilación de un programa, la función de biblioteca `__bsd_signal` es la que corresponde a la llamada a *signal*. Su definición se encuentra en el archivo `libc/sysdeps/linux/__bsd_sig.c` del código fuente de la biblioteca. Esta función es similar a la anterior, excepto los indicadores que se ponen a cero. Si se realiza una llamada a *siginterrupt* destinada a la señal antes de la ejecución de la función de la biblioteca, entonces se activa el indicador `SA_INTERRUPT`.

6.2.3 Las señales en espera

La llamada al sistema *sigpending* viene implementada por la función `sys_sigpending` que se encuentra en el archivo `kernel/signal.c`.

Esta función devuelve simplemente el resultado de la operación y binaria entre el campo `signal` y el campo `blocked` de la estructura `task_struct`. Esto representa la lista de señales bloqueadas en espera de recepción.

6.2.4 La suspensión del proceso en espera de señales

La llamada al sistema *sigsuspend* es implementada por una función dependiente de la arquitectura de la máquina. Para los modelos x86, esta función es `sys_sigsuspend` y se encuentra en el archivo fuente `arch/i386/kernel/signal.c`.

Esta llamada modifica la máscara de señales bloqueadas en función de los parámetros recibidos y guarda la máscara anterior (campo `blocked` del descriptor de proceso). La función libera seguidamente al procesador (función `schedule`) y espera la llegada de un evento (estado `TASK_INTERRUPTIBLE`). La función prosigue su ejecución cuando se envía una señal al proceso, y luego llama a la función interna de recepción: `do_signal`. Si la señal forma parte de la máscara de señales bloqueadas, la función repite al liberarse el procesador.

Cuando interviene una señal no bloqueada, la llamada al sistema termina y devuelve el error `EINTR`.

6.2.5 El envío de señales

Las funciones del núcleo correspondientes a la emisión de señales se definen en el archivo fuente `kernel/exit.c`. Dos funciones internas son utilizadas directamente por la función correspondiente a la llamada al sistema *kill*. Son `kill_pg` y `kill_proc`.

La primera función emite la señal a todos los procesos del número de grupo pasado como parámetro. Para ello, llama a la función `send_sig` descrita anteriormente para cada uno de los procesos. Los procesos se encuentran recorriendo la tabla de procesos gracias a la macroinstrucción `for_each_task`. El resultado devuelto es el número de procesos hacia los cuales se ha enviado la señal.

La segunda función emite la señal hacia el número de proceso pasado como parámetro. Devuelve el resultado de la función `send_sig` tras haber encontrado el descriptor del proceso como anteriormente.

La llamada al sistema *kill* se implementa por la función `sys_kill`. Esta función comprueba el valor del identificador de proceso para saber si la señal debe emitirse hacia uno o más procesos:

- Si este valor es nulo, la función devuelve el resultado de la función `kill_pg` con, como parámetro, el número de grupo al que pertenece el proceso que realiza la llamada al sistema.
- Si este valor vale `-1`, la función `send_sig` se llama para todos los procesos salvo el proceso `1` (*init*) y el propio proceso que llama. El resultado devuelto es o bien un error o bien el número de procesos que han recibido efectivamente la señal.
- Si este valor es estrictamente negativo, la función `kill_pg` se llama con el valor opuesto al identificador del proceso, y se devuelve su resultado.
- Si no, se devuelve el resultado de la llamada a la función `kill_proc`.

La llamada al sistema *killpg* está implementada en una función de la biblioteca C situada en el archivo fuente *posix/killpg.c*. Esta función utiliza la llamada al sistema *kill* para emitir la señal hacia el grupo de procesos pasándole el opuesto del número de grupo que recibe.

6.2.6 La gestión de las alarmas

La gestión interna de las tres alarmas es diferente. `ITIMER_VIRTUAL` e `ITIMER_PROF` no dependen más que del tiempo de ejecución del proceso. Son gestionadas pues en la ejecución del proceso. El coordinador actualiza los contadores de las alarmas cuando da el procesador al proceso. Esta operación la realiza la función `update_process_times` del archivo fuente *kernel/sched.c*. La función actualiza primero los campos `utime` y `stime` del descriptor de procesos mediante la función `do_process_times`. Luego los dos contadores son modificados por las funciones `do_it_virt` y `do_it_prof` que utilizan los campos `it_virt_value` e

`it_prof_value` del descriptor del proceso actual. Si expiran, las señales `SIGVTALRM` o `SIGPROF` son emitidas por la función `send_sig`.

La alarma `ITIMER_REAL` debe modificarse sin que el proceso tome el control del procesador. Su contador no puede gestionarse pues de la misma manera. Se utiliza un *timer* interno (véase capítulo 4, sección 6.1.3). El contador se actualiza pues automáticamente y la emisión de la señal se realiza por la función `it_real_fn` a la cual apunta, desde la inicialización del proceso, el campo `real_timer->function` de su descriptor.

Las llamadas al sistema *setitimer* y *getitimer* son implementadas por las funciones del núcleo `sys_setitimer` y `sys_getitimer`, definidas en el archivo fuente *kernel/itimer.c*. Estas dos funciones, tras haber verificado la validez de sus parámetros, llaman a las funciones `_setitimer` o `_getitimer`. Una vez más, las operaciones son simples para las dos alarmas que dependen del tiempo de ejecución del proceso, porque basta leer o modificar los campos del descriptor de proceso. Por el contrario, para la alarma de tiempo real, es necesario retirar el *timer* de la lista para consultarlo (`del_timer`) y luego añadirlo (`add_timer`).

Se utilizan dos funciones de conversión del tiempo interno de la máquina, desde o hacia un tiempo en segundos, por las llamadas al sistema anteriores: son `tvtojiffies` y `jiffies totv`.

La llamada al sistema *alarm* se encuentra aún en el código fuente del núcleo aunque está implementada como función de biblioteca en ciertas arquitecturas. *alarm* utiliza simplemente `_setitimer` tras haber construido una variable de estructura `itinterval`. La alarma utilizada es `ITIMER_REAL`.

Capítulo 6

Sistemas de archivos

Primitivas detalladas

access, bdf flush, chdir, chmod, chown, chroot, close, closedir, dup, dup2, exec, fchdir, fchmod, fchown, fcntl, fdatsync, flock, fstat, fstatfs, fsync, ftruncate, getdents, ioctl, link, llseek, lseek, lstat, mkdir, mount, open, opendir, quotactl, read, readdir, readlink, rename, rmdir, select, stat, statfs, symlink, sync, sysfs, truncate, umount, unlink, ustat, utime, utimes, write, writev

1 Conceptos básicos

1.1 Organización de los archivos

Los archivos bajo Linux, como bajo todo sistema Unix, se organizan en forma de un árbol. La cima del árbol se representa por el directorio raíz, llamado «/», y cada uno de los nodos es un directorio que puede contener subdirectorios (otros nodos) o archivos (las hojas). Cada directorio contiene un catálogo de nombres, y cada una de las entradas de este catálogo que describe un archivo o directorio. Existen dos entradas particulares en todo directorio: «.» representa el propio directorio, y «..» representa el directorio padre.

La figura 6.1 representa un árbol de este tipo, donde los directorios se representan por rectángulos y los archivos por círculos.

Cada archivo o directorio se caracteriza por un nombre absoluto. Este nombre cita todos los directorios a explorar desde la raíz para acceder al objeto, y los directorios citados se separan por el carácter «/». Por ejemplo, el nombre absoluto del directorio *include* en la figura es */usr/include* y el del archivo *cc* es */usr/bin/cc*.

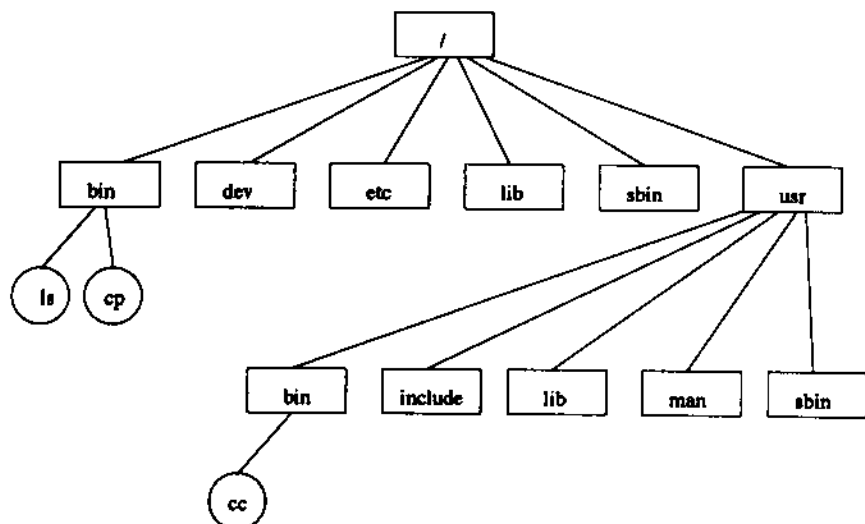


FIG. 6.1 – Organización jerárquica de los archivos

Los nombres de archivos pueden expresarse también de manera relativa a un directorio. Cada proceso posee un directorio actual que puede modificar, a partir del cual los nombres relativos que utiliza son resueltos. Por ejemplo, si el directorio actual es */usr*, el archivo */usr/bin/cc* puede representarse por el nombre relativo *bin/cc*, y el nombre relativo del archivo */bin/ls* es *../bin/ls*.

1.2 Tipos de archivos

Cada uno de los archivos presentes en el árbol se caracteriza por su tipo. Este tipo es utilizado por el sistema a fin de determinar la utilización que se hará de él. Linux define los tipos siguientes:

- directorio: un directorio es un catálogo de nombres de archivos;
- archivo normal: este tipo de archivo está destinado a contener datos pertenecientes a los usuarios, cualquiera que sea el formato de dichos datos (archivos de texto, programas fuente, programas ejecutables, etc.);
- enlace simbólico: un enlace simbólico es un puntero a otro archivo; cuando se actúa sobre un enlace, se accede de hecho al archivo apuntado;
- archivo especial: un archivo especial está asociado a un controlador de dispositivo integrado en el núcleo; cuando se accede a este archivo, se actúa de hecho sobre el dispositivo físico que tiene asociado;

- tubería con nombre: una tubería con nombre es un canal de comunicación que puede ser utilizado por varios procesos para intercambiar datos.

1.3 Enlaces con los archivos

Bajo Unix en general, es posible asociar varios nombres a un mismo archivo. Esta operación se denomina creación de un enlace con un archivo. Una vez creado un enlace, pueden utilizarse dos nombres diferentes para acceder al mismo archivo. En la práctica, esto significa que varias entradas de directorios, pudiendo ser estos últimos diferentes, se asocian al mismo contenido de archivo. Toda operación puede efectuarse citando indistintamente cualquier nombre de enlace.

El núcleo mantiene la cuenta de enlaces asociados a cada archivo. En la creación de un enlace, este número se incrementa. Al suprimir un enlace, por el mandato *rm*, este número se decrementa. Cuando este número llega a nulo, es decir, cuando se suprime el último enlace correspondiente al archivo, el contenido del archivo se suprime efectivamente, y se recupera el espacio en disco.

El número de enlaces puede mostrarse por el mandato *ls -l* (segundo campo de cada línea):

```
bbj>ls -l
total 10
-rw-r--r-- 1 card      users      9940 Apr 22 15:23 a
bbj>ln a b
bbj>ls -l
total 20
-rw-r--r-- 2 card      users      9940 Apr 22 15:23 a
-rw-r--r-- 2 card      users      9940 Apr 22 15:23 b
bbj>rm a
bbj>ls -l
total 10
-rw-r--r-- 1 card      users      9940 Apr 22 15:23 b
```

Existen varias restricciones al uso de enlaces:

- Es imposible crear enlaces a directorios. La existencia de enlaces entre directorios tendría por efecto transformar el árbol de archivos en un gráfico que podría contener ciclos, y el núcleo podría entrar en un bucle sin fin en la resolución de nombres de archivos.
- Es imposible crear enlaces entre archivos residentes en sistemas de archivos diferentes. Si */usr* y */home* se sitúan en particiones o discos diferentes, es imposible crear enlaces entre estas dos jerarquías.

A fin de saltar estas restricciones, se ha introducido un nuevo tipo de enlace: se trata de los enlaces simbólicos. Mientras que los enlaces clásicos son nombres adicionales a un archivo, los enlaces simbólicos son punteros a otros archivos. Pueden apuntar a todo tipo de archivo, incluyendo archivos inexistentes. Un enlace simbólico es en realidad un archivo de tipo especial cuyo contenido especifica el nombre del archivo destino:

```
bbj>ls -l
total 10
-rw-r--r-- 1 card      users 9940 Apr 22 15:31 a
bbj>ln -s a b
bbj>ls -l
total 10
-rw-r--r-- 1 card      users      9940 Apr 22 15:31 a
lrwxrwxrwx 1 card      users      1 Apr 22 15:31 b ->a
bbj>rm a
bbj>ls -l
total 0
lrwxrwxrwx 1 card      users      1 Apr 22 15:31 b ->a
bbj>cat b
cat: b: No such file or directory
```

1.4 Atributos de archivos

A cada archivo se le asocian varios atributos:

- su tamaño en bytes;
- el identificador del usuario propietario del archivo, es decir, el usuario que ha creado el archivo;
- el identificador del grupo de usuarios propietario del archivo;
- el número de enlaces;
- sus derechos de acceso;
- las fechas de acceso y modificación.

Los derechos de acceso a un archivo se expresan según una tripleta: los permisos del propietario, los permisos del grupo propietario, y los permisos del resto de usuarios. Cada uno de estos permisos está formado por un conjunto de tres derechos:

- el derecho de leer los datos del archivo (r);
- el derecho de escribir datos en el archivo (w);
- el derecho de ejecutar el contenido del archivo (x).

En el caso de un directorio, el derecho de lectura significa que se puede listar su contenido, el derecho de escritura significa que se pueden crear y suprimir entradas en este directorio, y el derecho de ejecución significa que se puede atravesar el directorio.

Cuando un proceso accede a un archivo, Linux verifica que está autorizado a efectuar la operación especificada. Para ello, compara los derechos requeridos por la operación con los permisos asociados al archivo.

Además de estos permisos, los derechos de acceso contienen tres bits que tienen un significado particular:

- el bit *setuid*: cuando un ejecutable que posee este bit se ejecuta, lo hace con la identidad del usuario propietario del archivo en lugar de con la identidad del usuario que lo llama;
- el bit *setgid*: cuando un ejecutable que posee este bit se ejecuta, lo hace con la identidad del grupo propietario del archivo en lugar de la identidad del grupo del usuario que llama;
- el bit *sticky*: cuando un directorio posee este bit, los archivos que contiene sólo pueden ser suprimidos por sus propietarios respectivos.

Se asocian tres fechas con cada archivo:

- la fecha del último acceso (*atime*), que se actualiza en cada acceso (lectura o escritura) al archivo;
- la fecha de la última modificación del contenido (*mtime*), que se actualiza con cada escritura de datos en el archivo;
- la fecha de la última modificación del archivo (*ctime*), que se actualiza con cada modificación del estado del archivo. Por ejemplo, el cambio de derechos de acceso de un archivo no modifica *mtime* porque el contenido no cambia, pero provoca la actualización de *ctime*.

1.5 Primitivas de entrada/salida

Linux proporciona dos conjuntos de funciones de entrada/salida:

- Las llamadas al sistema:

Estas primitivas las realiza directamente el núcleo. Permiten manipular archivos directamente (creación, supresión, cambio de nombre, etc.) así como efectuar entradas/salidas de datos sin tipo.

- La biblioteca de entradas/salidas estándar:

Esta biblioteca ofrece un conjunto de funciones que permiten efectuar entradas/salidas «de alto nivel» sobre los archivos. Contrariamente a las llamadas al sistema cuyas funcionalidades son potentes, pero en ocasiones complejas de usar, esta biblioteca permite especialmente efectuar conversiones de datos en las entradas/salidas, gracias a funciones de tipo *printf* y *scanf*.

En este capítulo sólo detallamos las llamadas al sistema de manipulación de archivos, las funciones «de alto nivel» que forman parte de la definición del lenguaje C.

1.6 Descriptores de entradas/salidas

Las llamadas al sistema que efectúan lecturas y escrituras sobre archivos utilizan descriptores de entradas/salidas. Linux asigna un descriptor en la apertura de un archivo y lo libera al cerrar dicho archivo, explícitamente o bien de manera implícita en la terminación del proceso. El proceso que actúa sobre un archivo debe proporcionar este descriptor a cada operación de lectura o escritura.

Los descriptores de entradas/salidas son simples enteros. Para el núcleo, representan entradas en una tabla de descriptores que gestiona y le permiten acceder a las informaciones de control correspondientes a los archivos abiertos.

Tres descriptores de entradas/salidas tienen un significado particular: el descriptor 0 representa la entrada estándar generalmente asociada al teclado, el descriptor 1 representa la salida estándar, y el descriptor 2 representa la salida de error. Estas dos salidas se asocian generalmente a la pantalla.

2 Llamadas básicas al sistema

2.1 Entradas/salidas sobre archivos

2.1.1 Apertura y cierre de archivos

Todo archivo debe abrirse antes de poder leer o escribir datos. La sintaxis de la llamada a *open* es la siguiente:

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

int open (const char *pathname, int flags);
int open (const char *pathname, int flags, mode_t mode);
```

El parámetro *pathname* especifica el nombre del archivo a abrir, según el modo de apertura indicado por *flags*. El parámetro *mode* especifica los derechos de acceso a posicionar en el archivo en una creación. *open* devuelve el descriptor de entradas/salidas correspondiente al archivo abierto, o el valor -1 en caso de error.

El modo de apertura se expresa en función de constantes definidas en el archivo de cabecera *<fcntl.h>*: *O_RDONLY* para un acceso en lectura exclusiva, *O_WRONLY* para un acceso en escritura exclusiva, y *O_RDWR* para un acceso en lectura y escritura. Otras opciones de apertura pueden especificarse también combinadas con una operación *O* binaria (operador «|» del lenguaje C):

<i>opción</i>	<i>significado</i>
O_CREAT	Creación del archivo si no existe
O_EXCL	Provoca un error si se especifica O_CREAT y el archivo ya existe
O_TRUNC	Supresión del contenido anterior del archivo si existe
O_APPEND	Apertura en modo añadir, toda escritura se realiza al final del archivo
O_NONBLOCK	Apertura en modo compartido
O_SYNC	Apertura del archivo en modo síncrono: toda actualización se escribe inmediatamente en el disco

Los derechos de acceso a posicionar (parámetro *mode*) se expresan por la combinación binaria de constantes definidas en el archivo de cabecera *<sys/stat.h>*:

<i>opción</i>	<i>significado</i>
S_ISUID	Bit <i>setuid</i>
S_ISGID	Bit <i>setgid</i>
S_ISVTX	Bit <i>sticky</i>
S_IRUSR	Derecho de lectura para el propietario
S_IWUSR	Derecho de escritura para el propietario
S_IXUSR	Derecho de ejecución para el propietario
S_IRWXU	Derechos de lectura, escritura y ejecución para el propietario
S_IRGRP	Derecho de lectura para el grupo de usuarios
S_IWGRP	Derecho de escritura para el grupo de usuarios
S_IXGRP	Derecho de ejecución para el grupo de usuarios
S_IRWXG	Derechos de lectura, escritura y ejecución para el grupo de usuarios
S_IROTH	Derecho de lectura para el resto de usuarios
S_IWOTH	Derecho de escritura para el resto de usuarios
S_IXOTH	Derecho de ejecución para el resto de usuarios
S_IRWXO	Derechos de lectura, escritura y ejecución para el resto de usuarios

En caso de error, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EACCESS	El acceso especificado no es posible
EXIST	pathname especifica un nombre de archivo existente y se han especificado las opciones O_CREAT y O_EXCL
EFAULT	pathname contiene una dirección no válida
EISDIR	pathname se refiere a un directorio y el acceso especificado incluye la escritura
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
EMFILE	Se ha alcanzado el número máximo de archivos abiertos por el proceso
ENAMETOOLONG	pathname especifica un nombre de archivo demasiado largo
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOENT	pathname se refiere a un nombre de archivo inexistente y la opción O_CREAT no se ha especificado
ENOSPC	El sistema de archivos está saturado
ENOTDIR	Uno de los componentes de pathname, utilizado como nombre de directorio, no es un directorio
EROFS	El sistema de archivos está en lectura exclusiva y el acceso especificado incluye la escritura
ETXTBSY	pathname se refiere a un programa binario en curso de ejecución y el acceso especificado incluye la escritura

Tras usar un archivo, un proceso debe utilizar la llamada al sistema *close* a fin de cerrarlo. La sintaxis de *close* es:

```
#include <unistd.h>

int close (int fd);
```

El parámetro *fd* corresponde a un descriptor de entrada/salida devuelto por la llamada *open*. Se devuelve el valor 0 en caso de éxito. En caso de error, se devuelve el valor -1. El único error posible es **EBADF**, que indica que el descriptor especificado por *fd* es inválido.

2.1.2 Lectura y escritura de datos

Dos llamadas permiten a un proceso leer o escribir datos en un archivo previamente abierto por una llamada a *open*. Linux considera los archivos como una serie de bytes sin tipo, sin ninguna estructura, y no impone ninguna restricción sobre los datos que un proceso puede leer o escribir en un archivo. Si el archivo debe ser estructurado, le corresponde a la aplicación gestionarlo.

La lectura de datos se efectúa por la llamada *read*:

```
#include <unistd.h>

ssize_t read (int fd, void *buf, size_t count);
```


read provoca la lectura de datos desde el archivo cuyo descriptor de entradas/salidas se pasa en el parámetro *fd*. Los datos leídos se colocan en la memoria intermedia cuya dirección se especifica por *buf*, con la variable *count* que indica su tamaño en bytes. *read* devuelve el número de bytes que se han leído desde el archivo (0 en el caso en que se llegue al fin del archivo) o el valor -1 en caso de error.

En caso de fallo, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EBADF	El descriptor de entradas/salidas especificado no es válido
EFAULT	<i>buf</i> contiene una dirección no válida
EINTR	La llamada al sistema ha sido interrumpida por la recepción de una señal
EINVAL	<i>fd</i> se refiere a un objeto sobre el que no es posible la lectura
EIO	Se ha producido un error de entradas/salidas
ISDIR	<i>fd</i> se refiere a un directorio

La llamada *write* permite escribir datos en un archivo. Su sintaxis es la siguiente:

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

```
ssize_t write (int fd, const char *buf, size_t count);
```

Los datos cuya dirección está contenida en *buf* y cuya longitud en bytes se pasa en el parámetro *count* se escriben en el archivo especificado por el descriptor de entradas/salidas *fd*. *write* devuelve el número de bytes escritos en el archivo, o el valor -1 en caso de error.

En caso de fallo, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EBADF	El descriptor de entradas/salidas especificado no es válido
EFAULT	<i>buf</i> contiene una dirección no válida
EINTR	La llamada al sistema ha sido interrumpida por la recepción de una señal
EINVAL	<i>fd</i> se refiere a un objeto sobre el que no es posible la escritura
EIO	Se ha producido un error de entradas/salidas
ISDIR	<i>fd</i> se refiere a un directorio
PIPE	<i>fd</i> se refiere a una tubería sobre la que no existe ya proceso lector
ENOSPC	El sistema de archivos está saturado

El programa siguiente utiliza las funciones de entradas/salidas sobre archivos para copiar el contenido de un archivo en otro:

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

```
void main (int argc, char *argv[])
{
    int fd1;
    int fd2;
    char buffer[1024];
    ssize_t n;

    /* Control de los argumentos */
    if (argc != 3) {
        printf ("Uso: %s origen destino\n", argv[0]);
        exit (1);
    }
    /* Apertura del archivo origen */
    fd1 = open (argv[1], O_RDONLY);
    if (fd1 == -1) {
        perror ("open 1");
        exit (1);
    }
    /* Apertura del archivo destino */
    fd2 = open (argv[2], O_WRONLY | O_TRUNC | O_CREAT,
               S_IRUSR | S_IWUSR);
    if (fd2 == -1) {
        perror ("open 2");
        exit (1);
    }
    /* Bucle de copia (lectura en origen, escritura en destino) */
    do {
        n = read (fd1, (void *) buffer, sizeof (buffer));
        if (n > 0)
            if (write (fd2, (void *) buffer, n) != n)
                perror ("write");
    } while (n > 0);
    /* cierre de los archivos */
    (void) close (fd1);
    (void) close (fd2);
    /* Terminación */
    exit (0);
}
```

2.1.3 Posicionamiento en un archivo

Las llamadas *read* y *write* permiten efectuar lecturas y escrituras secuenciales de datos. No existe ninguna primitiva que permita efectuar lecturas o escrituras directas por cuanto que el núcleo no gestiona ninguna estructuración.

Sin embargo, es posible modificar el valor del puntero de lectura/escritura de un archivo. De este modo, se pueden simular entradas/salidas directas. Las llamadas al sistema *lseek* y *llseek* permiten posicionar el puntero de lectura/escritura:

```
#include <unistd.h>

off_t lseek (int fd, off_t offset, int whence);

loff_t llseek (int fd, loff_t offset, int whence);
```

El parámetro *fd* representa el descriptor de entradas/salidas asociado al archivo, *offset* define el desplazamiento en bytes del puntero de lectura/escritura respecto a una base especificada por *whence*. Este último parámetro puede tomar los valores siguientes:

opción	significado
SEEK_SET	Posicionamiento respecto al inicio del archivo
SEEK_CUR	Posicionamiento respecto a la posición actual
SEEK_END	Posicionamiento respecto al fin del archivo

lseek devuelve la nueva posición actual en bytes respecto al principio del archivo, o el valor -1 en caso de fallo.

En caso de error, la variable *errno* puede tomar los valores siguientes:

error	significado
EINVAL	El descriptor de entrada/salida especificado no es válida
EBADF	<i>whence</i> especifica un valor no válido
ESPIPE	<i>fd</i> se refiere a una tubería

No existe ninguna llamada que permita obtener la posición actual en un archivo. Pero es posible llamar a *lseek* sin modificar la posición actual y utilizar su resultado, como sigue:

```
#include <unistd.h>

off_t
tell (int fd)
{
    return lseek (fd, (off_t) 0, SEEK_CUR);
}
```

La primitiva *llseek* es particular de Linux, tiene la misma función que *lseek*, pero permite especificar un desplazamiento expresado en 64 bits (tipo *loff_t*), incluso en arquitecturas de 32 bits.

2.1.4 Guardar los datos modificados

En una escritura en un archivo, los datos se escriben primero en memoria intermedia, mediante el *búffer caché*, y luego se guardan regularmente en disco por el proceso *update*. Si el archivo ha sido abierto con la opción `O_SYNC`, las modificaciones efectuadas se escriben de manera síncrona en disco.

Este guardado en memoria intermedia es particularmente interesante a nivel de rendimiento por varias razones:

- los procesos que escriben en archivos no se suspenden durante las escrituras, el núcleo se encarga de escribir los datos en disco de manera asíncrona;
- si varios procesos acceden a los mismos archivos, sólo la primera lectura debe efectuarse desde el disco; las lecturas siguientes de los mismos datos se limitan a una transferencia de datos entre el *búfer caché* y los espacios de direccionamiento de los distintos procesos.

La escritura asíncrona de los datos presenta, sin embargo, un problema de fiabilidad: si se produce un fallo, por ejemplo un corte de corriente, entre la llamada de *write* y la escritura real de los datos en disco, las modificaciones se pierden.

Linux ofrece tres llamadas al sistema que permiten provocar la escritura en disco de los datos modificados:

```
#include <unistd.h>
int sync (void);
int fsync (int fd);
int fdatasync (int fd);
```

La llamada *sync* provoca la escritura de todos los datos modificados, mientras que la llamada *fsync* provoca la escritura de las modificaciones correspondientes al archivo caracterizado por el descriptor de entradas/salidas *fd*. *fdatasync* efectúa el mismo tratamiento que *fsync*, pero puede no reescribir las informaciones de control del archivo (como la fecha de última modificación, por ejemplo) y puede así economizar una escritura en disco.

En caso de fallo de *fsync* o *fdatasync*, la variable `errno` puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EBADF	El descriptor de entradas/salidas especificado no es válido
EINVAL	<i>fd</i> se refiere a un objeto sobre el que no es posible la sincronización
EIO	Se ha producido un error de entrada/salida

2.2 Manipulación de archivos

2.2.1 Creación de enlaces

La llamada al sistema *link* crea un nuevo enlace a un archivo. Su sintaxis es:

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

```
int link (const char *oldpath, const char *newpath);
```

link crea un enlace, cuyo nombre se especifica por el parámetro *newpath*, al archivo cuyo nombre se pasa en el parámetro *oldpath*. El archivo existente y el enlace deben encontrarse en el mismo sistema de archivos y no deben ser directorios.

<i>error</i>	<i>significado</i>
EACCES	El proceso no tiene acceso en lectura al directorio que contiene el archivo especificado por <i>oldpath</i> o en escritura al directorio que contiene el archivo especificado por <i>newpath</i>
EXIST	<i>newpath</i> especifica un nombre de archivo existente
EFAULT	<i>oldpath</i> o <i>newpath</i> contiene una dirección no válida
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENLINK	Se ha alcanzado el número máximo de enlaces
ENAMETOOLONG	<i>oldpath</i> o <i>newpath</i> especifica un nombre de archivo demasiado largo
ENOENT	<i>oldpath</i> especifica un nombre de archivo inexistente
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOSPC	El sistema de archivos está saturado
ENOTDIR	Uno de los componentes de <i>oldpath</i> o <i>newpath</i> , utilizado como nombre de directorio, no lo es
EPERM	El sistema de archivos que contiene los archivos especificados por <i>oldpath</i> y <i>newpath</i> no soporta la creación de enlaces, o bien <i>oldpath</i> corresponde a un nombre de directorio
EROFS	El sistema de archivos es de lectura exclusiva
EXDEV	Los archivos especificados por <i>oldpath</i> y <i>newpath</i> se encuentran en sistemas de archivos diferentes

2.2.2 Supresión de archivos

La llamada al sistema *unlink* permite suprimir un enlace, y por tanto un archivo si se trata del último enlace. Su sintaxis es la siguiente:

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

```
int unlink (const char *pathname);
```

El archivo especificado por el parámetro `pathname` se suprime si el proceso que llama posee los derechos de acceso suficientes, es decir, el derecho de escritura en el directorio que contiene el archivo.

En caso de fallo, la variable `errno` puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EACCES	El proceso actual no tiene acceso en escritura sobre el directorio que contiene el archivo especificado por <code>pathname</code>
EFAULT	<code>pathname</code> contiene una dirección no válida
ENAMETOOLONG	<code>pathname</code> especifica un nombre de archivo demasiado largo
ENOENT	<code>pathname</code> se refiere a un nombre de archivo inexistente
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de <code>pathname</code> , utilizado como nombre de directorio, no lo es
EPERM	<code>pathname</code> especifica el nombre de un directorio
EROFS	El sistema de archivos es de lectura exclusiva

2.2.3 Cambio de nombre de un archivo

La llamada al sistema `rename` permite modificar el nombre de un archivo, y por tanto renombrarlo o desplazarlo entre dos directorios, situados en el mismo sistema de archivos. Su sintaxis es la siguiente:

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

```
int rename (const char *oldpath, const char *newpath);
```

El archivo fuente, cuyo nombre se pasa en el parámetro `oldpath`, se renombra según el nuevo nombre pasado en el parámetro `newpath`. Si este último parámetro especifica un nombre de archivo existente, este archivo se suprime previamente.

En caso de error, la variable `errno` puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EACCES	El proceso no tiene acceso en escritura al directorio que contiene el archivo especificado por <code>oldpath</code> o al directorio que contiene el archivo especificado por <code>newpath</code>
EBUSY	<code>newpath</code> especifica el nombre de un directorio utilizado como directorio actual o raíz de un proceso
EFAULT	<code>oldpath</code> o <code>newpath</code> contiene una dirección no válida
EINVAL	<code>newpath</code> especifica el nombre de un directorio existente mientras que <code>oldpath</code> especifica un nombre de archivo
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENAMETOOLONG	<code>oldpath</code> o <code>newpath</code> especifica un nombre de archivo demasiado largo
ENOENT	<code>oldpath</code> especifica un nombre de archivo inexistente
ENOSPC	El sistema de archivos está saturado

ENOENT	Uno de los componentes de <i>oldpath</i> o <i>newpath</i> , utilizado como nombre de directorio, no es un directorio
ENOTEMPTY	<i>newpath</i> especifica el nombre de un directorio no vacío
EROFS	El sistema de archivos es de lectura exclusiva
EXDEV	Los archivos especificados por <i>oldpath</i> y <i>newpath</i> se encuentran en sistemas de archivos diferentes

2.2.4 Cambio de tamaño de un archivo

Un proceso puede modificar el tamaño de un archivo, bien truncando su contenido, o bien aumentando su tamaño. Para hacerlo, se cuenta con dos primitivas:

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

```
int truncate (const char *pathname, size_t length);
int ftruncate (int fd, size_t length);
```

La llamada al sistema *truncate* modifica el tamaño del archivo cuyo nombre se especifica por el parámetro *pathname*. *ftruncate* modifica el tamaño de un archivo abierto cuyo descriptor de entrada/salida se pasa en el parámetro *fd*. El parámetro *length* indica el nuevo tamaño del archivo en bytes. Si el archivo es mayor que el tamaño especificado, su contenido se trunca, y si es más pequeño, su contenido se amplía.

En caso de error en la ejecución de *truncate*, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EACCES	El proceso no tiene acceso en escritura al archivo especificado por <i>pathname</i>
EFAULT	<i>pathname</i> contiene una dirección no válida
EIO	Se ha producido un error de entrada/salida
EINVAL	<i>pathname</i> se refiere a un directorio
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENAMETOOLONG	<i>pathname</i> especifica un nombre de archivo demasiado largo
ENOENT	<i>pathname</i> se refiere a un nombre de archivo inexistente
ENOTDIR	Uno de los componentes de <i>pathname</i> , utilizado como nombre de directorio, no lo es
EROFS	El sistema de archivos es de lectura exclusiva
EXISTS	<i>pathname</i> se refiere a un programa binario en curso de ejecución

En el caso de *ftruncate*, son también posibles los errores siguientes:

<i>error</i>	<i>significado</i>
EBADF	El descriptor de entrada/salida no es válido
EINVAL	<i>fd</i> se refiere a un archivo que no está abierto en escritura

2.2.5 Derechos de acceso sobre un archivo

Los derechos de acceso sobre un archivo pueden posicionarse en la creación del archivo por una llamada a *open*. También pueden modificarse luego mediante las llamadas *chmod* y *fchmod*. Su sintaxis es la siguiente:

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

int chmod (const char *pathname, mode_t mode);
int fchmod (int fd, mode_t mode);
```

chmod modifica los derechos de acceso al archivo cuyo nombre se pasa en el parámetro *pathname*. *fchmod* modifica los derechos de acceso a un archivo abierto cuyo descriptor de entrada/salida se especifica por el parámetro *fd*. El parámetro *mode* define los derechos de acceso a posicionar y es similar al tercer parámetro de la llamada *open*.

Las dos llamadas sólo se autorizan para el usuario propietario del archivo y el superusuario.

En caso de error de la llamada al sistema *chmod*, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EFAULT	<i>pathname</i> contiene una dirección no válida
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENAMETOOLONG	<i>pathname</i> especifica un nombre de archivo demasiado largo
ENOENT	<i>pathname</i> se refiere a un nombre de archivo inexistente
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de <i>pathname</i> , utilizado como nombre de directorio, no lo es
EPERM	El proceso no posee los derechos del propietario del archivo, y no es privilegiado
EROFS	El sistema de archivos está en lectura exclusiva

Además, *fchmod* puede devolver el error **EBADF**, indicando que el descriptor *fd* no es válido.

Un proceso puede comprobar si tiene acceso a un archivo. Para hacerlo, Linux ofrece la llamada *access*, cuya sintaxis es la siguiente:

```
#include <unistd.h>

int access (const char *pathname, int mode);
```

El parámetro *mode* representa los derechos de acceso a comprobar, y se expresa por una combinación de las constantes siguientes:

<i>opción</i>	<i>significado</i>
F_OK	Comprueba la existencia del archivo
R_OK	Comprueba el acceso en lectura
W_OK	Comprueba el acceso en escritura
X_OK	Comprueba el acceso en ejecución

access devuelve el valor 0 si el acceso es posible, y el valor -1 en caso contrario. En este último caso, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EACCES	El acceso se rechaza
EFAULT	pathname contiene una dirección no válida
EINVAL	El valor especificado por mode no es válido
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENAMETOOLONG	pathname especifica un nombre de archivo demasiado largo
ENOENT	pathname se refiere a un nombre de archivo inexistente
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de pathname, utilizado como nombre de directorio, no lo es

2.2.6 Modificación del usuario propietario

En la creación de un archivo, el usuario y el grupo propietarios se inicializan según la identidad del proceso que llama. Linux proporciona dos llamadas al sistema que permiten modificar las propiedades de un archivo:

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

int chown (const char *pathname, uid_t owner, gid_t group);
int fchown (int fd, uid_t owner, gid_t group);
```

chown permite modificar el usuario y el grupo propietarios del archivo cuyo nombre se pasa en el parámetro *pathname*. *fchown* actúa, por su parte, sobre un archivo abierto cuyo descriptor de entrada/salida se pasa en el parámetro *fd*. En ambos casos, *owner* representa el identificador del nuevo usuario propietario, *group* representa el identificador del grupo. Cada uno de estos dos parámetros puede omitirse especificando el valor -1.

Sólo un proceso que posea los derechos de superusuario puede modificar el usuario propietario de un archivo. El propietario de un archivo sólo puede modificar el grupo si es miembro del nuevo grupo.

En caso de error de la llamada al sistema *chown*, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EFAULT	pathname contiene una dirección no válida
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENAMETOOLONG	pathname especifica un nombre de archivo demasiado largo
ENOENT	pathname se refiere a un nombre de archivo inexistente
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de pathname, utilizado como nombre de directorio, no lo es
EPERM	El proceso no posee los privilegios necesarios
EROFS	El sistema de archivos es de lectura exclusiva

Además, *fchown* puede devolver el error **EBADF** indicando que el descriptor *fd* no es válido.

2.3 Gestión de directorios

2.3.1 Creación de directorios

Un directorio se crea por la llamada al sistema *mkdir* cuya sintaxis es la siguiente:

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

int mkdir (const char *pathname, mode_t mode);
```

El nombre del directorio a crear se especifica en el parámetro *pathname*. El parámetro *mode* indica los derechos de acceso a posicionar sobre el nuevo directorio. Es similar al tercer parámetro de la llamada *open*.

En caso de fallo, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EACCES	El proceso no tiene acceso en escritura al directorio padre del directorio especificado por <i>pathname</i>
EXIST	<i>pathname</i> especifica un nombre de archivo existente
EFAULT	<i>pathname</i> contiene una dirección no válida
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENAMETOOLONG	<i>pathname</i> especifica un nombre de archivo demasiado largo
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de <i>pathname</i> , utilizado como nombre de directorio, no lo es
EROFS	El sistema de archivos es de lectura exclusiva
ENOSPC	El sistema de archivos está saturado

2.3.2 Supresión de directorios

La llamada al sistema *rmdir* permite suprimir un directorio. Este último deberá estar vacío, excepto las entradas «.» y «.». La sintaxis de *rmdir* es:

```
#include <unistd.h>

int rmdir (const char *pathname);
```

El parámetro *pathname* indica el directorio a suprimir.

En caso de fallo, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EINVAL	El proceso no tiene acceso en escritura al directorio padre del directorio especificado por <i>pathname</i>
EBUSY	<i>pathname</i> especifica el nombre de un directorio utilizado como directorio actual o raíz de un proceso
EFAULT	<i>pathname</i> contiene una dirección no válida
ENAMETOOLONG	<i>pathname</i> especifica un nombre de archivo demasiado largo
ENOENT	<i>pathname</i> se refiere a un nombre de archivo inexistente
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de <i>pathname</i> , utilizado como nombre de directorio, no lo es
ENOTEMPTY	El directorio especificado por <i>pathname</i> no está vacío
EROFS	El sistema de archivos es de lectura exclusiva

2.3.3 Directorio actual

A todo proceso se le asocia un directorio actual. Los nombres de archivos relativos utilizados por el proceso se resuelven desde este directorio.

Las llamadas al sistema *chdir* y *fchdir* permiten a un proceso modificar su directorio actual. Su sintaxis es la siguiente:

```
#include <unistd.h>

int chdir (const char *pathname);
int fchdir (int fd);
```

chdir utiliza el parámetro *pathname* que especifica el nombre del nuevo directorio actual. *fchdir*, por su parte, utiliza un descriptor de entrada/salida (parámetro *fd*) obtenido por una llamada anterior a la primitiva *open*.

En caso de fallo de *chdir*, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EFAULT	pathname contiene una dirección no válida
ENOENT	pathname se refiere a un nombre de archivo inexistente
ENAMETOOLONG	pathname especifica un nombre de archivo demasiado largo
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de pathname, utilizado como nombre de directorio, no lo es, o pathname no especifica el nombre de un directorio
EPERM	El proceso no tiene acceso en ejecución al directorio especificado por pathname

Además, *fchdir* puede devolver el error **EBADF**, que indica que el descriptor *fd* no es válido.

No existe una llamada al sistema que permita a un proceso obtener el nombre de su directorio actual, pero la función de biblioteca *getcwd* cumple esta función. Su sintaxis es la siguiente:

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

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

getcwd devuelve el nombre absoluto del directorio actual en el parámetro *buf*. El parámetro *size* especifica el tamaño de la memoria intermedia apuntada por *buf*. El puntero devuelto por *getcwd* es el parámetro *buf* en caso de éxito. En caso de error, se devuelve el valor **NULL**. El único código de error que puede devolverse es **ERANGE** en el caso en que el tamaño de la memoria intermedia sea demasiado pequeño para contener el valor absoluto del directorio actual.

2.3.4 Directorio raíz local

A todo proceso se le asocia un directorio raíz, que puede ser distinto del verdadero directorio raíz. Los nombres de archivos absolutos utilizados por el proceso se resuelven desde este directorio. Normalmente, este directorio es el mismo para todos los procesos y corresponde a la raíz del árbol de archivos, pero un proceso que posea los derechos del superusuario puede modificar su directorio raíz para restringir a un subárbol el conjunto de archivos a los que puede acceder gracias a la llamada de sistema *chroot*:

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

```
int chroot (const char *pathname);
```

El parámetro *pathname* especifica el nombre del nuevo directorio raíz a utilizar. En caso de éxito, *chroot* devuelve el valor 0, y el proceso actual se restringe al conjunto de archivos y directorios contenidos en el subárbol situado bajo el directorio especificado. En caso de error, *chroot* devuelve el valor -1.

En caso de error de *chroot*, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EFAULT	pathname contiene una dirección no válida
ENOENT	pathname se refiere a un nombre de archivo inexistente
ENAMETOOLONG	pathname especifica un nombre de archivo demasiado largo
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de pathname, utilizado como nombre de directorio, no lo es, o pathname no especifica el nombre de un directorio
EPERM	El proceso no es un proceso privilegiado

La principal aplicación de *chroot* consiste en ejecutar una aplicación en un entorno restringido por razones de seguridad. Los servidores FTP anónimos utilizan esta primitiva para ofrecer sólo un subárbol de sus archivos a los usuarios no identificados.

2.3.5 Exploración de los directorios

Aunque el núcleo gestiona los directorios como archivos en disco, un proceso no puede acceder directamente al catálogo que contienen. El formato de estos catálogos es dependiente del tipo de sistema de archivos, y Linux impide su lectura directa por la llamada al sistema *read* para evitar una interpretación errónea de su contenido por parte de los procesos.

A fin de permitir a los procesos el acceso a las entradas de un directorio, Linux ofrece tres llamadas al sistema, llamadas *opendir*, *readdir* y *closedir*. Su sintaxis es la siguiente:

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

DIR *opendir (const char *pathname);
struct dirent *readdir (DIR *dir);
int closedir (DIR *dir);
```

El tipo *DIR* está definido en el archivo *<dirent.h>* y representa un descriptor de directorio abierto. Su contenido no es accesible, de igual modo que el tipo *FILE* utilizado por la biblioteca de entrada/salida estándar.

opendir abre un directorio en modo lectura, siendo el nombre de este directorio especificado en el parámetro *pathname*, y devuelve un descriptor de directorio abierto.

En caso de fallo, la variable `errno` puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EACCESS	El proceso no tiene acceso en lectura al directorio especificado por <code>pathname</code>
EMFILE	Se ha alcanzado el número máximo de archivos abiertos para el proceso actual
ENFILE	Se ha alcanzado el número máximo de archivos abiertos para el sistema
ENOENT	<code>pathname</code> se refiere a un nombre de archivo inexistente
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de <code>pathname</code> , utilizado como nombre de directorio, no lo es, o <code>pathname</code> no especifica el nombre de un directorio

`readdir` efectúa la lectura de una entrada del directorio cuyo descriptor se pasa en el parámetro `dir`, y devuelve un puntero a una estructura `dirent` (esta variable está contenida en el espacio de direccionamiento de la biblioteca C, y cada llamada a `readdir` reemplaza su contenido anterior), que contiene las características de la entrada actual de directorio. Cuando se alcanza el fin del directorio, se devuelve el valor `NULL`.

En caso de error, la variable `errno` puede tomar el valor `EBADF`, indicando que el descriptor especificado por `dir` no es válido.

La estructura `dirent` está definida en el archivo de cabecera `<dirent.h>` y contiene los campos siguientes:

<i>tipo</i>	<i>campo</i>	<i>descripción</i>
long	<code>d_ino</code>	número del i-nodo correspondiente a la entrada
unsigned short	<code>d_reclen</code>	tamaño de la estructura devuelta
char []	<code>d_name</code>	nombre del archivo contenido en la entrada

Finalmente, `closedir` cierra el directorio cuyo descriptor se pasa en el parámetro `dir`.

En caso de error, la variable `errno` puede tomar el valor `EBADF`, indicando que el descriptor especificado por `dir` no es válido.

El programa siguiente es una emulación simplificada del mandato `ls`: abre el directorio actual, muestra el contenido de cada entrada obtenido llamando a `readdir`, y cierra el directorio.

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

```
void main (void)
{
```

```
DIR *dir;
struct dirent *dp;

/* Apertura del directorio actual */
dir = opendir (".");
if (dir == NULL) {
    perror ("opendir");
    exit (1);
}
/* Exploración del directorio*/
dp = readdir (dir);
while (dp != NULL) {
    printf ("%s\n", dp->d_name);
    dp = readdir (dir);
}
/* Cierre del directorio */
(void) closedir (dir);
/* Terminación */
exit (0);
}
```

La primitiva *getdents* se ha añadido recientemente a Linux, y efectúa la misma acción que *readdir*, pero permite leer varias entradas consecutivas en una sola llamada. Su sintaxis es la siguiente:

```
#include <unistd.h>
#include <dirent.h>
#include <unistd.h>

int getdents (int fd, struct dirent *dirp, unsigned int size);
```

El parámetro *fd* especifica un descriptor de entrada/salida correspondiente a un directorio, *dirp* contiene la dirección de una memoria intermedia donde se colocarán los resultados, y *size* indica el tamaño en bytes de esta memoria. Tras la llamada a *getdents*, se colocan una o más variables de tipo estructura *dirent* en la memoria intermedia, y se devuelve el número de bytes inicializados. Para usar los resultados, hay que explorar la lista de entradas de directorio creada en la memoria intermedia.

En caso de error, la variable *errno* puede tomar los valores siguientes:

error	significado
EINVAL	El descriptor especificado por <i>dir</i> no es válido
EFAULT	<i>buf</i> contiene una dirección no válida

El programa siguiente es una variación del anterior: abre el directorio actual, obtiene las entradas de directorio llamando a *getdents*, las muestra y cierra el directorio.

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

void main (void)
{
    int            dir;
    struct dirent  *dp;
    char           buf[1024];
    int            i, n;

    /* Apertura del directorio actual */
    dir = open (".", O_RDONLY);
    if (dir == -1) {
        perror ("open");
        exit (1);
    }
    /* Exploración del directorio */
    n = getdents (dir, buf, sizeof (buf));
    if (n == -1) {
        perror ("getdents");
        exit (1);
    }
    while (n > 0) {
        /* Exploración de la lista devuelta por getdents */
        i = 0;
        while (i < n) {
            dp = (struct dirent *) (buf + i);
            printf ("%s\n", dp->d_name);
            i += dp->d_reclen;
        }
        n = getdents (dir, buf, sizeof (buf));
        if (n == -1) {
            perror ("getdents");
            exit (1);
        }
    }
    /* Cierre del directorio */
    (void) close (dir);
    /* Terminación */
    exit (0);
}
```


2.4 Enlaces simbólicos

Los enlaces simbólicos constituyen un tipo especial de archivo y no pueden manipularse con las mismas primitivas que los archivos regulares. Linux proporciona dos llamadas al sistema que permiten crear un enlace simbólico y leer el nombre del archivo al cual apunta. Su sintaxis es la siguiente:

```
#include <unistd.h>

int symlink (const char *oldpath, const char *newpath);
int readlink (const char *pathname, char *buf, size_t bufsiz);
```

La llamada *symlink* crea un enlace simbólico, cuyo nombre se especifica por el parámetro *newpath*, que apunta al archivo cuyo nombre se pasa en el parámetro *oldpath*. Este último archivo puede ser de un tipo cualquiera y puede incluso no existir.

En caso de error, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EXIST	<i>newpath</i> especifica un nombre de archivo existente
FAULT	<i>oldpath</i> o <i>newpath</i> contiene una dirección inválida
LOOP	Se ha encontrado un ciclo de enlaces simbólicos
NAMETOOLONG	<i>oldpath</i> o <i>newpath</i> especifica un nombre de archivo demasiado largo
NOTDIR	Uno de los componentes de <i>newpath</i> , utilizado como nombre de directorio, no lo es
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOSPC	El sistema de archivos está saturado
EPERM	El sistema de archivos que contiene el archivo especificado por <i>newpath</i> no soporta la creación de enlaces simbólicos, o el proceso no tiene acceso en escritura al directorio que contiene el archivo especificado por <i>newpath</i>
EROFS	El sistema de archivos es de lectura exclusiva

readlink permite a un proceso leer el nombre del archivo al que apunta un enlace simbólico. El nombre del enlace se especifica por el parámetro *pathname*, *buf* contiene la dirección de una memoria intermedia proporcionada por el proceso, y *bufsiz* especifica el número de bytes de esta memoria intermedia. *readlink* devuelve el número de bytes que se han colocado en la memoria *buf*, o el valor -1 en caso de error.

En caso de error, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
FAULT	<i>buf</i> o <i>pathname</i> contiene una dirección inválida
EINVAL	El archivo especificado por <i>pathname</i> no es un enlace simbólico
EIO	Se ha producido un error de entrada/salida
LOOP	Se ha encontrado un ciclo de enlaces simbólicos
NAMETOOLONG	<i>pathname</i> especifica un nombre de archivo demasiado largo
ENOENT	<i>pathname</i> se refiere a un nombre de archivo inexistente
NOTDIR	Uno de los componentes de <i>pathname</i> , utilizado como nombre de directorio, no lo es

3 Conceptos avanzados

3.1 i-nodos

Mientras que el usuario representa los archivos por nombres, absolutos o relativos, el núcleo Linux utiliza identificadores internos. Cada archivo es referenciado de forma única por dos números:

- el número de dispositivo: referencia un dispositivo, y por tanto el sistema de archivos que éste contiene;
- el número de archivo: referencia de manera única un archivo presente en el sistema de archivos.

Cada sistema de archivos, de tipo Unix, conserva en disco una tabla de descriptores de archivos. Estos descriptores, llamados i-nodos, contienen las informaciones de control utilizadas para gestionar los archivos, y especialmente los atributos de archivos así como las direcciones de los bloques de datos que los componen. El número de archivo único es utilizado por el núcleo como índice en la tabla de i-nodos, a fin de convertirlo en descriptor de archivo.

Cuando un proceso llama a una primitiva de sistema pasándole un nombre de archivo, el núcleo debe convertir el nombre especificado en descriptor de archivos. Para ello, Linux explora cada uno de los directorios contenidos en el nombre del archivo, y compara cada entrada de directorio con el nombre simple del elemento siguiente, conteniendo cada entrada de directorio un par (nombre de archivo, número de i-nodo), como puede verse en la figura 6.2. Por ejemplo, si se especifica el nombre */usr/src/linux/fs/dcache.c*, el núcleo efectúa las operaciones siguientes:

1. carga del i-nodo de la raíz (*/*), búsqueda de la entrada «*usr*»;
2. carga del i-nodo de */usr* obtenido en la etapa anterior, y búsqueda de la entrada «*src*»;
3. carga del i-nodo de */usr/src* obtenido en la etapa anterior, y búsqueda de la entrada «*linux*»;
4. carga del i-nodo de */usr/src/linux* obtenido en la etapa anterior, y búsqueda de la entrada «*fs*»;
5. carga del i-nodo de */usr/src/linux/fs* obtenido en la etapa anterior, y búsqueda de la entrada «*dcache.c*», lo que proporciona el número de i-nodo deseado.

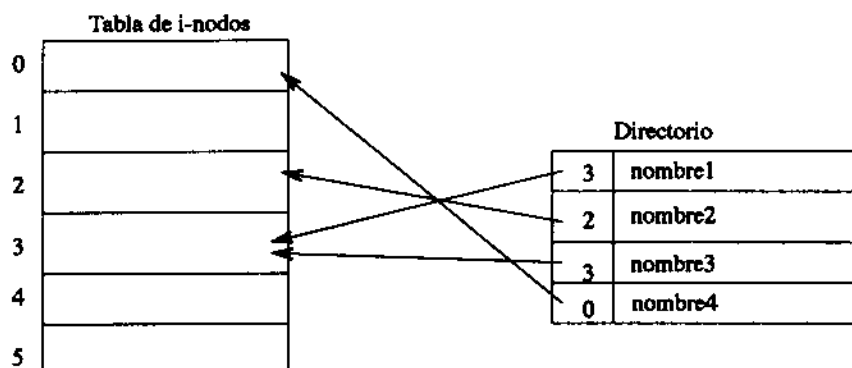


FIG. 6.2 – Formato de un directorio

3.2 Descriptores de entrada/salida

Los descriptores de entrada/salida devueltos por la primitiva *open* y utilizados por las llamadas al sistema de entrada/salida con archivos no tienen un especial significado para los procesos de usuario. El núcleo, por el contrario, conserva tablas de descripción de los archivos abiertos.

Linux gestiona varias tablas en memoria:

- el descriptor de cada proceso (estructura `task_struct`) apunta a una tabla de archivos abiertos por el proceso;
- cada uno de los elementos de esta tabla contiene un puntero a un elemento de una tabla que contiene la descripción de todos los archivos abiertos en el sistema;
- desde cada uno de los descriptores de archivos abiertos, un puntero envía a un elemento de la tabla de i-nodos correspondiente a los archivos en curso de utilización.

Cuando un proceso utiliza la primitiva *open* para abrir un archivo, el núcleo convierte el nombre del archivo en un par (número de dispositivo, número de i-nodo). Luego carga el i-nodo correspondiente en memoria si no está aún presente en la tabla de i-nodos. Linux asigna seguidamente un descriptor en la tabla de archivos abiertos, inicializa este descriptor, y hace que apunte al elemento de la tabla de i-nodos asignados en la etapa anterior. Finalmente, el núcleo asigna un descriptor en la tabla de archivos abiertos por el proceso y lo hace apuntar al descriptor de archivo abierto.

Cuando un proceso proporciona al núcleo un número de descriptor de entrada/salida, Linux utiliza este número como índice en la tabla de archivos abiertos por el proceso. Una indirección le permite acceder al lugar correspondiente en la tabla de archivos

abiertos, lo que le proporciona la entrada correspondiente de la tabla de i-nodos. Las informaciones contenidas en este último descriptor permiten al núcleo acceder al archivo en disco.

La figura 6.3 representa este mecanismo.

3.3 Compartir descriptors

Cuando un proceso abre un archivo, el núcleo asigna una entrada en la tabla de archivos abiertos, así como una entrada en la tabla de archivos del proceso. También carga el i-nodo del archivo en memoria si no está ya presente en la tabla de i-nodos.

Cuando un proceso crea un hijo, su tabla de archivos se duplica para el hijo. Los dos procesos pueden pues efectuar seguidamente entradas/salidas sobre los archivos que han sido abiertos por el proceso padre. El estado del archivo abierto (modo de apertura, posición actual, etc.) se almacena en la tabla de archivos abiertos, y se comparte pues entre los dos procesos. Toda modificación efectuada por uno de los procesos afecta al otro.

En el programa de ejemplo *CompartirDesc.c*, un proceso abre un archivo (el descriptor de entrada/salida se almacena en la variable *d1*) y crea un proceso hijo que abre otro archivo (el descriptor de entrada/salida se almacena en la variable *d2*).

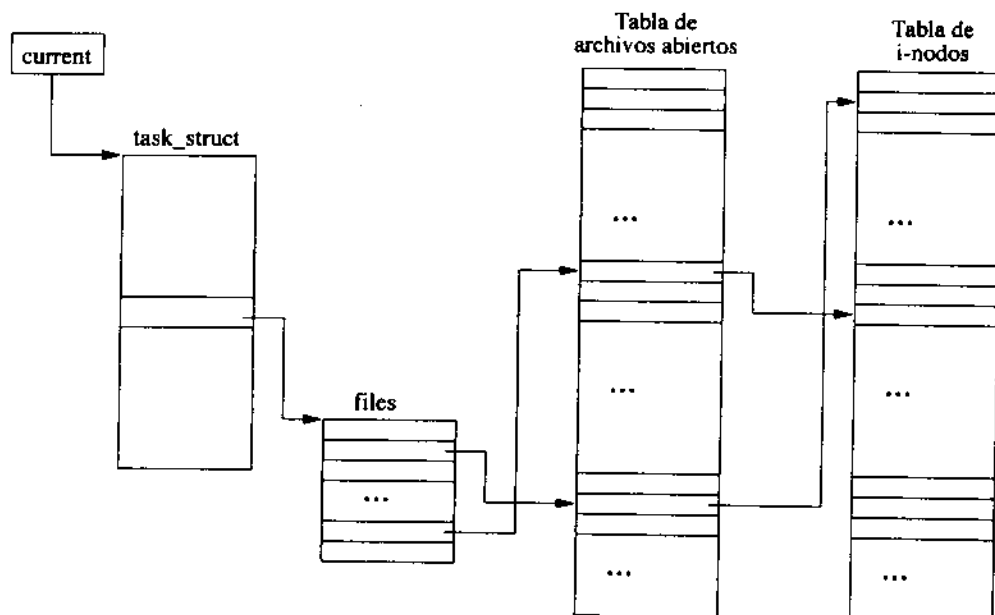


FIG 6.3 – Tablas de descriptors de entrada/salida

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

#define tratamiento_padre()      /**/
#define tratamiento_hijo()      /**/

void main (void)
{
    int pid;
    int d1;
    int d2;

    /* Apertura de un archivo por el padre */
    d1 = open ("/bin/sh", O_RDONLY);
    if (d1 == -1) {
        perror ("open (padre)");
        exit (1);
    }
    printf ("padre: posición actual = %ld\n",
            (long) lseek (d1, (off_t) 0, SEEK_CUR));
    /* Creación de un proceso hijo */
    pid = fork ();
    if (pid == -1) {
        perror ("fork");
        exit (1);
    } else if (pid == 0) {
        /* Proceso hijo */
        d2 = open ("/bin/ls", O_RDONLY);
        if (d2 == -1) {
            perror ("open (hijo)");
            exit (1);
        }
        printf ("hijo: posición actual = %ld\n",
                (long) lseek (d1, (off_t) 0, SEEK_CUR));
        (void) lseek (d1, (off_t) 1000, SEEK_SET);
        tratamiento_hijo();
    } else {
        /* Proceso padre */
        sleep (5);
        printf ("padre: posición actual = %ld\n",
                (long) lseek (d1, (off_t) 0, SEEK_CUR));
        tratamiento_padre();
    }
}
```

Tras estas operaciones, los dos procesos pueden acceder al archivo referenciado por d1. Sólo el proceso hijo puede acceder al archivo referenciado por d2. El estado de las tablas de descriptores se representa en la figura 6.4. Cuando el proceso hijo modifica la posición actual del primer archivo llamando a la primitiva *lseek*, esta modificación es inmediatamente visible a nivel del proceso padre.

```
bbj>CompartirDesc
padre: posición actual = 0
hijo: posición actual = 0
padre: posición actual = 1000
```

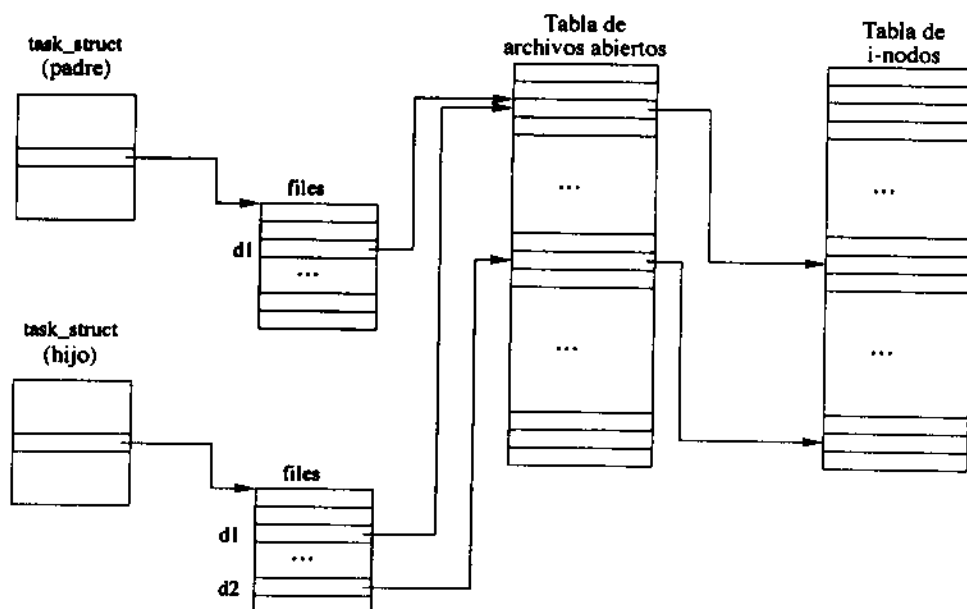


FIG. 6.4 – Compartir descriptores entre procesos emparentados

3.4 Bloqueo de archivos

En el caso en que varios procesos accedan de manera compartida a los mismos archivos, puede ser necesario sincronizar los procesos, es decir, serializar las lecturas y escrituras entre procesos a fin de evitar resultados aberrantes. Linux proporciona varios mecanismos de bloqueo de archivos que permiten sincronizar procesos cooperativos.

Los bloqueos ofrecidos por Linux sólo se aplican a procesos cooperativos. El núcleo no verifica estos bloqueos en las entradas/salidas clásicas, por las primitivas *read* y *write*, y a los procesos les corresponde llamar correctamente a las llamadas al sistema

de bloqueo. Si un proceso lee o escribe datos sin adquirir un bloqueo, Linux no lo impedirá.

Existen dos tipos de bloqueo:

1. Los bloqueos compartidos: varios procesos pueden poseer un bloqueo compartido simultáneamente. Este tipo de bloqueo es utilizado por procesos que efectúan únicamente lecturas en archivos que desean protegerse de un proceso que escribe.
2. Los bloqueos exclusivos: un solo proceso puede poseer un bloqueo exclusivo. Este tipo de bloqueo es utilizado por procesos que escriben concurrentemente y desean impedir varias escrituras simultáneas así como la lectura de datos durante una escritura.

3.5 Montaje de sistemas de archivos

Los archivos forman un árbol jerárquico compuesto por nodos (los directorios) y hojas (los demás tipos de archivos). Esta visión de árbol es una visión lógica que no tiene en cuenta la organización física de los datos en los discos. Aunque Linux presenta al usuario la visión de un solo árbol de archivos, es frecuente que esta jerarquía esté compuesta por varios sistemas de archivos situados sobre particiones o discos diferentes.

El ensamblado lógico de los diferentes sistemas de archivos se realiza por una operación de montaje. Esta última se efectúa generalmente al arrancar el sistema, pero también puede lanzarse durante el funcionamiento de Linux, por ejemplo para acceder a datos presentes en soportes removibles, como los CD-ROM.

La figura 6.5 representa una operación de montaje efectuada en la inicialización del sistema. En esta figura existen dos sistemas de archivos: el sistema de archivos raíz y el sistema de archivos que contiene la jerarquía */usr*. Cuando se efectúa el montaje, por el mandato *mount* que utiliza la llamada *mount*, el contenido del segundo sistema de archivos se vincula al sistema de archivos raíz y es accesible bajo el directorio */usr*.

3.6 Cuotas de disco

El espacio en disco no es ilimitado, y Linux proporciona un mecanismo de limitación: las cuotas de disco. En cada sistema de archivos montado, el administrador puede definir un límite para el número de archivos o de bloques asignados a un usuario o a un grupo de usuarios.

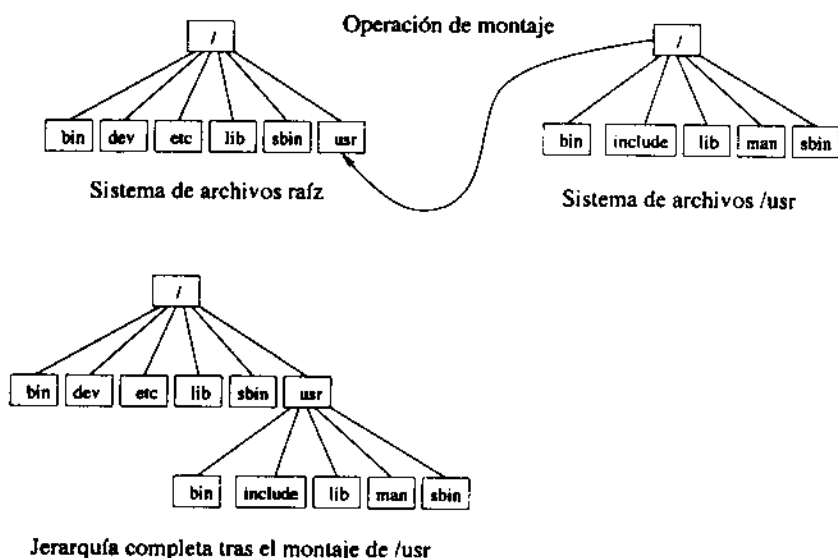


FIG. 6.5 – Montaje de sistema de archivos

Existen dos tipos de límites:

1. El límite absoluto: cuando se alcanza este límite, la asignación de nuevos archivos o bloques es rechazada por el núcleo.
2. El límite «flexible»: cuando este límite, que es inferior al límite absoluto, se alcanza, el núcleo muestra un mensaje de advertencia en las asignaciones de archivos o bloques, y se memoriza la fecha de la infracción. Al cabo de un cierto período (en principio siete días), Linux trata al usuario como si hubiera alcanzado el límite absoluto, es decir, rechaza toda asignación de recursos. El usuario debe suprimir entonces archivos a fin de ocupar menos espacio en disco que el límite «flexible».

Para cada usuario o grupo de usuarios, el administrador puede definir los límites vinculados al número de archivos poseídos (número de i-nodos asignados) y al número de bloques asignados. Estos límites deben ser definidos en cada sistema de archivos y se almacenan en los archivos *cuota.user* y *cuota.group*, situados en la raíz de los sistemas de archivos afectados.

4 Llamadas al sistema complementarias

4.1 Lectura y escritura de varias memorias intermedias

Las llamadas al sistema *read* y *write* permiten a un proceso leer o escribir datos en un archivo. Un proceso que quiera leer o escribir datos en o desde varias memorias intermedias debe utilizar varias llamadas a estas primitivas, lo que puede presentar varios inconvenientes:

- el proceso debe utilizar la misma llamada al sistema varias veces, lo que provoca varios cambios de modo de ejecución (de modo usuario a modo núcleo, y viceversa). El rendimiento es entonces limitado.
- mientras que la escritura por *write* es atómica, una serie de estas llamadas puede interrumpirse y los datos pueden ser escritos por otro proceso mientras el proceso actual está suspendido en medio de su secuencia de llamadas a *write*.

Por estas razones, Linux ofrece dos llamadas al sistema que permiten efectuar lecturas y escrituras a partir de varias memorias intermedias no contiguas en memoria:

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

int readv (int fd, const struct iovec *vector, size_t count);
int writev (int fd, const struct iovec *vector, size_t count);
```

La primitiva *readv* permite leer datos en varias memorias intermedias. El parámetro *vector* define las memorias intermedias a utilizar, con el número de memorias especificado en *count*. La operación efectuada es similar a la lectura de datos por *read*, excepto que los datos leídos se colocan en las memorias intermedias descritas por *vector*, en lugar de en una sola memoria intermedia contigua. El número de bytes leídos se devuelve en *readv*. En caso de fallo, se devuelve el valor -1.

Para las dos primitivas, el parámetro *iovec* contiene la dirección de una tabla de descriptores de memorias intermedias, cuyo número de elementos se indica por el parámetro *count*. La estructura *iovec*, definida en el archivo *<sys/uio.h>*, caracteriza a cada uno de los elementos de la tabla *iovec*:

tipo	campo	descripción
void *	iov_base	Dirección de la memoria intermedia
int	iov_len	Tamaño de la memoria intermedia en bytes

En la lectura de datos, el número de bytes especificado por `iovec[0].iov_len` se leen y colocan a partir de la dirección `iovec[0].iov_base`, luego el número de bytes especificado por `iovec[1].iov_len` se lee y coloca a partir de la dirección `iovec[1].iov_base`, y así sucesivamente.

En la escritura de datos, el número de bytes especificado por `iovec[0].iov_len` se escriben desde los datos situados a partir de la dirección `iovec[0].iov_base`, luego el número de bytes especificado por `iovec[1].iov_len` se escribe desde los datos situados a partir de la dirección `iovec[1].iov_base`, y así sucesivamente.

En caso de error de `readv` o de `writv`, la variable `errno` puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EBADF	El descriptor de entrada/salida especificado no es válido
EFAULT	vector o uno de los campos <code>iov_base</code> contiene una dirección no válida
EINTR	La llamada al sistema se ha interrumpido por la recepción de una señal
EINVAL	count es superior a <code>MAX_IOVEC</code> , o fd se refiere a un objeto sobre el cual no es posible la lectura o la escritura
EISDIR	fd se refiere a un directorio
EPIPE	fd se refiere a una tubería sobre la que no existe ya un proceso lector
ENOSPC	El sistema de archivos está saturado

4.2 Duplicación de descriptor de entrada/salida

Los descriptors de entrada/salida devueltos por la primitiva `open` y utilizados por todas las llamadas al sistema de entrada/salida pueden duplicarse. Ello significa que un proceso tiene la posibilidad de acceder al mismo archivo abierto por varios descriptors de entrada/salida. Se proporcionan dos llamadas al sistema para efectuar la duplicación de descriptors:

```
#include <unistd.h>

int dup (int oldfd);
int dup2 (int oldfd, int newfd);
```

La llamada `dup` duplica el descriptor `oldfd` y devuelve otro descriptor correspondiente al mismo archivo abierto. La primitiva `dup2` hace el descriptor `newfd` equivalente a `oldfd`. Si `newfd` correspondiera a un archivo abierto, este último se cierra antes de la duplicación. `dup` y `dup 2` devuelven el nuevo descriptor de entrada/salida, o el valor `-1` en caso de error.

En caso de error, la variable `errno` puede tomar los valores siguientes:

error	significado
EBADF	El descriptor de entradas/salidas especificado no es válido
EMFILE	Se ha alcanzado el número máximo de archivos abiertos por el proceso actual

Este mecanismo de duplicación de descriptores es particularmente interesante para las redirecciones. Un proceso puede redirigir su entrada o su salida estándar hacia un archivo y utilizar luego de manera transparente las funciones de la biblioteca estándar (*scanf* y *printf* por ejemplo), las lecturas y/o las escrituras se hacen en archivos y no desde el teclado o en la pantalla.

El fragmento de programa siguiente redirige su salida estándar hacia un archivo llamado *salida*:

```
int fd;
/* apertura del archivo de salida */
fd = open ("salida", O_WRONLY | O_CREAT | O_TRUNC);
if (fd == -1) {
    perror ("open");
    exit(1);
}
/* redirección de la salida estándar (descriptor 1) */
if (dup2 (fd, 1) != 1) {
    perror ("dup2");
    exit (1);
}
```

4.3 Atributos de archivos

Linux ofrece tres llamadas al sistema que permiten obtener los atributos de un archivo. Estas llamadas tienen la sintaxis siguiente:

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

int stat (const char *pathname, struct stat *buf);
int fstat (int fd, struct stat *buf);
int lstat (const char *pathname, struct stat *buf);
```

Las llamadas *stat* y *lstat* devuelven los atributos del archivo, cuyo nombre se pasa en el parámetro *pathname*, en la memoria intermedia cuya dirección se especifica por el parámetro *buf*. La diferencia entre las dos primitivas es que *stat* sigue los enlaces simbólicos, es decir, devuelve los atributos del archivo apuntado si es llamado con un nombre de enlace simbólico, mientras que *lstat* devuelve los atributos del propio enlace simbólico.

fstat permite obtener los atributos de un archivo abierto cuyo descriptor se pasa en el parámetro *fd*.

En caso de fallo, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EBADF	El descriptor de entrada/salida especificado no es válido
EFAULT	buf o pathname contiene una dirección no válida
ENAMETOOLONG	pathname especifica un nombre de archivo demasiado largo
ENOENT	pathname se refiere a un nombre de archivo inexistente
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de pathname, utilizado como nombre de directorio, no es un directorio

La estructura *stat* se define en el archivo de cabecera *<sys/stat.h>* y contiene los campos siguientes:

<i>tipo</i>	<i>campo</i>	<i>descripción</i>
<i>dev_t</i>	<i>st_dev</i>	Identificador del sistema de archivos
<i>ino_t</i>	<i>st_ino</i>	Número de i-nodo
<i>umode_t</i>	<i>st_mode</i>	Modo del archivo (tipo y derechos de acceso)
<i>nlink_t</i>	<i>st_nlink</i>	Número de enlaces
<i>uid_t</i>	<i>st_uid</i>	Identificador del usuario propietario
<i>gid_t</i>	<i>st_gid</i>	Identificador del grupo propietario
<i>dev_t</i>	<i>st_rdev</i>	Identificador de dispositivo en el caso de un archivo especial
<i>off_t</i>	<i>st_size</i>	Tamaño en bytes
unsigned long	<i>st_blksize</i>	Tamaño de los bloques en bytes
unsigned long	<i>st_blocks</i>	Número de bloques de 1 KB utilizados
<i>time_t</i>	<i>st_atime</i>	Fecha del último acceso
<i>time_t</i>	<i>st_mtime</i>	Fecha de la última modificación del contenido
<i>time_t</i>	<i>st_ctime</i>	Fecha de la última modificación

Los campos *st_atime*, *st_mtime* y *st_ctime* contienen fechas expresadas en número de segundos transcurridos desde el 1 de enero de 1970. Se utilizan generalmente las funciones proporcionadas por la biblioteca estándar (*asctime*, *ctime*, *gmtime*, *localtime* y *mktime*) para gestionarlas.

El campo *st_mode* contiene a la vez el tipo del archivo y sus derechos de acceso. El tipo puede probarse por macroinstrucciones definidas en *<sys/stat.h>*:

<i>macroinstrucción</i>	<i>significado</i>
S_ISLNK	Verdad si el archivo es un enlace simbólico, falso si no
S_ISREG	Verdad si el archivo es un archivo regular, falso si no
S_ISDIR	Verdad si el archivo es un directorio, falso si no
S_ISCHR	Verdad si el archivo es un archivo especial de carácter, falso si no
S_ISBLK	Verdad si el archivo es un archivo especial de bloque, falso si no
S_ISFIFO	Verdad si el archivo es una tubería con nombre, falso si no
S_ISSOCK	Verdad si el archivo es un <i>socket</i> , falso si no

Los derechos de acceso pueden probarse mediante constantes definidas en `<sys/stat.h>`:

<i>opción</i>	<i>significado</i>
S_ISUID	Bit <i>setuid</i>
S_ISGID	Bit <i>setgid</i>
S_ISVTX	Bit <i>sticky</i>
S_IRUSR	Derecho de lectura para el propietario
S_IWUSR	Derecho de escritura para el propietario
S_IXUSR	Derecho de ejecución para el propietario
S_IRWXU	Derecho de lectura, escritura y ejecución para el propietario
S_IRGRP	Derecho de lectura para el grupo de usuarios
S_IWGRP	Derecho de escritura para el grupo de usuarios
S_IXGRP	Derecho de ejecución para el grupo de usuarios
S_IRWXG	Derecho de lectura, escritura y ejecución para el grupo de usuarios
S_IROTH	Derecho de lectura para el resto de usuarios
S_IWOTH	Derecho de escritura para el resto de usuarios
S_IXOTH	Derecho de ejecución para el resto de usuarios
S_IRWXO	Derecho de lectura, escritura y ejecución para el resto de usuarios

Puede utilizarse una operación Y binaria (operador `&` del lenguaje C) para comprobar los derechos de acceso devueltos en `st_mode`. Por ejemplo, si se desea comprobar si el propietario tiene derechos de lectura y escritura en el archivo, se utilizará la prueba siguiente:

```
if ((st.st_mode & S_IRUSR) && (st.st_mode & S_IWUSR))
    /* el propietario tiene el derecho de leer y escribir */
```

El programa siguiente utiliza la primitiva `lstat` para mostrar el tamaño y el tipo de todos los archivos cuyos nombres se le pasan como argumentos:

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

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

    /* Bucle por los argumentos */
    for (i = 1; i < argc; i++) {
        if (lstat (argv[i], &st) == -1)
            perror (argv[i]);
        else {
            printf ("%s: tamaño = %d, tipo = ",
```

```
        argv[i], st.st_size);
if (S_ISREG (st.st_mode))
    printf ("archivo\n");
else if (S_ISDIR (st.st_mode))
    printf ("directorio\n");
else if (S_ISLNK (st.st_mode))
    printf ("enlace simbólico\n");
else if (S_ISCHR (st.st_mode))
    printf ("archivo especial carácter\n");
else if (S_ISBLK (st.st_mode))
    printf ("archivo especial bloque\n");
else if (S_ISFIFO (st.st_mode))
    printf ("tubería con nombre\n");
else if (S_ISSOCK (st.st_mode))
    printf ("socket\n");
else
    printf ("desconocido\n");
    }
}
exit (0);
}
```

4.4 Fechas asociadas a los archivos

Las fechas asociadas a cada archivo pueden modificarse: toda operación sobre un archivo puede actualizar una o varias fechas (*atime*, *ctime*, *mtime*). También es posible que un proceso modifique explícitamente las fechas *atime* y *mtime* utilizando las primitivas *utime* y *utimes*:

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

int utime (const char *pathname, struct utimbuf *buf);

#include <sys/time.h>

int utimes (char *pathname, struct timeval *tvp);
```

Estas dos primitivas modifican las fechas de último acceso y de última modificación del contenido del archivo, cuyo nombre se especifica por el parámetro *pathname*. *utime* utiliza un parámetro *buf* que apunta a una variable de tipo *utimbuf*. Esta estructura se define en el archivo de cabecera *<utime.h>* y contiene los campos siguientes:

tipo	campo	descripción
time_t	actime	Fecha del último acceso
time_t	mtime	Fecha de la última modificación del contenido

Por su parte, la primitiva *utimes* utiliza un parámetro *tvp* que contiene la dirección de una tabla de dos elementos de tipo *timeval*. El primer elemento de la tabla (*tvp[0]*) representa la fecha de último acceso, y el segundo elemento (*tvp[1]*) representa la fecha de última modificación del contenido del archivo. La estructura *timeval* se define en el archivo de cabecera `<sys/time.h>` y contiene los campos siguientes:

tipo	campo	descripción
long	tv_sec	Número de segundos
long	tv_usec	Número de microsegundos

En caso de error en la ejecución de las primitivas *utime* y *utimes*, la variable *errno* puede tomar los valores siguientes:

error	significado
EACCES	El proceso no tiene acceso en escritura al archivo especificado por <i>pathname</i>
EFAULT	<i>buf</i> , <i>tvp</i> o <i>pathname</i> contiene una dirección no válida
ENAMETOOLONG	<i>pathname</i> especifica un nombre de archivo demasiado largo
ENOENT	<i>pathname</i> se refiere a un nombre de archivo inexistente
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de <i>pathname</i> , utilizado como nombre de directorio, no lo es

4.5 Propiedades de los archivos abiertos

Linux proporciona la primitiva *fcntl* que permite efectuar operaciones diversas y variadas sobre un archivo abierto:

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

int fcntl (int fd, int cmd);
int fcntl (int fd, int cmd, long arg);
```

La operación realizada depende del parámetro *cmd*. Se definen diversas constantes en el archivo `<fcntl.h>`:

- **F_DUPFD**: es el equivalente de la primitiva *dup2*. El descriptor de entrada/salida *fd* se duplica en el descriptor *arg*.
- **F_GETFD**: devuelve el valor del indicador «close-on-exec». Si este indicador tiene el valor nulo, el archivo se mantiene abierto si el proceso actual llama a una primitiva de tipo *exec* para ejecutar un nuevo programa, si no el archivo se cierra automáticamente con la llamada a *exec*.

- **F_SETFD**: posiciona el indicador «close-on-exec».
- **F_GETFL**: devuelve las opciones utilizadas en la apertura del archivo (parámetro *flags* de la primitiva *open*).
- **F_SETFL**: modifica las opciones de apertura. Sólo las opciones **O_APPEND** y **O_NONBLOCK** pueden estar posicionadas.
- **F_GETOWN**: devuelve el número del proceso o del grupo de procesos propietario de un *socket*. Se devuelve un número de grupo en forma de un valor negativo.
- **F_SETOWN**: posiciona el número del proceso o del grupo de procesos propietario de un *socket*.

fcntl acepta también otras opciones relativas al bloqueo de archivos. Estas opciones se detallan en la sección 4.7.2.

En caso de fallo, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EBADF	El descriptor de entrada/salida especificado no es válido
EINVAL	<i>cmd</i> o <i>arg</i> especifica un valor no válido
EMFILE	Se ha alcanzado el número máximo de archivos abiertos por el proceso actual (en el caso de la petición F_DUPFD)

4.6 Control del proceso *bdflush*

El contenido de las memorias intermedias modificadas se reescribe periódicamente en disco. Dos procesos se encargan de esta reescritura: *update*, que ejecuta la primitiva *sync* cada treinta segundos a fin de reescribir el contenido de todas las memorias modificadas, y *bdflush*. Este último es un proceso interno al núcleo creado automáticamente en el arranque del sistema: efectúa las reescrituras de memorias modificadas teniendo en cuenta las prioridades asociadas a las memorias. Una memoria que contenga una estructura de control de un sistema de archivos es, por ejemplo, más prioritaria que una memoria intermedia que contenga datos.

El proceso *bdflush* es ejecutado automáticamente por el núcleo cuando surge la necesidad. Existe una llamada al sistema que permite especificar los parámetros de *bdflush*. Esta llamada al sistema no se incluye en la biblioteca C y hay que declararla explícitamente:

```
#include <syscall.h>
```

```
_syscall2(int bdflush, int func, long data);
```


Esta declaración corresponde al prototipo siguiente:

```
int bdflush (int func, long data);
```

El parámetro *func* especifica la función a ejecutar. Si es igual a 1, se efectúa una reescritura de memorias modificadas. Si *func* es superior o igual a 2, su valor se interpreta como sigue:

- si es par, el parámetro número $(func-2)/2$ se devuelve en el entero largo del que *data* especifica la dirección en memoria;
- si es impar, *data* especifica el valor a asignar al parámetro número $(func-3)/2$.

Los parámetros de *bdflush* se numeran de 0 a 8:

- 0: porcentaje máximo de memorias intermedias modificadas: cuando este porcentaje se sobrepasa, *bdflush* se activa para reescribir memorias intermedias en disco.
- 1: número máximo de memorias intermedias a reescribir en cada activación de *bdflush*.
- 2: número de memorias intermedias a colocar en la lista de memorias intermedias disponibles, cada vez que esta lista se reconstruye.
- 3: número de memorias intermedias modificadas a partir del que *bdflush* debe activarse cuando la lista de memorias disponibles se reconstruye. Este parámetro no lo utiliza Linux 2.0.
- 4: porcentaje de memorias intermedias a explorar en la búsqueda de *cluster* disponible (véase sección 6.2.8). Este parámetro no lo utiliza Linux 2.0.
- 5: tiempo durante el cual una memoria intermedia está autorizada a conservar sus modificaciones antes de ser reescrito en disco. Este tiempo se expresa en número de ciclos de reloj.
- 6: tiempo durante el cual una memoria intermedia que contenga un descriptor de sistema de archivos está autorizado a conservar sus modificaciones antes de ser reescrito en disco. Este tiempo se expresa en número de ciclos de reloj.
- 7: constante utilizada para calcular el porcentaje de utilización de memorias intermedias asociadas a los bloques de un cierto tamaño.
- 8: este parámetro no es utilizado por Linux 2.0.

En caso de fallo, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EFAULT	data contiene una dirección inválida
EINVAL	func o data especifica un valor no válido
EPERM	El proceso que llama no posee los privilegios necesarios

4.7 Bloqueo

4.7.1 Bloqueo de un archivo

La primitiva *flock* permite adquirir un bloqueo sobre un archivo completo. Su sintaxis es la siguiente:

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

int flock (int fd, int operation);
```

flock bloquea o desbloquea el acceso al archivo cuyo descriptor se pasa en el parámetro *fd*. El parámetro *operation* especifica la operación a efectuar. Se definen varias constantes en *<sys/file.h>*:

<i>opción</i>	<i>significado</i>
LOCK_SH	Solicita un bloqueo compartido sobre el archivo. Varios procesos pueden tener un bloqueo compartido simultáneamente
LOCK_EX	Solicita un bloqueo exclusivo. Un solo proceso puede tener un bloqueo exclusivo
LOCK_UN	Libera un bloqueo exclusivo o compartido

En el caso de una petición de bloqueo, el proceso que llama puede ser suspendido. Sin embargo, puede especificar la opción **LOCK_NB** para no ser suspendido; la primitiva *flock* devuelve entonces un error si el bloqueo no puede realizarse.

En caso de error, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EBADF	El descriptor de entrada/salida especificado no es válido
EINVAL	operation contiene un valor no válido
ENOLCK	El archivo está bloqueado y se ha especificado la opción LOCK_NB

4.7.2 Bloqueo de una sección de un archivo

La primitiva *fcntl*, ya detallada en la sección 4.5, permite bloquear una parte de un archivo. Para ello, el argumento *arg* debe contener la dirección de una estructura *flock*. Esta estructura está definida en el archivo *<fcntl.h>* y contiene los campos siguientes:

tipo	campo	descripción
off_t	l_start	Ubicación del bloqueo en el archivo
off_t	l_len	Longitud de la zona bloqueada
short	l_type	Tipo de bloqueo
short	l_whence	Tipo de la ubicación (SEEK_SET , SEEK_CUR o SEEK_END)
pid_t	l_pid	Número del proceso que ha adquirido el bloqueo

Están disponibles varias operaciones:

- **F_SETLK**: pone o quita un bloqueo según las informaciones pasadas en la estructura `flock`. El tipo puede ser **F_RDLCK** para adquirir un bloqueo compartido, **F_WRLCK** para adquirir un bloqueo exclusivo, y **F_UNLCK** para liberar un bloqueo. Si el bloqueo no puede obtenerse inmediatamente, `fcntl` devuelve el error **EACCESS**.
- **F_SETLKW**: similar a **F_SETLK**. La única diferencia es que el proceso que llama se suspende en espera de bloqueo si no puede realizarlo inmediatamente.
- **F_GETLK**: obtiene la descripción de bloqueos existentes. Si existe ya un bloqueo que entra en conflicto con el especificado en la estructura `flock`, el contenido de ésta se modifica y pasa a contener las informaciones que caracterizan el bloqueo existente.

La función `lockf` ofrece una interfaz simplificada respecto a las posibilidades de bloqueo de `fcntl`. Su sintaxis es la siguiente:

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

```
int lockf (int fd, int cmd, off_t len);
```

El parámetro `fd` especifica el descriptor del archivo, `len` representa el tamaño del bloqueo respecto a la posición actual, y `cmd` puede tomar los valores siguientes:

opción	significado
F_ULOCK	Desbloquea una sección anteriormente bloqueada
F_TLOCK	Bloquea una sección. Si el bloqueo no puede obtenerse inmediatamente, se devuelve el error EACCESS
F_LOCK	Bloquea una sección con suspensión eventual del proceso que llama en espera del bloqueo
F_TEST	Comprueba la presencia de un bloqueo sobre la sección especificada: si existe un bloqueo, se devuelve el error EACCESS

4.8 Montaje de sistemas de archivos

Linux proporciona dos llamadas al sistema que permiten montar y desmontar sistemas de archivos. En otras palabras, los sistemas de archivos se conectan y desconectan lógicamente al árbol de archivos. Estas llamadas se reservan para los procesos que poseen los derechos del superusuario.

La sintaxis de las primitivas *mount* y *umount* es la siguiente:

```
#include <sys/mount.h>
#include <linux/fs.h>

int mount (const char *specialfile, const char *dir,
           const char *filesystemtype, unsigned long rwflag,
           const void *data);

int umount (const char *specialfile);
int umount (const char *dir);
```

La llamada al sistema *mount* monta el sistema de archivos presente en el dispositivo cuyo nombre se pasa en el parámetro *specialfile*. El parámetro *dir* indica el nombre del punto de montaje, es decir, el nombre del directorio a partir del cual el sistema de archivos debe ser accesible. El tipo del sistema de archivos se pasa en el parámetro *filesystemtype*, y se trata de una cadena de caracteres que representa un tipo de sistema de archivos conocido por el núcleo Linux, como «minix», «ext2», «proc» o «iso9660». Los parámetros *rwflag* y *data* especifican las opciones de montaje y sólo se tienen en cuenta si los 16 bits de mayor peso de *rwflag* son iguales al valor 0xC0ED.

Las opciones de montaje posibles se definen en el archivo de cabecera *<linux/fs.h>* en forma de constantes:

opción	significado
MS_RDONLY	Montaje del sistema de archivos en lectura exclusiva
MS_NOSUID	No se usan los bits <i>setuid</i> y <i>setgid</i>
MS_NODEV	No hay acceso a los archivos especiales
MS_NOEXEC	Prohibición de ejecutar programas
MS_SYNCHRONOUS	Escrituras síncronas

Es posible combinar estas diferentes opciones mediante una operación O binaria (operador «|» del lenguaje C). Una constante particular, **MS_REMOUNT**, puede utilizarse para modificar las opciones de montaje de un sistema de archivos montado anteriormente.

El parámetro *data* apunta a una cadena de caracteres que contiene opciones suplementarias. El contenido de esta cadena es dependiente del tipo de sistema de archivos.

En caso de error, la variable *errno* puede tomar los valores siguientes:

error	significado
EBUSY	El dispositivo especificado por <i>specialfile</i> ya está montado
EFAULT	<i>specialfile</i> , <i>dir</i> , <i>filesystemtype</i> o <i>data</i> contiene una dirección no válida
ENAMETOOLONG	<i>specialfile</i> o <i>dir</i> especifica un nombre de archivo demasiado largo
ENODEV	El tipo de sistema de archivos especificado por <i>filesystemtype</i> no es soportado por el núcleo

ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENOENT	specialfile o dir se refiere a un nombre de archivo inexistente
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTBLK	specialfile no especifica un nombre de archivo especial
ENOTDIR	Uno de los componentes de specialfile o dir no especifica un nombre de directorio
EPERM	El proceso no posee los privilegios necesarios

La primitiva *umount* desmonta un sistema de archivos montado anteriormente por una llamada a *mount*. Acepta como parámetro tanto un nombre de archivo especial (parámetro *specialfile*) como un nombre de punto de montaje (parámetro *dir*).

En caso de error, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EBUSY	El dispositivo especificado contiene archivos abiertos
EFAULT	specialfile o dir contiene una dirección no válida
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENAMETOOLONG	specialfile o dir especifica un nombre de archivo demasiado largo
ENOENT	specialfile o dir se refiere a un nombre de archivo inexistente
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTBLK	specialfile no especifica un nombre de archivo especial
ENOTDIR	Uno de los componentes de specialfile o dir, utilizado como nombre de directorio, no lo es, o dir no especifica un nombre de directorio
EPERM	El proceso no posee los privilegios necesarios

4.9 Informaciones sobre un sistema de archivos

Las primitivas *statfs* y *fstatfs* permiten obtener las estadísticas de uso de un sistema de archivos. Su sintaxis es la siguiente:

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

```
int statfs (const char *pathname, struct statfs *buf);
int fstatfs (int fd, struct statfs *buf);
```

statfs obtiene las estadísticas de uso del sistema de archivos que contiene el archivo cuyo nombre se pasa en el parámetro *pathname*; *fstatfs* utiliza un descriptor de entrada/salida especificado por el parámetro *fd*. Las dos primitivas devuelven las informaciones en la variable apuntada por el parámetro *buf*. El tipo de esta variable es una estructura *statfs*. Esta estructura se define en el archivo de cabecera *<sys/vfs.h>* y contiene los campos siguientes:

<i>tipo</i>	<i>campo</i>	<i>descripción</i>
long	<i>f_type</i>	Tipo del sistema de archivos
long	<i>f_bsize</i>	Tamaño en bytes de los bloques a utilizar para entradas/salidas óptimas
long	<i>f_blocks</i>	Número total de bloques de datos

long	f_bfree	Número de bloques no asignados
long	f_bavail	Número de bloques disponibles para un usuario no privilegiado
long	f_files	Número total de i-nodos
long	f_ffree	Número de i-nodos disponibles
fsid_t	f_fsid	Identificador del sistema de archivos
long	f_namelen	Tamaño máximo de los nombres de archivos
long [6]	f_spare	No utilizados

En caso de error, la variable `errno` puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EFAULT	<code>buf</code> o <code>pathname</code> contiene una dirección no válida
EIO	Se ha producido un error de entrada/salida
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENAMETOOLONG	<code>pathname</code> especifica un nombre de archivo demasiado largo
ENOENT	<code>pathname</code> se refiere a un nombre de archivo inexistente
ENOTDIR	Uno de los componentes de <code>pathname</code> , utilizado como nombre de directorio, no lo es

Además, la primitiva `fstatfs` puede devolver el error `EBADF`, indicando que `fd` especifica un descriptor de entrada/salida no válido.

Otra primitiva, `ustat`, permite obtener informaciones similares, pero se implementa únicamente para asegurar una cierta compatibilidad con Unix System V; además, se recomienda usar `statfs`. Por ello no la detallaremos aquí.

4.10 Información sobre los tipos de sistemas de archivos soportados

La primitiva `sysfs` permite conocer los tipos de sistemas de archivos soportados por el núcleo. Su sintaxis es la siguiente:

```
int sysfs (int opcion, const char *fsname);
int sysfs (int opcion, unsigned int fs_index, char *buf);
int sysfs (int opcion);
```

Su sintaxis de uso es dependiente del valor del parámetro `opcion`:

- si `opcion` tiene el valor 1, `sysfs` devuelve el índice del tipo de sistema de archivos especificado por `fsname`;
- si `opcion` vale 2, `sysfs` devuelve el nombre del tipo de sistema de archivos correspondiente al índice especificado por `fs_index` en la memoria intermedia apuntada por `buf`;

- si *option* tiene el valor 3, *sysfs* devuelve el número de tipos de sistemas de archivos soportados por el núcleo.

4.11 Manipulación de cuotas de disco

Las cuotas de disco asociadas a los usuarios pueden manipularse con la primitiva *quotactl*:

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

int quotactl (int cmd, const char *special, int id, caddr_t addr);
```

El tratamiento efectuado por *quotactl* depende del parámetro *cmd*. Este parámetro se inicializa por la macroinstrucción *QCMD(subcmd, type)*, donde *type* puede ser *USRQUOTA* o *GRPQUOTA*; los valores posibles de *subcmd* se detallan a continuación. El parámetro *special* especifica el nombre del dispositivo que contiene el sistema de archivos afectado. Este sistema de archivos debe estar previamente montado, *id* contiene el identificador del usuario o del grupo de usuarios al que se aplica la operación, y el parámetro *addr* debe contener la dirección de una estructura de datos dependiente de la operación.

Los diferentes valores de *subcmd* son los siguientes:

opción	significado
Q_QUOTAON	Activación de las cuotas en el sistema de archivos especificado por <i>special</i> . <i>addr</i> debe contener el nombre del archivo que contiene las cuotas (normalmente <i>quota.user</i> o <i>quota.group</i>)
Q_QUOTAOFF	Desactivación de las cuotas en el sistema de archivos especificado por <i>special</i>
Q_GETQUOTA	Obtención de los límites y el estado actual de los bloques y archivos asignados por el usuario o el grupo de usuarios que se indica por <i>id</i> en el sistema de archivos especificado por <i>special</i>
Q_SETQUOTA	Modificación de los límites y el estado actual de los bloques y archivos asignados por el usuario o el grupo de usuarios, que se indica por <i>id</i> en el sistema de archivos especificado por <i>special</i>
Q_SETQLIM	Modificación de los límites asignados por el usuario o el grupo de usuarios, que se indica por <i>id</i> en el sistema de archivos especificado por <i>special</i>
Q_SETUSE	Modificación del estado actual de los bloques y archivos asignados por el usuario o el grupo de usuarios, que se indica por <i>id</i> en el sistema de archivos especificado por <i>special</i>
Q_SYNC	Reescritura de las estructuras de control de cuotas, guardadas en memoria por el núcleo, en disco
Q_GETSTATS	Obtención de estadísticas

En el caso de las opciones `Q_GETQUOTA`, `Q_SETQUOTA`, `Q_SETQLIM` y `Q_SETUSE`, el parámetro `arg` debe contener la dirección de una variable del tipo estructura `dqblk`. Esta estructura se define en el archivo `<sys/quota.h>` y contiene los campos siguientes:

tipo	campo	descripción
unsigned long	<code>dqb_bhardlimit</code>	Límite absoluto en el número de bloques asignados
unsigned long	<code>dqb_bsoftlimit</code>	Límite «suave» en el número de bloques asignados
unsigned long	<code>dqb_curblocks</code>	Número de bloques asignados
unsigned long	<code>dqb_ihardlimit</code>	Límite absoluto en el número de archivos asignados
unsigned long	<code>dqb_isoftlimit</code>	Límite «suave» en el número de archivos asignados
unsigned long	<code>dqb_curinodes</code>	Número de archivos asignados
time_t	<code>dqb_btime</code>	Lapso de espera utilizado cuando se sobrepasa el límite «suave» en el número de bloques
time_t	<code>dqb_itime</code>	Lapso de espera utilizado cuando se sobrepasa el límite «suave» en el número de i-nodos

La operación `Q_GETSTATS` permite obtener estadísticas sobre el funcionamiento interno de las cuotas. El parámetro `arg` debe contener la dirección de una variable de tipo estructura `dqstats`. Esta estructura se define en la sección 5.2.7.

El proceso que llama generalmente debe poseer los derechos del superusuario para utilizar la primitiva `quotactl`. Sólo las operaciones `Q_GETSTATS`, `Q_SYNC` y `Q_GETQUOTA`, con `id` correspondiente a la identidad del usuario que ejecuta el proceso actual, pueden ser ejecutadas por un proceso no privilegiado.

En caso de error, la variable `errno` puede tomar los valores siguientes:

error	significado
EINVAL	El archivo de definición de cuotas es incorrecto
EBUSY	<code>Q_QUOTAON</code> se ha especificado sobre un sistema de archivos en que ya se han activado las cuotas
EFAULT	<code>addr</code> contiene una dirección no válida
EINVAL	<code>type</code> contiene un valor incorrecto
EIO	Se ha producido un error de entrada/salida
EMFILE	Se ha alcanzado el número máximo de archivos abiertos para el proceso actual
ENODEV	<code>special</code> no contiene el número de un dispositivo en el que se haya montado un sistema de archivos
ENOPRG	El núcleo no ha sido compilado con el soporte de cuotas
ENOTBLK	<code>special</code> no contiene el nombre de un archivo especial en modo bloque
EPERM	El proceso que llama no posee los derechos necesarios
ESRCH	Se ha especificado un sistema de archivos en el que no se han activado las cuotas

5 Presentación general de la implementación

5.1 El sistema virtual de archivos

5.1.1 Principio

Linux soporta varios tipos de sistemas de archivos, ya se trate de sistemas de archivos nativos como Ext2 o de sistemas que permiten acceder a los datos de otros sistemas operativos como Minix o MS/DOS. A fin de permitir a los procesos el acceso de manera uniforme a los archivos, sea cual sea el tipo de sistema de archivos que los contiene, el núcleo posee una capa lógica cuyo objetivo es asegurar la interfaz entre las llamadas al sistema respecto a los archivos y el código de gestión de archivos propiamente dicho. Esta capa se llama el sistema virtual de archivos (SVA). El término original es *Virtual File System* (VFS).

Cuando un proceso efectúa una llamada al sistema de archivos, esta llamada se dirige al SVA. Este último se encarga de efectuar los tratamientos independientes del formato del sistema de archivos afectado, y redirige la petición hacia el módulo que gestiona el archivo, como lo muestra la figura 6.6.

El principio del SVA no es propio de Linux: también se ha utilizado en otros sistemas [Kleiman 1986], pero la implementación de Linux es diferente.

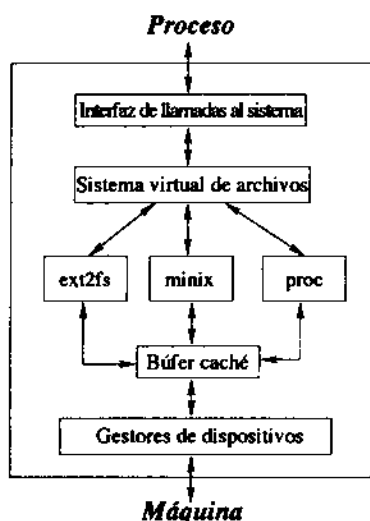


FIG. 6.6 – El sistema virtual de archivos

5.1.2 Operaciones aseguradas por el SVA

El SVA proporciona funcionalidades independientes de los sistemas de archivos físicos, e implementa, por ejemplo:

- El *caché* de nombres:

Cada conversión de un nombre de archivo en números de dispositivo y de i-nodo es costosa, porque necesita una exploración iterativa de directorios. Por razones de rendimiento, el SVA implementa un *caché* de nombres de archivo: cuando un nombre de archivo se convierte en números de dispositivo e i-nodo, el resultado se memoriza en una lista. En las llamadas siguientes con el mismo nombre de archivo, el resultado guardado puede reutilizarse sin proceder de nuevo a la exploración de directorios. El funcionamiento interno de este *caché* se explica en la sección 6.1.5.

- El *búfer* *caché*:

Linux mantiene una lista de memorias intermedias en curso de utilización. Al realizar una lectura de bloque un sistema de archivo, el contenido del bloque se guarda en una memoria intermedia, llamada un *búfer*. Esta memoria se guarda mientras el bloque está en curso de utilización, y mientras el espacio de memoria que ocupa no se necesita para otro *búfer*. Al realizarse una modificación de datos en un bloque, el cambio se efectúa en el contenido del *búfer*, que se marca como modificado pero no se escribe en disco inmediatamente. A intervalos regulares, el proceso *update* llama a la primitiva *sync* para forzar la reescritura de todos los *búfers* modificados. Este principio permite economizar numerosas entradas/salidas en disco:

- en una lectura de bloque, el SVA verifica primero que el bloque no ha sido ya leído. Si el bloque ha sido ya cargado en memoria, su contenido es accesible en una memoria intermedia, y no es necesaria ninguna entrada/salida;
- en modificaciones sucesivas de datos en el mismo bloque, los cambios se efectúan sobre el contenido de la memoria intermedia correspondiente, y no es necesario reescribir el bloque con cada modificación: todos los cambios se escriben en disco en la reescritura de la memoria intermedia.

El funcionamiento interno del *búfer* *caché* se detalla en la sección 6.2.

Las funciones internas del SVA aseguran también las partes comunes de las llamadas al sistema que operan sobre archivos. En su mayor parte, estas llamadas comparten el mismo algoritmo:

Verificación de argumentos

Conversión de nombre(s) de archivo(s) en número de dispositivo e i-nodo

Verificación de los permisos

Llamada a la función del tipo de sistema de archivos correspondiente

A fin de llamar a las funciones correspondientes a las diferentes acciones, el SVA utiliza una aproximación orientada a objetos: a cada sistema de archivos montado, archivo abierto, e i-nodo en curso de utilización se les asignan operaciones implementadas en el código del sistema de archivos correspondiente. Se puede ver cada entidad como un objeto de una cierta clase, a la que se asocian métodos, redefinidos en cada módulo que contiene la implementación de un tipo de sistema de archivos. Como el núcleo Linux está programado en lenguaje C, esta orientación a objetos se implementa asociando un descriptor a cada objeto. Este descriptor contiene una serie de punteros a funciones.

Se definen cuatro conjuntos de operaciones:

- Operaciones sobre sistemas de archivos: son las operaciones dependientes del formato físico del sistema de archivos, como la lectura o la escritura de un i-nodo. Estas operaciones se detallan en la sección 5.3.1.
- Operaciones sobre i-nodos en curso de utilización: son las operaciones directamente vinculadas a los i-nodos, como la supresión de un archivo. Estas operaciones se detallan en la sección 5.3.2.
- Operaciones sobre archivos abiertos: son las operaciones correspondientes a primitivas de entrada/salida sobre archivos, como la lectura o la escritura de datos. Estas operaciones se detallan en la sección 5.3.3.
- Operaciones sobre cuotas: son las operaciones llamadas para validar la asignación de bloques e i-nodos. Estas operaciones se detallan en la sección 5.3.4.

5.2 Estructuras del SVA

El SVA utiliza ciertas estructuras genéricas para describir los sistemas de archivos y los archivos en curso de utilización.

5.2.1 Tipos de sistemas de archivos

Cada tipo de sistema de archivos se describe por la estructura `file_system_type`. Esta estructura se define en `<linux/fs.h>` y contiene los campos siguientes:

tipo	campo	descripción
<code>struct super_block(*)()</code>	<code>read_super</code>	Función de inicialización
<code>const char *</code>	<code>name</code>	Nombre del tipo de sistema de archivos

int	requires_dev	Booleano que indica si el sistema de archivos está vinculado a un dispositivo ffsico
struct file_system_type*	next	Puntero de encadenamiento

Al inicializar el sistema, cada tipo de sistema de archivos cuyo soporte ha sido compilado en el núcleo, se registra en el SVA, llamando a la función `register_filesystem`. Esta función registra el descriptor del tipo de sistema de archivos en una lista encadenada referenciada por la variable `file_systems`. Al montar un sistema de archivos, el SVA explora esta lista encadenada buscando el descriptor del tipo de sistema de archivos solicitado, y luego llama a la función `read_super` correspondiente.

5.2.2 Sistemas de archivos montados

El núcleo mantiene dos listas diferentes que referencian los sistemas de archivos montados. La tabla `super_blocks` contiene `NR_SUPER` descriptors utilizados por el SVA para todas las operaciones de entrada/salida.

El tipo de estos descriptors es la estructura `super_block`. Esta estructura se define en el archivo de cabecera `<linux/fs.h>` y contiene los campos siguientes:

tipo	campo	descripción
kdev_t	s_dev	Identificador del dispositivo
unsigned long	s_blocksize	Tamaño de bloque en bytes
unsigned long	s_blocksize_bits	Tamaño de bloque en bytes expresado en potencias de 2
unsigned char	s_lock	Indicador de bloqueo
unsigned char	s_rd_only	Indicador de lectura exclusiva
unsigned char	s_dirt	Indicador de modificación
struct file_system_type*	s_type	Puntero al descriptor de tipo de sistemas de archivos correspondiente
struct super_operations*	s_op	Operaciones relacionadas con el sistema de archivos (véase sección 5.3.1)
struct dquot_operations*	dq_op	Operaciones relacionadas con las cuotas de disco (véase sección 5.3.4)
unsigned long	s_flags	Opciones de montaje
unsigned long	s_magic	Firma: todo tipo de sistema de archivos posee un número «mágico» que le sirve de firma: este número permite reconocer la presencia de un sistema de archivos de ese tipo en una partición
struct inode *	s_covered	Puntero al descriptor de i-nodo del punto de montaje
struct inode *	s_mounted	Puntero al descriptor de i-nodo del directorio raíz del sistema de archivos

struct wait_queue *	s_wait	Variable utilizada para sincronizar los accesos concurrentes al descriptor
union	u	Informaciones dependientes del tipo de sistema de archivos

La segunda lista mantenida por el núcleo memoriza las correspondencias entre los nombres de dispositivos sobre los que se encuentran los sistemas de archivos y los nombres de puntos de montaje. El tipo de los elementos de esta lista es la estructura `vfsmount`. Esta estructura está definida en el archivo `<linux/mount.h>` y contiene los campos siguientes:

tipo	campo	descripción
kdev_t	mnt_dev	Identificador del dispositivo
char *	mnt_devname	Nombre del archivo especial que representa el dispositivo
char *	mnt_dirname	Nombre del punto de montaje
unsigned int	mnt_flags	Opciones de montaje
struct semaphore	mnt_sem	Semáforo utilizado para bloquear el descriptor
struct super_block*	mnt_sb	Puntero al descriptor de superbloque correspondiente
struct file *	mnt_quotas	Descriptores de archivos de descripción de cuotas
[MAXQUOTAS]		
time_t [MAXQUOTAS]	mnt_iexp	Lapso de espera utilizado en el desbordamiento de la cuota de i-nodos
time_t [MAXQUOTAS]	mnt_bexp	Lapso de espera utilizado en el desbordamiento de la cuota de bloques
struct vfsmount *	mnt_next	Puntero al descriptor siguiente en la lista

5.2.3 I-nodos en curso de utilización

Linux utiliza un descriptor de i-nodo para referenciar cada archivo en curso de utilización. El tipo de estos descriptores es la estructura `inode`. Esta estructura, definida en el archivo `<linux/fs.h>`, contiene los campos siguientes:

tipo	campo	descripción
kdev_t	i_dev	Identificador del dispositivo
unsigned long	i_ino	Número de i-nodo
umode_t	i_mode	Modo del archivo (tipo y derechos de acceso)
nlink_t	i_nlink	Número de enlaces
iud_t	i_uid	Identificador del usuario propietario
gid_t	i_gid	Identificador del grupo propietario
kdev_t	i_rdev	Identificador de dispositivo si el i-nodo representa un archivo especial
off_t	i_size	Tamaño del archivo en bytes
time_t	i_atime	Fecha de último acceso
time_t	i_mtime	Fecha de última modificación del contenido
time_t	i_ctime	Fecha de última modificación del i-nodo
unsigned long	i_blksize	Tamaño de los bloques en bytes
unsigned long	i_blocks	Número de bloques de 512 bytes asignados al archivo

unsigned long	i_version	Número de versión incrementado automáticamente con cada nuevo uso
unsigned long	i_nrpages	Número de páginas cargadas en memoria de este archivo
struct semaphore	i_sem	Semáforo utilizado para serializar los accesos concurrentes al archivo
struct inode_operations *	i_op	Operaciones vinculadas al i-nodo (véase la sección 5.3.2)
struct super_block *	i_sb	Puntero al descriptor del sistema de archivos correspondiente
struct wait_queue *	i_wait	Variable utilizada para sincronizar los accesos concurrentes al i-nodo
struct file_lock *	i_flock	Puntero a los descriptores de bloqueos asociados al i-nodo
struct vm_area_struct *	i_map	Puntero a los descriptores de secciones del i-nodo proyectados en memoria
struct page *	i_pages	Punteros a los descriptores de páginas del i-nodo proyectados en memoria
struct dquot * [MAXQUOTAS]	i_dquot	Punteros a los descriptores de cuotas de disco asociados al i-nodo
struct inode *	i_next	Puntero al i-nodo siguiente en la lista
struct inode *	i_prev	Puntero al i-nodo anterior en la lista
struct inode *	i_hash_next	Puntero al i-nodo siguiente en la lista de <i>hash</i>
struct inode *	i_hash_prev	Puntero al i-nodo anterior en la lista de <i>hash</i>
struct inode *	i_mount	Puntero al i-nodo raíz de un sistema de archivos en el caso de un punto de montaje
unsigned short	i_count	Número de usos del i-nodo
unsigned short	i_flags	Opciones de montaje del sistema de archivos que contienen el i-nodo
unsigned char	i_lock	Booleano que indica si el i-nodo está bloqueado en memoria
unsigned char	i_dirt	Booleano que indica si el i-nodo ha sido modificado
unsigned char	i_pipe	Booleano que indica si el i-nodo corresponde a una tubería
unsigned char	i_sock	Booleano que indica si el i-nodo corresponde a un <i>socket</i>
unsigned short	i_writcount	Número de aperturas en escritura de este archivo
union	u	Información dependiente del tipo de sistema de archivos. Esta variable contiene también un campo, <i>generic_ip</i> , de tipo <i>void*</i> , que puede usarse para guardar la dirección de datos privados

Los descriptores de i-nodos se encadenan en varias listas:

- una lista global, cuya dirección del primer elemento se almacena en la variable *first_inode*, que contiene todos los descriptores de i-nodos;
- varias listas de *hashing*: una función de *hashing*, que actúa sobre el identificador de dispositivo y el número de i-nodo, permite colocar los descriptores en listas dife-

rentes de tamaño más reducido que la lista global. De este modo, la búsqueda de un descriptor en una lista es más rápida que en la lista global.

5.2.4 Archivos abiertos

A cada archivo abierto en el sistema corresponde un descriptor. La estructura `file` se define en el archivo de cabecera `<linux/fs.h>` y contiene los campos siguientes:

tipo	campo	descripción
<code>mode_t</code>	<code>f_mode</code>	Modo de apertura del archivo modificado (ver seguidamente)
<code>loff_t</code>	<code>f_pos</code>	Posición actual en bytes desde el inicio del archivo
<code>unsigned long</code>	<code>f_flags</code>	Modo de apertura del archivo especificado en la llamada a <code>open</code>
<code>unsigned long</code>	<code>f_reada</code>	Número de bloques a leer de manera anticipada
<code>unsigned long</code>	<code>f_ramax</code>	Número máximo de bloques a leer de manera anticipada
<code>unsigned long</code>	<code>f_raend</code>	Posición del primer byte en el archivo, tras la última página leída de manera anticipada
<code>unsigned long</code>	<code>f_ralen</code>	Tamaño en bytes del último grupo de datos leídos de manera anticipada
<code>unsigned long</code>	<code>f_rawin</code>	Tamaño de la ventana de lectura anticipada
<code>struct file *</code>	<code>f_next</code>	Puntero de encadenamiento al descriptor siguiente
<code>struct file *</code>	<code>f_prev</code>	Puntero de encadenamiento al descriptor anterior
<code>int</code>	<code>f_owner</code>	Número de proceso o grupo de procesos propietario de un <code>socket</code>
<code>struct inode *</code>	<code>f_inode</code>	Puntero al descriptor del i-nodo correspondiente
<code>struct file_operations*</code>	<code>f_op</code>	Operaciones relacionadas con el archivo abierto (véase la sección 5.3.3)
<code>unsigned long</code>	<code>f_version</code>	Número de versión incrementado en cada uso del descriptor
<code>void *</code>	<code>private_data</code>	Puntero a una zona de datos privada dependiente del módulo de gestión del archivo

El campo `f_mode` es un poco particular: se basa en el modo de apertura especificado en la llamada a `open` y está formado por la conjunción de las constantes `FMODE_READ` y `FMODE_WRITE`, que indican respectivamente si son posibles la lectura y la escritura en ese archivo.

Los descriptors de archivos se encadenan en una lista global, cuya dirección del primer elemento se almacena en la variable `first_file`, que contiene todos los descriptors de archivos.

5.2.5 Archivos abiertos por un proceso

A cada proceso se le asocia una tabla de descriptors locales. Esta tabla se referencia por el campo `files` de la estructura `task_struct`. La definición de esta tabla (estructura `files_struct`), que está contenida en el archivo de cabecera `<linux/sched.h>`, es la siguiente:

tipo	campo	descripción
int	count	Número de descriptores de archivos asociados
fd_set	close_on_exec	Indicadores «close-on-exec» agrupados en una cadena de bits
struct file * [NR_OPEN]	fd	Punteros a los descriptores de archivos abiertos

Otra estructura describe las informaciones de gestión de archivos para cada proceso; se trata de la estructura `fs_struct`, definida en el archivo `<linux/sched.h>`. Esta estructura contiene los campos siguientes:

tipo	campo	descripción
int	count	Número de procesos que referencian este descriptor
unsigned short	umask	Derechos de acceso predeterminados utilizados en la creación de archivos
struct inode *	root	Descriptor del i-nodo correspondiente a la raíz del sistema de archivos para el proceso
struct inode *	pwd	Descriptor del i-nodo correspondiente al directorio actual del proceso

5.2.6 Descriptores de bloqueos

Linux mantiene una lista global de bloqueos asociados a los archivos. Los bloqueos asociados a un i-nodo se encadenan también y se referencian por el campo `i_flock` del descriptor de i-nodo.

El tipo de descriptores de bloqueos (estructura `file_lock`) se define en el archivo de cabecera `<linux/fs.h>` y contiene los campos siguientes:

tipo	campo	descripción
struct file_lock *	fl_next	Puntero al descriptor de bloqueo siguiente en la lista asociada a un i-nodo
struct file_lock *	fl_nextlink	Puntero al descriptor de bloqueo siguiente en la lista global
struct file_lock *	fl_prevlink	Puntero al descriptor de bloqueo anterior en la lista global
struct file_lock *	fl_block	Puntero a la lista de bloqueos en espera de actuar
struct task_lock *	fl_owner	Puntero al descriptor de procesos que han creado el bloqueo
struct wait_queue *	fl_queue	Cola de espera en la que se registran los procesos que solicitan bloqueos
struct file *	fl_file	Puntero al descriptor de archivo al que está asociado el bloqueo
char	fl_flags	Modo de bloqueo (ver más adelante)
char	fl_type	Tipo de bloqueo (ver más adelante)
off_t	fl_start	Índice del primer byte tocado por el bloqueo
off_t	fl_end	Índice del último byte tocado por el bloqueo

El campo `fl_flags` especifica el modo de bloqueo, y puede tomar los valores siguientes:

constante	significado
F_FLOCK	El bloqueo se ha creado con la primitiva <code>flock</code> . Todo el archivo está bloqueado. El bloqueo se asocia a un descriptor de archivo, y puede compartirse entre un proceso padre y su hijo. El bloqueo se suprime cuando se cierra el último descriptor referido al archivo, o cuando se suprime explícitamente
F_POSIX	El bloqueo se ha creado por la primitiva <code>fcntl</code> o <code>lockf</code> . Especifica que una parte del archivo está bloqueada y se asocia a un proceso

El campo `fl_type` contiene el tipo del bloqueo, y puede contener los valores `F_RDLCK`, `F_WRLCK` y `F_UNLCK`.

5.2.7 Descriptores de cuotas de disco

Las cuotas de disco se gestionan mediante descriptores de cuotas. Cada uno de estos descriptores se asocia a un usuario o a un grupo particular. La estructura `dquot`, definida en el archivo de cabecera `<linux/quota.h>`, posee los campos siguientes:

tipo	campo	descripción
unsigned int	<code>dq_id</code>	Identificador de usuario o grupo a quien se aplica esta cuota
short	<code>dq_type</code>	Tipo de cuota, <code>USERQUOTA</code> o <code>GRPQUOTA</code>
kdev_t	<code>dq_dev</code>	Identificador de dispositivo
short	<code>dq_flags</code>	Estado del descriptor (ver más adelante)
short	<code>dq_count</code>	Número de usos del descriptor
struct vfsmount *	<code>dq_mnt</code>	Puntero al descriptor de sistema de archivos montado afectado
struct dqblk	<code>dq_dqb</code>	Límites y uso
struct wait_queue *	<code>dq_wait</code>	Variable utilizada para sincronizar los accesos concurrentes al descriptor
struct dquot *	<code>dq_prev</code>	Puntero al descriptor anterior en la lista
struct dquot *	<code>dq_next</code>	Puntero al descriptor siguiente en la lista
struct dquot *	<code>dq_hash_prev</code>	Puntero al descriptor anterior en la lista de <i>hash</i>
struct dquot *	<code>dq_hash_next</code>	Puntero al descriptor siguiente en la lista de <i>hash</i>

El valor del campo `dq_flags` está formado por la combinación de las constantes siguientes:

opción	significado
DQ_LOCKED	El descriptor de cuota está bloqueado
DQ_WAIT	Hay un proceso en espera de disponibilidad de este descriptor de cuota
DQ_MOD	El descriptor de cuota ha sido modificado
DQ_BLK	Se ha emitido una advertencia de desbordamiento del límite en el número de bloques
DQ_INODES	Se ha emitido una advertencia de desbordamiento del límite en el número de i-nodos
DQ_FAKE	El descriptor de cuota no contiene límite

De la misma manera que los descriptores de i-nodos, los descriptores de cuotas se encadenan en varias listas: una lista global que contiene todos los descriptores, y listas de *hashing* utilizadas para acelerar las búsquedas.

El módulo de gestión de cuotas en disco mantiene estadísticas sobre la gestión de descriptores de cuotas. Para ello, la estructura `dqstats`, definida en el archivo de cabecera `<linux/quota.h>`, contiene los campos siguientes:

tipo	campo	descripción
_u32	lookups	Número de veces que se ha buscado un descriptor de cuota en la lista
_u32	drops	Número de veces que se ha liberado un descriptor de cuota
_u32	reads	Número de veces que un descriptor de cuota se ha leído del disco
_u32	writes	Número de veces que un descriptor de cuota se ha escrito en disco
_u32	cache_hits	Número de veces que un descriptor de cuota se ha encontrado en la lista en lugar de ser leído del disco
_u32	pages_allocated	Número de páginas asignadas para descriptores de cuota
_u32	allocated_dquots	Número de descriptores de cuota asignados
_u32	free_dquots	Número de descriptores de cuota asignados pero no usados
_u32	syncs	Número de veces que los descriptores de cuota se han reescrito por una operación <code>O_SYNC</code>

5.2.8 Memorias intermedias del búfer caché

El *búfer caché* gestiona las memorias intermedias asociadas a los bloques en disco. La estructura `buffer_head`, declarada en el archivo `<linux/fs.h>`, define el formato de descriptores de memorias intermedias. Contiene los campos siguientes:

tipo	campo	descripción
unsigned long	b_blocknr	Número de bloque en el dispositivo
kdev_t	b_dev	Identificador del dispositivo lógico
kdev_t	b_rdev	Identificador del dispositivo físico
unsigned long	b_rsector	Número de sector de inicio del bloque en el dispositivo físico
struct buffer_head *	b_next	Puntero al descriptor siguiente
struct buffer_head *	b_this_page	Puntero al descriptor de la memoria intermedia siguiente cuyo contenido está en la misma página de memoria
unsigned long	b_state	Estado de la memoria intermedia (véase más adelante)
struct buffer_head *	b_next_free	Puntero al descriptor libre siguiente
unsigned int	b_count	Número de utilizaciones de este bloque
unsigned long	b_size	Tamaño del bloque en bytes
char *	b_data	Puntero al contenido de la memoria intermedia
unsigned int	b_list	Lista en la que se encuentra la memoria intermedia (véase más adelante)
unsigned long	b_flush_time	Hora en la que el contenido de la memoria intermedia debe escribirse en disco

unsigned long	b_lru_time	Hora en la que el contenido de la memoria intermedia se ha usado por última vez
struct wait_queue *	b_wait	Variable utilizada para sincronizar los accesos concurrentes a esta memoria intermedia
struct buffer_head *	b_prev	Puntero al descriptor anterior
struct buffer_head *	b_prev_free	Puntero al descriptor libre anterior
struct buffer_head *	b_reqnext	Puntero a la memoria intermedia siguiente que forma parte de la misma petición de entrada/salida (véase el capítulo 7, sección 4.2)

El valor del campo `b_state` está formado por la combinación de bits referenciados por las constantes siguientes, definidas en el archivo de cabecera `<linux/fs.h>`:

opción	significado
BH_Uptodate	La memoria intermedia contiene datos válidos
BH_Dirty	Los datos contenidos en la memoria intermedia han sido modificados
BH_Lock	La memoria intermedia se está utilizando y está bloqueada
BH_Req	Si este bit está a 0, la memoria intermedia se ha invalidado
BH_Touched	La memoria intermedia se ha utilizado recientemente
BH_Protected	La memoria intermedia no debe reutilizarse
BH_FreeOnIO	La memoria intermedia se ha asignado de manera temporal para efectuar una entrada/salida y debe liberarse tras el fin de la entrada/salida

Las funciones `clear_bit`, `set_bit` y `test_bit` se utilizan para acceder a este campo. Varias macroinstrucciones que permiten probar los diferentes indicadores de este campo se definen en `<linux/fs.h>`:

macroinstrucción	significado
buffer_uptodate	Comprobación del bit BH_Uptodate
buffer_dirty	Comprobación del bit BH_Dirty
buffer_locked	Comprobación del bit BH_Locked
buffer_req	Comprobación del bit BH_Req
buffer_touched	Comprobación del bit BH_Touched
buffer_has_aged	Comprobación del bit BH_Has_aged
buffer_protected	Comprobación del bit BH_Protected

Las memorias intermedias se colocan en varias listas encadenadas según su estado y su contenido. Las constantes siguientes, definidas en el archivo `<linux/fs.h>`, representan estas listas:

opción	significado
BUF_CLEAN	Lista de memorias intermedias no modificadas
BUF_UNSHARED	Lista de memorias intermedias que se han compartido pero ya no lo están
BUF_LOCKED	Lista de memorias intermedias a reescribir en disco
BUF_LOCKED1	Lista de memorias intermedias que contienen metadatos, es decir, estructuras de control de los sistemas de archivos, como los i-nodos, los superbloques, etc.
BUF_DIRTY	Lista de memorias intermedias modificadas
BUF_SHARED	Lista de memorias intermedias compartidas

Además, una función de hashing, basada en el identificador de dispositivo y el número de bloque, se utiliza para colocar las memorias intermedias en varias listas de *hashing*. De este modo, la búsqueda de una memoria intermedia es más rápida.

5.2.9 Caché de nombres

El caché de nombres funciona utilizando dos niveles:

- Una lista, de nivel 1, en la que se añaden las entradas obtenidas al convertir el nombre de archivo en número de i-nodo o en la lectura de directorios por la llamada *readdir*. Esta lista se almacena en la tabla *level1_cache*, y el puntero *level1_head* contiene la dirección del primer elemento.
- Una lista, de nivel 2, a la que se añaden las entradas cuando se encuentran en la primera lista. Esta lista se almacena en la tabla *level2_cache*, y el puntero *level2_head* contiene la dirección del primer elemento.

Cada una de estas listas se gestiona en LRU (*Least Recently Used*): las nuevas entradas se añaden al principio de la lista. Una entrada no utilizada se «desplaza» así poco a poco hacia la cola de la lista y se elimina cuando alcanza el fin.

A fin de acelerar las búsquedas en las dos listas, se emplea la técnica de *hashing* siguiente: una función de *hashing* combina el número de dispositivo, el número de i-nodo del directorio y el nombre de la entrada. Mediante esta función, las entradas de las dos listas se distribuyen en 32 listas con *hashing*, en las que los punteros están contenidos en la tabla *hash_table*.

Las estructuras de la lista están definidas en el archivo fuente *fs/dcache.c*, así como las funciones que las manipulan. Una entrada de las listas (estructura *dir_cache_entry*) contiene los campos siguientes:

tipo	campo	descripción
struct hash_list	h	Puntero a la lista que contiene la entrada
kdev_t	dc_dev	Identificador del dispositivo
unsigned long	dir	Número de i-nodo del directorio
unsigned long	version	Número de versión del directorio
unsigned long	ino	Número de i-nodo del archivo
unsigned char	name_len	Tamaño del nombre de la entrada en bytes
char [DCACHE_NAME_LEN]	name	Nombre de archivo contenido en la entrada de directorio
struct dir_cache_entry **	lru_head	Dirección del puntero de cabecera de la lista que contiene la entrada, es decir, <i>level1_head</i> o <i>level2_head</i>
struct dir_cache_entry *	next_lru	Puntero al elemento siguiente en la lista LRU
struct dir_cache_entry *	prev_lru	Puntero al elemento anterior en la lista LRU

La estructura `hash_list` contiene dos punteros utilizados para asegurar el encadenamiento en las listas con *hashing*:

tipo	campo	descripción
<code>struct dir_cache_entry *</code>	<code>next</code>	Puntero al elemento siguiente en la lista de <i>hash</i>
<code>struct dir_cache_entry *</code>	<code>prev</code>	Puntero al elemento anterior en la lista de <i>hash</i>

5.3 Operaciones genéricas

5.3.1 Operaciones sobre superbloques

La estructura `super_operations` contiene punteros a funciones que son llamadas por el SVA y son implementadas en el código de los sistemas de archivos. Los campos de esta estructura son los siguientes:

- `void (*read_inode) (struct inode *inode)`

La operación `read_inode` es llamada por el SVA cuando un i-nodo debe ser leído desde un sistema de archivos. Los campos `i_dev` e `i_ino` del parámetro `inode` se inicializan por el SVA antes de la llamada.

- `int (*notify_change) (struct inode *inode, struct iattr *iattr)`

La operación `notify_change` es llamada por el SVA cuando los atributos de un i-nodo han sido modificados. El parámetro `iattr` indica las modificaciones efectuadas, su estructura se describe más adelante.

- `void (*write_inode) (struct inode *inode)`

La operación `write_inode` es llamada por el SVA cuando un i-nodo debe escribirse en un sistema de archivos.

- `void (*put_inode) (struct inode *inode)`

La operación `put_inode` es llamada por el SVA cuando un i-nodo deja de utilizarse.

- `void (*put_super) (struct super_block *super)`

La operación `put_super` es llamada por el SVA cuando un superbloque deja de utilizarse, es decir, cuando el sistema de archivos correspondiente se desmonta.

- `void (*write_super) (struct super_block *super)`

La operación `write_super` es llamada por el SVA cuando el contenido de un superbloque ha sido modificado y debe reescribirse en disco, por ejemplo en la llamada al sistema `sync`.

- `void (*statfs) (struct super_block *super, struct statfs *buf, int bufsize)`

La operación `statfs` es llamada por el SVA para obtener las informaciones de control del sistema de archivos. El parámetro `buf` apunta a una variable de tipo estructura `statfs` cuya definición se da en la sección 4.9.

- `int (*remount_fs) (struct super_block *super, int *flags, char *data)`

La operación `remount_fs` se llama cuando el sistema de archivos se «remonta», es decir, cuando las opciones de montaje se modifican por una llamada a la primitiva `mount` con la opción `MS_REMOUNT`.

La estructura `iattr`, utilizada en la operación `notify_change`, se define en el archivo `<linux/fs.h>` y contiene los campos siguientes:

tipo	campo	descripción
unsigned int	<code>ia_valid</code>	Indicación de modificaciones efectuadas (véase más adelante)
<code>umode_t</code>	<code>ia_mode</code>	Nuevo modo (tipo y derechos de acceso)
<code>uid_t</code>	<code>ia_uid</code>	Nuevo identificador de propietario
<code>gid_t</code>	<code>ia_gid</code>	Nuevo identificador de grupo
<code>off_t</code>	<code>ia_size</code>	Nuevo tamaño en bytes
<code>time_t</code>	<code>ia_atime</code>	Nueva fecha de último acceso
<code>time_t</code>	<code>ia_mtime</code>	Nueva fecha de última modificación del contenido
<code>time_t</code>	<code>ia_ctime</code>	Nueva fecha de última modificación del i-nodo

El campo `ia_valid` indica las modificaciones a tener en cuenta. Está formado por la combinación, mediante una operación O binaria, de las constantes siguientes, que están definidas en `<linux/fs.h>`:

opción	significado
<code>ATTR_MODE</code>	El modo ha sido modificado
<code>ATTR_UID</code>	El identificador del propietario se ha modificado
<code>ATTR_GID</code>	El identificador del grupo se ha modificado
<code>ATTR_SIZE</code>	El tamaño ha sido modificado
<code>ATTR_ATIME</code>	La fecha de último acceso del contenido se ha modificado
<code>ATTR_MTIME</code>	La fecha de última modificación del contenido se ha modificado
<code>ATTR_CTIME</code>	La fecha de de última modificación del i-nodo se ha modificado
<code>ATTR_ATIME_SET</code>	La fecha de último acceso se ha modificado llamando a <code>utime</code> o <code>utimes</code>
<code>ATTR_MTIME_SET</code>	La fecha de última modificación se ha modificado llamando a <code>utime</code> o <code>utimes</code>
<code>ATTR_FORCE</code>	No se ha efectuado ninguna verificación de la validez de las modificaciones

5.3.2 Operaciones sobre i-nodos

La estructura `inode_operations` contiene punteros a funciones que son llamadas por el SVA e implementadas en el código de los sistemas de archivos. Los campos de esta estructura son los siguientes:

- `struct file_operations *default_file_ops;`

Este campo proporciona la dirección de las operaciones sobre archivos asociados del i-nodo (véase la sección 5.3.3) a utilizar de modo predeterminado al abrir un archivo.

- `int (*create) (struct inode *dir, const char *name, int len, int mode, struct inode **result)`

La operación `create` es llamada por el SVA para crear una nueva entrada en el directorio referenciado por el parámetro `dir`. El nombre de la entrada a crear es especificado por los parámetros `name` y `len`. `mode` indica el modo (tipo y derechos de acceso) del archivo a crear. El i-nodo correspondiente al archivo creado se devuelve en el parámetro `result`.

- `int (*lookup) (struct inode *dir, const char *name, int len, struct inode **result)`

La operación `lookup` es llamada por el SVA para efectuar la búsqueda de una entrada en el directorio referenciado por el parámetro `dir`. El nombre de la entrada a buscar es especificado por los parámetros `name` y `len`. El i-nodo correspondiente a la entrada encontrada se devuelve en el parámetro `result`.

- `int (*link) (struct inode *inode, struct inode *dir, const char *name, int len)`

La operación `link` es llamada para crear un enlace a un i-nodo referenciado por el parámetro `inode`. El nombre del enlace a crear se especifica por los parámetros `name` y `len`, y el directorio en el que el enlace debe crearse se indica por el parámetro `dir`.

- `int (*unlink) (struct inode *dir, const char *name, int len)`

La operación `unlink` se llama para suprimir una entrada en el directorio referenciado por el parámetro `dir`. El nombre de la entrada a suprimir se especifica por los parámetros `name` y `len`.

- `int (*symlink) (struct inode *dir, const char *name, int len, const char *symname)`

La operación `symlink` se llama para crear un enlace simbólico en el directorio referenciado por el parámetro `dir`. Los parámetros `name` y `len` especifican el nombre del enlace a crear. El nombre del destino se indica en `symname`, en forma de una cadena de caracteres terminada por un carácter nulo.

- `int (*mkdir) (struct inode *dir, const char *name, int len, int mode)`

La operación `mkdir` es llamada por el SVA para crear un subdirectorio en el directorio referenciado por el parámetro `dir`. El nombre del subdirectorio a crear se especifica por los parámetros `name` y `len`. `mode` indica los derechos de acceso del directorio a crear.

- `int (*rmdir) (struct inode *dir, const char *name, int len)`

La operación `rmdir` se llama para suprimir un subdirectorio en el directorio referenciado por el parámetro `dir`. El nombre del subdirectorio a suprimir se especifica por los parámetros `name` y `len`.

- `int (*mknod) (struct inode *inode, const char *name, int len, int mode, int rdev)`

La operación `mknod` es llamada por el SVA para crear un archivo especial en el directorio referenciado por el parámetro `dir`. El nombre del archivo a crear se especifica por los parámetros `name` y `len`. `mode` indica el tipo y los derechos de acceso del directorio a crear, y `rdev` contiene el identificador de dispositivo correspondiente al archivo especial.

- `int (*rename) (struct inode *old_dir, const char *old_name, int old_len, struct inode *new_dir, const char *new_name, int new_len)`

La operación `rename` se llama para renombrar una entrada de directorio. Los parámetros `old_dir`, `old_name` y `old_len` especifican el nombre de la entrada a renombrar, mientras que el nuevo nombre se indica en los parámetros `new_dir`, `new_name` y `new_len`.

- `int (*readlink) (struct inode *inode, char *buf, int bufsize)`

La operación `readlink` se llama para leer el contenido del enlace simbólico referenciado por el parámetro `inode`. El resultado se devuelve en la memoria intermedia apuntada por `buf`, cuya longitud en bytes se indica en `bufsize`.

- `int (*follow_link) (struct inode *dir, struct inode *inode, int flag, int mode, struct inode **result)`

La operación `follow_link` se llama para resolver un enlace simbólico referenciado por el parámetro `inode`. El parámetro `dir` especifica el i-nodo del directorio a partir del cual debe efectuarse la interpretación. El i-nodo resultante se devuelve en `result`.

- `int (*readpage) (struct inode *inode, struct page *page)`

La operación `readpage` se llama para leer el contenido de una página de memoria desde el archivo correspondiente al parámetro `inode`.

- `int (*writepage) (struct inode *inode, struct page *page)`

La operación `writepage` se llama para escribir el contenido de una página de memoria en el archivo correspondiente al parámetro `inode`.

- `int (*bmap) (struct inode *inode, int block)`

La operación `bmap` se llama para obtener el número de bloque físico (es decir, el número de bloque en disco) correspondiente al bloque lógico, indicado por el parámetro `block`, del archivo referenciado por el parámetro `inode`.

- `void (*truncate) (struct inode *inode)`

La operación `truncate` se llama para modificar el tamaño del archivo referenciado por el parámetro `inode`. El SVA modifica el campo `i_size` del i-nodo antes de llamar esta operación.

- `int (*permission) (struct inode *inode, int perm)`

La operación `permission` se llama para verificar que el proceso que llama posee derechos de acceso suficientes sobre el archivo referenciado por el parámetro `inode`. El parámetro `perm` indica los derechos de acceso deseados.

- `int (*smmap) (struct inode *inode, int sector)`

La operación `smmap` es similar a `bmap`, pero manipula números de sectores físicos en disco en lugar de números de bloques. Se define a fin de permitir leer y escribir páginas en memoria desde un sistema de archivos, de tipo MS/DOS.

5.3.3 Operaciones sobre archivos abiertos

La estructura `file_operations` contiene punteros a funciones que son llamadas por el SVA y están implementadas en el código de los sistemas de archivos. Los campos de esta estructura son los siguientes:

- `int (*lseek) (struct inode *inode, struct file *file, off_t offset, int whence)`

La operación `lseek` se llama para modificar la posición actual del archivo referenciado por el parámetro `file`. El parámetro `offset` especifica el desplazamiento expresado en bytes respecto a la base indicada por `whence`.

- `int (*read) (struct inode *inode, struct *file, char *buf, int count)`

La operación `read` se llama para leer datos desde el archivo referenciado por el parámetro `file`. El parámetro `buf` especifica la dirección de la memoria intermedia donde deben almacenarse los datos, y `count` indica el número de bytes a leer.

- `int (*write) (struct inode *inode, struct file *file, const char *buf, int bufsize)`

La operación `write` se llama para escribir datos en el archivo referenciado por el parámetro `file`. El parámetro `buf` especifica la dirección de la memoria intermedia donde se almacenan los datos a escribir, y `count` indica el número de bytes a escribir.

- `int (*readdir) (struct inode *inode, struct file *file, void *dirent, filldir_t filldir)`

La operación `readdir` se llama para leer entradas desde el directorio referenciado por el parámetro `file`. `dirent` contiene la dirección de una memoria intermedia donde el resultado debe guardarse, y el parámetro `filldir` es un puntero a una función que guarda el resultado (véase la sección 6.3.5).

- `int (*select) (struct inode *inode, struct file *file, int sel_type, select_table *wait)`

La operación `select` permite efectuar multiplexado sobre varios descriptores de entradas/salidas, con la ejecución de la primitiva *select* (véase el capítulo 7, sección 2.3). Ésta debe devolver el valor 1 si la entrada/salida especificada por `sel_type` es posible sobre el descriptor de archivo indicado. Si esta entrada/salida no es posible, debe añadir una entrada de la tabla de multiplexado `wait` en una cola de espera y devolver el valor 0. Su uso se detalla en el capítulo 7, sección 4.4.3.

- `int (*ioctl) (struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)`

La operación `ioctl` se llama para realizar una operación de control sobre un dispositivo (véase el capítulo 7, sección 2.4).

- `int (*mmap) (struct inode *inode, struct file *file, struct vm_area_struct *vma)`

La operación `mmap` se llama para proyectar en memoria el contenido de un archivo. Se detalla en el capítulo 8.

- `int (*open) (struct inode *inode, struct file *file)`

La operación `open` se llama en la apertura del archivo referenciado por el parámetro `file`.

- `void (*release) (struct inode *inode, struct file *file)`

La operación `release` se llama en el último cierre del archivo referenciado por el parámetro `file`.

- `int (*fsync) (struct inode *inode, struct file *file)`

La operación `fsync` se llama para reescribir los bloques modificados en el archivo referenciado por el parámetro `file`.

- `int (*fasync) (struct inode *inode, struct file *file, int on)`

La operación `fasync` se llama cuando un proceso utiliza la llamada al sistema `fcntl` para activar o desactivar las entradas/salidas asíncronas sobre el archivo especificado por el parámetro `file`. Cuando las entradas/salidas asíncronas están activas, la señal `SIGIO` se envía al proceso al final de la lectura o la escritura de datos. Sólo ciertos gestores de dispositivos y la capa `socket` de los protocolos de red implementan esta operación.

- `int (*check_media_change) (kdev_t dev)`

La operación `check_media_change` se utiliza para verificar si el dispositivo removible (disquete o CD-ROM, por ejemplo) especificado por el parámetro `dev` ha sido cambiado por el usuario. Debe devolver el valor 0 si el dispositivo no ha cambiado, y un valor no nulo en caso contrario.

- `int (*revalidate) (kdev_t dev)`

La operación `revalidate` se llama tras un cambio de dispositivo removible especificado por el parámetro `dev`. Debe efectuar las operaciones necesarias para adaptarse al nuevo dispositivo (por ejemplo, el gestor de disquete deberá efectuar una detección del formato del nuevo disquete).

5.3.4 Operaciones sobre cuotas de disco

La estructura `dquot_operations` contiene punteros a funciones llamadas por el SVA. Los diferentes sistemas de archivos pueden implementar estas operaciones, pero existe una versión genérica en el SVA.

- `void (*initialize) (struct inode *inode, short type)`

La operación `initialize` se llama para inicializar los descriptores de cuotas correspondientes al i-nodo referenciado por el parámetro `inode`. El parámetro `type` especifica el tipo de cuota a inicializar: `USRQUOTA`, `GRPQUOTA`, o el valor `-1` para indicar los dos tipos.

- `void (*drop) (struct inode *inode)`

La operación `drop` se llama para liberar los descriptores de cuotas asociados al i-nodo referenciado por el parámetro `inode`.

- `int (*alloc_block) (const struct inode *inode, unsigned long count)`

La operación `alloc_block` se llama a fin de verificar que nuevos bloques pueden asignarse al i-nodo referenciado por el parámetro `inode`. El número de bloques de 1 KB a asignar se especifica por el parámetro `count`.

- `int (*alloc_inode) (const struct inode *inode, unsigned long count)`

La operación `alloc_block` se llama a fin de verificar que es posible asignar nuevos i-nodos al propietario y al grupo del i-nodo referenciado por el parámetro `inode`. El número de i-nodos a asignar se especifica por el parámetro `count`.

- `void (*free_block) (const struct inode *inode, unsigned long count)`

La operación `free_block` se llama en la liberación de bloques para el i-nodo referenciado por el parámetro `inode`, a fin de actualizar el número de bloques contabilizados. El parámetro `count` especifica el número de bloques a liberar.

- `void (*free_inode) (const struct inode *inode, unsigned long count)`

La operación `free_inode` llama en la liberación de i-nodos al propietario y al grupo del i-nodo referenciado por el parámetro `inode`, a fin de actualizar el número de i-nodos contabilizados. El parámetro `count` especifica el número de i-nodos a liberar.

- `int (*transfer) (struct inode *inode, struct iattr *iattr, char direction)`

La operación `transfer` se llama al cambiar de propietario o de grupo el archivo referenciado por el parámetro `inode`. Transfiere el número de bloques e i-nodos contabilizados de un descriptor de cuota a otro.

6 Presentación detallada de la implementación

6.1 Funciones internas del SVA

6.1.1 Gestión de descriptores de sistemas de archivos montados

La gestión de descriptores de sistemas de archivo se implementa en el archivo fuente `fs/super.c`. Este archivo define diversas variables globales:

- `ROOT_DEV`, de tipo `kdev_t`: esta variable contiene el identificador del dispositivo que contiene el sistema de archivos raíz;
- `super_blocks`, matriz de elementos de tipo `struct super_block`: esta matriz contiene los descriptores de los sistemas de archivos montados;
- `file_systems`, de tipo `struct file_system_type*`: este puntero contiene la dirección del primer elemento de la lista de tipos de sistemas de archivos soportados por Linux;
- `vfsmntlist`, de tipo `struct vfsmount *`: este puntero contiene la dirección del último elemento de la lista de descriptores de sistemas de archivos montados;
- `vfsmnttail`, de tipo `struct vfsmount*`: este puntero contiene la dirección del último elemento de la lista de descriptores de sistemas de archivos montados;

- `mru_vfsmnt`, de tipo `struct vfsmount *`: este puntero contiene la dirección del descriptor de sistema de archivos montados obtenido en la última búsqueda; se utiliza como caché del resultado anterior.

Varias funciones permiten actuar sobre la lista de archivos montados:

- `lookup_vfsmnt`: esta función efectúa una búsqueda en la lista de sistemas de archivos montados. Efectúa un bucle de comparación en la lista apuntada por `vfsmntlist` y devuelve la dirección del descriptor encontrado, o el valor `NULL` en caso de error de la búsqueda.
- `add_vfsmnt`: esta función añade una entrada en la lista de sistemas de archivos montados. Asigna un descriptor llamando a la función `kmalloc`, lo inicializa, y lo encadena en la lista. Devuelve la dirección del descriptor asignado o el valor `NULL` en caso de error.
- `remove_vfsmnt`: esta función suprime una entrada en la lista de sistemas de archivos montados. Efectúa primero un bucle de búsqueda del descriptor en la lista apuntada por `vfsmntlist`. Una vez ha encontrado el descriptor a suprimir, lo quita de la lista modificando los encadenamientos, y los libera llamando a la función `kfree`.

Este módulo define también funciones que gestionan la lista de tipos de sistemas de archivos soportados:

- `register_filesystem`: esta función se llama para registrar un tipo de sistema de archivos soportado. Efectúa primero una búsqueda en la lista apuntada por `file_systems`. Si se encuentra el tipo de sistema de archivos, ello significa que ya ha sido registrado, y devuelve entonces el error `EBUSY`. En caso contrario, añade el descriptor de tipo de sistema de archivos al final de la lista.
- `unregister_filesystem`: esta función se llama para suprimir una entrada de la lista de tipos de sistemas de archivos soportados, y efectúa una búsqueda de la entrada a suprimir. Si se encuentra la entrada, se quita de la lista, por una modificación de los encadenamientos, y se devuelve el valor 0. En caso contrario, se devuelve el error `EINVAL`.
- `fs_name`: esta función se llama para buscar una entrada cuyo nombre se especifica, en la lista de tipos de sistemas de archivos soportados. Si la entrada se encuentra en la lista, se devuelve su número de orden, si no la función devuelve el error `EINVAL`.
- `fs_index`: esta función se llama para buscar una entrada cuyo número de orden se especifica, en la lista de tipos de sistemas de archivos soportados. Si se encuentra la entrada se devuelve su nombre, si no la función devuelve el error `EINVAL`.

- `fs_maxindex`: esta función devuelve el número de entradas en la lista de tipos de sistemas de archivos soportados.

Para terminar, en este módulo se definen funciones de gestión de sistemas de archivos montados:

- `get_super`: esta función efectúa la búsqueda de una entrada asociada a un dispositivo en la tabla `super_blocks`. Devuelve la dirección del descriptor correspondiente, o el valor `NULL` en caso de error.
- `__wait_on_super`: esta función asegura la sincronización entre varios procesos en modo núcleo que acceden al mismo descriptor de sistema de archivos.
- `sync_supers`: esta función provoca la reescritura en disco de todos los superbloques modificados. Explora la tabla `super_blocks`, y llama a la operación `write_super` asociada a cada entrada modificada.
- `put_super`: esta función se llama cuando un sistema de archivos deja de usarse, típicamente al desmontarlo. Efectúa ciertas comprobaciones de coherencia, y llama a la operación `put_super` asociada.

6.1.2 Gestión de los i-nodos

La gestión de los i-nodos se implementa en el archivo fuente `fs/inode.c`. Este archivo define varias constantes y variables globales:

- `NR_IHASH`: número de listas de *hash*;
- `hash_table`, tabla de elementos de tipo `inode_hash_entry`: esta tabla contiene los punteros a las listas de *hashing*;
- `first_inode`, de tipo `struct inode *`: este puntero contiene la dirección del primer descriptor de la lista de i-nodos;
- `nr_inodes`, de tipo `int`: esta variable contiene el número de i-nodos asignados; se inicializa a 0;
- `inode_wait`, de tipo `struct wait_queue *`: esta variable se utiliza para acceder a un i-nodo disponible: si no puede asignarse ningún i-nodo, el proceso que llama utiliza la función `sleep_on` sobre esta variable; un proceso que libera un i-nodo utiliza la función `wake_up` sobre esta variable para despertar a los procesos suspendidos en espera de i-nodos;

- `nr_free_inodes`, de tipo `int`: esta variable contiene el número de i-nodos asignados pero no utilizados; se inicializa a 0;
- `max_inodes`, de tipo `int`: esta variable contiene el número máximo de descriptores de i-nodos; se inicializa con la constante `NR_INODE` definida en `<linux/fs.h>` pero su valor puede modificarse llamando a la primitiva `sysctl`.

Se definen varias funciones internas:

- `hashfn`: esta función implementa la función de *hash* sobre los i-nodos.
- `hash`: esta función llama a la función `hashfn`, y devuelve la dirección del puntero al primer elemento de la lista de *hash* correspondiente.
- `insert_inode_free`: esta función inserta un descriptor al principio de la lista.
- `remove_inode_free`: esta función suprime un descriptor de la lista, modificando los encadenamientos.
- `insert_inode_hash`: esta función inserta un i-nodo en la lista de *hash* apropiada.
- `remove_inode_hash`: esta función suprime un i-nodo en la lista de *hash* apropiada.
- `put_last_free`: esta función pasa un descriptor al final de la lista. Llama primero a `remove_inode_free` para suprimir el i-nodo de la lista, y lo inserta al final de la lista.
- `grow_inodes`: esta función se llama cuando todos los descriptores están asignados y cuando el núcleo debe asignar otros descriptores. Asigna una página de memoria llamando a la función `__get_free_page`, descompone dicha página en estructuras `inode` que añade a la lista de descriptores llamando a `insert_inode_free`, y modifica el valor de las variables `nr_inodes` y `nr_free_inodes`.
- `lock_inode`, `unlock_inode`, `__wait_on_inode` y `wait_on_inode`: estas funciones aseguran la sincronización entre varios procesos en modo núcleo que acceden al mismo descriptor de i-nodo. Permiten respectivamente bloquear el i-nodo, desbloquearlo, y esperar a que no esté bloqueado.
- `write_inode`: esta función se llama para escribir el contenido de un i-nodo modificado en disco. Bloquea el i-nodo, llama a la operación `write_inode` asociada al sistema de archivos que contiene el i-nodo, y lo desbloquea.
- `read_inode`: esta función se llama para leer el contenido de un i-nodo desde el disco. Se bloquea el i-nodo, se llama a la operación `read_inode` asociada al sistema de archivos que contiene el i-nodo, y se desbloquea.

Este módulo proporciona también funciones de servicio:

- `inode_init`: esta función se llama en la inicialización del sistema. Pone a cero las listas de *hash* e inicializa `first_inode` con el valor `NULL`.
- `clear_inode`: esta función devuelve un i-nodo puesto a cero. El i-nodo se suprime de las listas llamando a `remove_inode_hash` y `remove_inode_free`, se pone a cero, y se añade a la lista llamando a `insert_inode_free`.
- `fs_may_mount`: esta función verifica que un sistema de archivos presente en un dispositivo puede montarse. Explora la lista de i-nodos, comprobando cada entrada. Si al menos un i-nodo utilizado apunta al dispositivo especificado, devuelve el valor 0, si no devuelve el valor 1.
- `fs_may_umount`: esta función verifica que un sistema de archivos puede desmontarse. Explora la lista de i-nodos, comprobando cada entrada. Si ningún i-nodo del sistema de archivos, excepto el i-nodo raíz, está en uso, devuelve el valor 1, si no devuelve el valor 0.
- `fs_may_remount_ro`: esta función verifica que un sistema de archivos puede volver a montarse en lectura exclusiva. Explora la lista de descriptores de archivos abiertos, comprobando cada entrada. Si no hay ningún archivo correspondiente a un i-nodo presente en el sistema de archivos abierto en escritura, devuelve el valor 1, si no devuelve el valor 0.
- `inode_change_ok`: esta función verifica que el proceso que llama tiene permiso para cambiar los atributos del i-nodo. Las verificaciones efectuadas son las siguientes:
 - sólo el superusuario puede modificar el propietario;
 - sólo el superusuario y el propietario pueden modificar el grupo del archivo; en este último caso, el propietario debe ser miembro del grupo especificado;
 - sólo el superusuario y el propietario pueden modificar los permisos y las fechas asociadas al archivo.

Si las pruebas son positivas, `inode_change_ok` devuelve el valor 1, si no devuelve el valor 0.

- `inode_setattr`: esta función modifica los atributos de un i-nodo.
- `notify_change`: esta función se llama para modificar los atributos de un i-nodo. Si una operación `notify_change` está definida para el sistema de archivos que contiene el i-nodo, la llama; si no llama a `inode_change_ok` y a `inode_setattr` para modificar los atributos.

- `bmap`: esta función se llama para obtener el número de bloque físico correspondiente a un número de bloque lógico del i-nodo. Si hay una operación `bmap` definida para el i-nodo, se llama; si no se devuelve el valor 0.
- `invalidate_inodes`: esta función invalida los i-nodos correspondientes a un sistema de archivos. Explora la lista de i-nodos y libera cada entrada asociada al dispositivo especificado, llamando a `clear_inode`.
- `sync_inodes`: esta función reescribe en disco los i-nodos correspondientes a un sistema de archivos. Explora la lista de i-nodos y reescribe cada entrada asociada al dispositivo especificado, llamando a `write_inode`.
- `iput`: esta función se llama cuando un proceso deja de utilizar un i-nodo. Si el número de usuarios del i-nodo (campo `i_count`) es superior a 1, simplemente se decrementa. En caso contrario, significa que el i-nodo ya no se utiliza, y entonces `iput` despierta los procesos en espera de un i-nodo disponible llamando a `wake_up` sobre la variable `inode_wait`. Se llama a la operación `put_inode` asociada al sistema de archivos correspondiente si está definida, y el contenido del i-nodo se reescribe llamando a `write_inode` si ya ha sido modificado. Finalmente, el campo `i_count` del i-nodo se decrementa, y se incrementa `nr_free_inodes`.
- `get_empty_inode`: se llama esta función para obtener un i-nodo no utilizado. Si quedan pocos i-nodos no asignados, se llama primero a la función `grow_inodes`. Seguidamente, se explora la lista de i-nodos. Todos los i-nodos no utilizados (cuyo campo `i_count` es nulo) se examinan, observando el valor de sus campos `i_lock`, `i_dirt` e `i_nrpages`. Si un i-nodo tiene estos tres campos a nulo, se utiliza.

Tras un bucle de búsqueda, si ningún i-nodo tiene los tres campos nulos, y si no se ha alcanzado el número máximo de i-nodos, la función `grow_inodes` se llama de nuevo, y la búsqueda vuelve a empezar. Si el número máximo de i-nodos se alcanza y no se ha encontrado ningún i-nodo disponible, se llama a la función `sleep_on` sobre la variable `inode_wait` a fin de dejar al proceso que llama en espera de un i-nodo. El proceso será despertado cuando otro proceso libere un i-nodo llamando a la función `iput`.

Cuando se encuentra un i-nodo, su contenido se inicializa, la variable `nr_free_inodes` se decrementa, y se devuelve la dirección del i-nodo.

- `get_pipe_inode`: esta función se llama para obtener un i-nodo correspondiente a una tubería.
- `__iget`: esta función se llama para obtener un descriptor correspondiente a un sistema de archivos montado y a un número de i-nodo. Efectúa una búsqueda en la lista de `hash` correspondiente. Si no se encuentra ningún descriptor correspondien-

te al i-nodo especificado, se asigna un nuevo descriptor llamando a `get_empty_inode`, se inicializa y se inserta en las listas llamando a las funciones `put_last_free` e `insert_inode_hash`, luego su contenido se carga en memoria por una llamada a `read_inode`.

Una vez se ha encontrado o asignado un i-nodo y cargado en memoria, su número de usos (campo `i_count`) se incrementa, y `nr_free_inodes` se decrementa eventualmente. Finalmente, se devuelve la dirección del descriptor.

Hay que observar que un parámetro especifica si los puntos de montaje deben atravesarse.

- `iget`: esta función llama a `__iget` especificando que atravesase los puntos de montaje.

6.1.3 Gestión de descriptores de archivos abiertos

La gestión de la lista de descriptores de archivos abiertos se implementa en el archivo fuente `fs/file_table.c`. Este archivo define tres variables globales:

- `first_file`, de tipo `struct file *`: este puntero contiene la dirección del primer descriptor de la lista: se inicializa a `NUL`;
- `nr_file`, de tipo `int`: esta variable contiene el número de descriptores utilizados; se inicializa a 0;
- `max_files`, de tipo `int`: esta variable contiene el número máximo de descriptores de archivos; se inicializa con la constante `NR_FILE` definida en `<linux/fs.h>` pero su valor puede modificarse llamando a la primitiva `sysctl`.

Se definen varias funciones internas:

- `insert_file_free`: esta función inserta un descriptor al principio de la lista tras haber puesto a 0 el campo `f_count` para indicar que el descriptor no se utiliza.
- `remove_file_free`: esta función suprime un descriptor de la lista, modificando los encadenamientos.
- `put_last_free`: esta función inserta un descriptor al final de la lista.
- `grow_files`: esta función se llama cuando todos los descriptores están asignados y el núcleo debe asignar nuevos descriptores. Asigna una página de memoria llamando a la función `__get_free_page`, descompone dicha página en estructuras `file` que añade a la lista de descriptores llamando a `insert_file_free`, y modifica el valor de `nr_files`.

También se definen funciones de servicio:

- `file_table_init`: esta función se llama en la inicialización del sistema. No efectúa ningún tratamiento, porque las variables globales de este módulo se inicializan en su declaración.
- `get_empty_filp`: esta función se llama cuando el núcleo debe asignar un descriptor de archivo, en el tratamiento de la primitiva `open` por ejemplo. Explora la lista buscando un descriptor no utilizado, es decir, que tiene el campo `f_count` a 0. Si la lista no contiene ningún descriptor inutilizado, y si el número de descriptors asignados es inferior al número máximo, llama a `grow_files` para asignar nuevos descriptors, y reemprende la búsqueda.

Cuando `get_empty_filp` encuentra un descriptor sin usar, lo coloca al final de la lista llamando sucesivamente a `remove_file_free` y a `put_last_free`. Inicializa el campo `f_count` para indicar que el descriptor no se usa. Finalmente, devuelve la dirección del descriptor a quien llama.

También se definen dos funciones relacionadas con las cuotas de disco en este módulo, `add_dquot_ref` y `reset_dquot_ptrs`. Estas funciones se detallan en la sección 6.3.12.

Asimismo, se definen varias funciones de servicio en `<linux/file.h>` y en el archivo fuente `fs/open.c`:

- `fget`: esta función efectúa la conversión entre un descriptor de archivo abierto por el proceso actual y un descriptor de archivo, utilizando la tabla de archivos abiertos asociada al proceso actual. Luego incrementa el número de referencias del descriptor de archivo, y devuelve su dirección.
- `__fput`: esta función se llama en el cierre final de un archivo. La operación `release` asociada al archivo se llama si existe, luego `__fput` llama a `put_write_access` si el archivo estaba abierto en escritura, y finalmente se llama a la función `iput` para liberar el i-nodo correspondiente al archivo.
- `fput`: esta función se llama tras usar un archivo. Decrementa el número de referencias del descriptor, y llama a `__fput` si este número llega a nulo.

6.1.4 Gestión de descriptors de cuotas de disco

La gestión de la lista de descriptors de cuotas en disco se implementa en el archivo fuente `fs/dquot.c`. Este archivo define diversas variables globales:

- `nr_quots`, de tipo `int`: esta variable contiene el número de descriptores de cuotas de disco asignadas pero no utilizadas, y se inicializa a 0;
- `nr_free_quots`, de tipo `int`: esta variable contiene el número de descriptores de cuotas de disco asignadas pero no utilizadas, y se inicializa a 0;
- `first_dquot`, de tipo `struct dquot *`: este puntero contiene la dirección del primer descriptor de la lista; se inicializa a `NULL`;
- `hash_table`, tabla de elementos de tipo `struct dquots *`: esta tabla contiene los punteros a las listas de *hash*;
- `dqstats`, de tipo `struct dqstats`: esta variable contiene las estadísticas de utilización de las cuotas de disco;
- `dquot_wait`, de tipo `struct wait_queue *`: esta variable se utiliza para obtener un descriptor disponible: si no puede asignarse ningún descriptor, el proceso que llama utiliza la función `sleep_on` sobre esta variable; un proceso que libera un descriptor utiliza la función `wake_up` sobre esta variable para despertar a los procesos suspendidos en espera de descriptores.

Las funciones internas de gestión de cuotas se inspiran fuertemente en el módulo de gestión de los i-nodos:

- `hashfn`: esta función implementa la función de *hash*.
- `hash`: esta función llama a la función `hashfn`, y devuelve la dirección del puntero al primer elemento de la lista de *hash* correspondiente.
- `has_quota_enabled`: esta función llama a `lookup_vfsmnt` para obtener el descriptor de sistema de archivos montado especificado, y luego comprueba si las cuotas están activas para este sistema de archivos. Si lo están, devuelve el valor 1, si no devuelve 0.
- `insert_dquot_free`: esta función inserta un descriptor al principio de la lista.
- `remove_dquot_free`: esta función suprime un descriptor de la lista, modificando los encadenamientos.
- `insert_dquot_hash`: esta función inserta un descriptor en la lista de *hash* apropiada.
- `remove_dquot_hash`: esta función suprime un i-nodo de la lista de *hash* apropiada.

- `put_last_free`: esta función pasa un descriptor al fin de la lista. Llama primero a `remove_dquot_free` para suprimir el descriptor de la lista, y luego lo inserta al fin de la lista.
- `grow_dquots`: esta función se llama cuando todos los descriptors están asignados y el núcleo debe asignar nuevos descriptors. Asigna una página de memoria llamando a la función `__get_free_page`, descompone dicha página en estructuras `dquot` que añade a la lista de descriptors llamando a `insert_dquot_free`, y luego modifica el valor de las variables `nr_dquots` y `nr_free_dquots`.
- `lock_dquot`, `unlock_dquot`, `__wait_on_dquot`, y `wait_on_dquot`: estas funciones aseguran la sincronización entre varios procesos en modo núcleo que acceden al mismo descriptor de cuota. Permiten respectivamente bloquear el descriptor, desbloquearlo, y esperar a que no esté bloqueado.
- `clear_dquot`: esta función devuelve un descriptor puesto a cero. El descriptor se suprime de las listas llamando a `remove_dquot_hash` y `remove_dquot_free`, se pone a cero y se añade a la lista llamando a `insert_dquot_free`.
- `write_dquot`: esta función se llama para escribir el contenido de un descriptor modificado en disco. Bloquea el descriptor, llama a la operación `lseek` asociada al archivo de definición de cuotas para posicionarse, y llama a la operación `write` para escribir el descriptor en disco.
- `read_dquot`: se llama esta función para leer el contenido de un descriptor desde el disco. Bloquea el descriptor, llama a la operación `lseek` asociada al archivo de definición de cuotas para posicionarse, y llama a la operación `read` para leer el descriptor desde el disco.
- `dquot_incr_inodes`, `dquot_incr_blocks`: estas funciones incrementan respectivamente el número de i-nodos y el número de bloques en un descriptor de cuotas. El descriptor se marca seguidamente como modificado.
- `dquot_decr_inodes`, `dquot_decr_blocks`: estas funciones decrementan respectivamente el número de i-nodos y el número de bloques en un descriptor de cuotas. Si esta operación hace pasar el número de i-nodos o de bloques por debajo del límite «flexible», el campo `dq_itime` o `dq_btime`, respectivamente, se pone a cero. El descriptor se marca seguidamente como modificado.
- `check_idq`: esta función se llama en la asignación de nuevos i-nodos. Verifica que estos i-nodos pueden añadirse al descriptor:

- si este añadido sobrepasa el límite absoluto, se muestra un mensaje de error en el terminal asociado al proceso que llama, y se devuelve el valor `NO_QUOTA`;
 - si este añadido sobrepasa el límite «flexible», y ha expirado el margen concedido, se muestra un mensaje de error en el terminal asociado al proceso que llama, y se devuelve el valor `NO_QUOTA`;
 - si este añadido significa sobrepasar el límite «flexible», se muestra un mensaje de aviso en el terminal asociado al proceso que llama, se memoriza la fecha de fin de margen de demora en el campo `dq_itime`, y se devuelve el valor `QUOTA_OK`;
 - en todos los demás casos, se devuelve el valor `QUOTA_OK`.
- `check_bdq`: esta función se llama en la asignación de nuevos bloques. Verifica que estos bloques pueden añadirse al descriptor de manera similar a `check_idq`.
 - `dqput`: esta función se llama cuando un proceso deja de utilizar un descriptor de cuota. Si el número de usos del descriptor (campo `dq_count`) es superior a 1, simplemente se decrementa. En caso contrario, significa que el descriptor ya no se utiliza, y `dqput` despierta los procesos en espera de un descriptor disponible llamando a `wake_up` sobre la variable `dquot_wait`. El descriptor se reescribe seguidamente en disco llamando a `write_dquot` si se ha modificado. Finalmente, el campo `dq_count` del descriptor se decrementa, y se incrementa `nr_free_dquots`.
 - `get_empty_dquot`: esta función se llama para obtener un descriptor sin utilizar. Si quedan pocos descriptors no utilizados, la función `grow_dquots` se llama primero para asignar nuevos descriptors. Seguidamente se efectúa un bucle de búsqueda en la lista. Todos los descriptors no utilizados (cuyo campo `dq_count` es nulo) se examinan comprobando el campo `dq_flags` para verificar si los descriptors han sido modificados o se han bloqueado en memoria.

Tras el bucle de búsqueda, si no se ha encontrado ningún descriptor libre, y si el número máximo de descriptors no ha sido asignado, la función `grow_dquots` se llama de nuevo, y la búsqueda vuelve a empezar. Si el número máximo de descriptors se alcanza y no se ha encontrado ningún descriptor disponible, la función `sleep_on` se llama sobre la variable `dquot_wait` para dejar al proceso que llama en espera de un descriptor. El proceso será despertado cuando otro proceso libere un descriptor llamando a la función `dqput`.

Cuando se encuentra un descriptor, se pone a cero llamando a `clear_dquot`, la variable `nr_free` se decrementa y se devuelve la dirección del descriptor.

- `dqget`: esta función se llama para obtener un descriptor de cuota correspondiente a un sistema de archivos, a un tipo de cuotas, y a un identificador de usuario o de grupo. Verifica primero que las cuotas están activas en el sistema de archivos especificado y devuelve el valor `NODQUOT` si no es así. Se efectúa un bucle de búsqueda en la lista de *hash* correspondiente. Si la entrada se encuentran en la lista, su número de usos (campo `dq_count`) se incrementa y se devuelve su dirección. Si no, se asigna una nueva entrada llamando a `get_empty_dquot`, se inicializa el descriptor, y se inserta en la lista por llamadas sucesivas a `put_last_free` e `insert_dquot_hash`. Finalmente, se lee el contenido del descriptor desde el disco llamando a `read_dquot`, y se devuelve la dirección del descriptor.
- `dquot_init`: esta función se llama en la inicialización del sistema. Inicializa a cero las listas de *hash*, así como la variable `dqstats` que contiene las estadísticas de uso.

El módulo de gestión de cuotas define dos funciones de servicio:

- `sync_dquots`: esta función se llama para reescribir en disco todos los descriptors de cuotas modificados. Efectúa un bucle sobre la lista de descriptors y reescribe todos los descriptors modificados llamando a `write_dquot`.
- `invalidate_dquots`: esta función invalida todos los descriptors de cuotas asociados a un sistema de archivos. Efectúa un bucle sobre la lista de descriptors, reescribe los descriptors correspondientes llamando a `write_dquot` si han sido modificados, y libera los descriptors correspondientes por la función `clear_dquot`.

El archivo fuente *fs/dquot.c* define también las variables y funciones vinculadas a las operaciones sobre cuotas:

- `dquot_initialize`: esta función se llama para inicializar un descriptor de cuota asociado a un i-nodo. Llama a `dqget` para obtener el descriptor de cuota correspondiente al sistema de archivos que contiene el i-nodo, al tipo de cuota, y al identificador de usuario o de grupo. La dirección del descriptor se memoriza en el campo `i_dquot` del i-nodo, y se apaga el indicador `S_WRITE` en el campo `i_flags` para indicar que hay un descriptor de cuota asociado a este i-nodo.
- `dquot_drop`: esta función se llama para liberar los descriptors de cuotas asociados a un i-nodo. Para ello, se llama a la función `dqput`. Luego el campo `i_dquot` del i-nodo se pone a `NODQUOT` y el indicador `S_WRITE` se desactiva en el campo `i_flags` para indicar que el i-nodo no posee ya descriptor de cuota asociado.
- `dquot_alloc_block`, `dquot_alloc_inode`: estas funciones se llaman en la asignación de bloques o i-nodos, respectivamente, a un i-nodo. Llamam a

`check_bdq` o `check_idq`, respectivamente, para verificar que la asignación está permitida, y luego a la función `dquot_incr_blocks` o `dquot_incr_inodes`, respectivamente, para guardar la asignación en el descriptor de cuota correspondiente. En caso de fallo de la asignación, estas funciones devuelven `NO_QUOTA`, si no devuelven `QUOTA_OK`.

- `dquot_free_block`, `dquot_free_inode`: estas funciones se llaman en la liberación de bloques o i-nodos, respectivamente. Llamam a la función `dquot_decr_blocks` o `dquot_decr_inodes`, respectivamente, para guardar la liberación en el descriptor de cuota correspondiente.
- `dquot_transfer`: esta función se llama cuando el propietario o el grupo de un archivo se modifica. En este caso, un i-nodo suplementario y sus bloques asociados deben contabilizarse al nuevo propietario o al nuevo grupo y deben ser asociados a los números de i-nodos y de bloques del anterior propietario o del grupo. En un primer momento, `dquot_transfer` verifica que la transferencia puede efectuarse llamando a las funciones `check_bdq` y `check_idq`. Si no es así, se devuelve el valor `NO_QUOTA`. Si la transferencia es posible, se llama a las funciones `dquot_decr_blocks` y `dquot_decr_inodes` para reducir el número de i-nodos y bloques contabilizados al anterior propietario o al grupo, y luego se llama a las funciones `dquot_incr_blocks` y `dquot_incr_inodes` para añadir estos números al nuevo propietario o al grupo.

La variable `dquot_operations`, de tipo estructura `dquot_operations`, contiene los punteros a estas funciones.

6.1.5 El caché de nombres

La gestión del caché de nombres se implementa en el archivo fuente *fs/dcache.c*. Este módulo define las constantes y variables globales utilizadas para gestionar el caché:

- `DCACHE_NAME_LEN`: número máximo de caracteres de los nombres de archivos ubicados en el caché;
- `DCACHE_SIZE`: número de entradas en las listas de niveles 1 y 2;
- `DCACHE_HASH_QUEUES`: número de listas de *hash*;
- `level1_dcach`, matriz de elementos de tipo `struct dir_cache_entry`: esta matriz contiene las entradas de la lista de nivel 1;
- `level2_dcach`, matriz de elementos de tipo `struct dir_cache_entry`: esta matriz contiene las entradas de la lista de nivel 2;

- `level1_head`, de tipo `struct dir_cache_entry *`: este puntero contiene la dirección del primer elemento de la lista de nivel 1;
- `level2_head`, de tipo `struct dir_cache_entry *`: este puntero contiene la dirección del primer elemento de la lista de nivel 2;
- `hash_table`, matriz de elementos de tipo `struct hash_list`: esta matriz contiene los punteros a las listas de *hash*.

Se definen varias funciones internas:

- `remove_lru`: esta función suprime una entrada de una lista LRU, modificando los encadenamientos.
- `add_lru`: esta función inserta una entrada en una lista LRU, modificando los encadenamientos.
- `update_lru`: esta función pasa una entrada al principio de la lista, modificando los encadenamientos. Para ello, llama sucesivamente a las funciones `remove_lru` y `add_lru`.
- `remove_hash`: esta función suprime un elemento de una lista de *hash* modificando los encadenamientos.
- `add_hash`: esta función inserta una entrada en una lista de *hash*, modificando los encadenamientos.
- `find_entry`: esta función efectúa la búsqueda de una entrada en una lista de *hash* especificada. Para cada elemento de la lista de *hash*, compara el número de dispositivo, el número de i-nodo del directorio, el número de versión del directorio, y el nombre de la entrada con los parámetros que se le pasan.
- `move_to_level2`: esta función cambia una entrada de lista, suprimiéndola de la lista de nivel 1, e insertándola en la lista de nivel 2. Si la entrada se encuentra ya en la segunda lista, se coloca al principio de la lista.

También se definen otras funciones que pueden ser llamadas por otros módulos del SVS y por los sistemas de archivos:

- `dcache_lookup`: esta función se llama para resolver una entrada. Utiliza la función de *hash* `hash_fn` para determinar la lista de *hash* a explorar, y llama a la función `find_entry` que efectúa la búsqueda. Si la entrada se encuentra en la lista, la función `move_to_level2` se llama para colocarla al principio de la lista LRU de nivel 2.

- `dcache_add`: esta función se llama para insertar una entrada en la creación de un archivo o en la lectura de un directorio con `readdir`. Se llama primero a `find_entry` para determinar si la entrada ya ha sido registrada. Si es así, la entrada se coloca al principio de la lista llamando a `update_lru`, si no se suprime una entrada de la lista de nivel 1 y la nueva entrada se añade al principio.
- `name_cache_init`: esta función se llama en la inicialización del sistema. Inicializa las dos listas encadenadas y crea las listas de *hash* vacías.

Hay que observar que este módulo no ofrece una función `dcache_remove`, que podría llamarse en la supresión de un archivo, a fin de invalidar una entrada en el caché. Linux utiliza otra técnica para invalidar automáticamente las entradas obsoletas. Se asocia un número de versión a cada i-nodo de directorio cargado en memoria. Este número se memoriza en las listas en la llamada a `dcache_add`, y toda modificación en el directorio, es decir, todo añadido o supresión de archivos, provoca el cambio de este número de versión. Si un archivo cuyo nombre está presente en el caché se suprime en un directorio, el número de versión se modifica, y una llamada ulterior a `find_entry` fracasará porque el número de versión del directorio es ahora diferente del que está almacenado en el caché. De este modo, todas las entradas correspondientes a un directorio se hacen inválidas al añadir o suprimir archivos en este directorio, y esas entradas se eliminan progresivamente de las listas LRU al añadirse otras entradas.

Linux utiliza una variable interna, llamada `event`, para gestionar los números de versiones. Esta variable se define en el archivo `kernel/sched.c` y se inicializa a 0 al inicializarse el sistema. Ésta se incrementa con cada modificación de un directorio, y se asigna al número de versión del i-nodo correspondiente.

6.1.6 Funciones de soporte de gestión de i-nodos

Varias funciones de soporte manipulan los descriptores de i-nodos:

- `permission`: esta función se llama para verificar que el proceso que llama posee los permisos necesarios sobre un i-nodo. Si una operación `permission` se asocia al i-nodo, se llama; si no los permisos se comparan directamente con los derechos de acceso especificados en el campo `i_mode` del i-nodo.
- `get_write_access`: esta función se llama para obtener el derecho de escribir sobre un i-nodo. Explora la lista de procesos y comprueba si un proceso posee una sección de este i-nodo proyectado en su espacio de direccionamiento con la opción `VM_DENYWRITE`. Si es así, se devuelve el error `ETXBSY`. Si no, el campo `i_writecount` del i-nodo se incrementa para indicar que un proceso suplementario accede al archivo en escritura, y se devuelve el valor 0.

- `put_write_access`: esta función se llama cuando un proceso deja de acceder en escritura a un i-nodo. Decrementa el campo `i_writecount` del i-nodo.

6.1.7 Gestión de los nombres de archivos

Las llamadas al sistema de manipulación de archivos aceptan como parámetro nombres de archivos o descriptores de entrada/salida. Las funciones que implementan estas llamadas, así como las funciones internas del SVA y de sistemas de archivos, actúan sobre i-nodos o descriptores de archivos abiertos, y asimismo el SVA proporciona funciones de manipulación y de conversión de nombres de archivos. Estas funciones se implementan en el archivo fuente `fs/namei.c`.

- `getname`: esta función se llama para copiar un nombre de archivo en el espacio de direccionamiento del núcleo. Verifica que el nombre de archivo es válido, es decir, que no se trata de una cadena vacía, y asigna una página de memoria llamando a la función `__get_free_page`, y copia el nombre en esta página.
- `putname`: esta función se llama para liberar un nombre de archivo devuelto por `getname`. Se libera la página asignada por esta función llamando a `free_page`.
- `lookup`: esta función se llama para buscar un i-nodo correspondiente a una entrada de directorio. Primero verifica que el proceso que llama tiene derecho a explorar el directorio especificado llamando a `permission`. Luego trata el caso de la entrada especial «..»: si el directorio tratado es la raíz, «..» se refiere a sí mismo; si el directorio es la raíz de un sistema de archivos, «..» se refiere a la entrada «..» del punto de montaje. Finalmente, `lookup` llama a la operación `lookup` asociada al i-nodo del directorio, y devuelve su resultado.
- `follow_link`: esta función se llama para resolver un nombre de enlace simbólico. Verifica que una operación `follow_link` se asocia al i-nodo del enlace, llama a esta operación y devuelve su resultado. Si no existe ninguna operación para este i-nodo, significa que no es un enlace simbólico y se devuelve el propio i-nodo.
- `dir_namei`: esta función interna se llama para resolver el nombre del directorio que contiene un archivo cuyo nombre se pasa como parámetro. Elige primero un directorio de partida para la exploración: si el nombre empieza por el carácter «/», es un nombre absoluto y el directorio raíz se elige como base, si no es un nombre relativo y el directorio actual del proceso que llama se elige como base. Luego efectúa un bucle de resolución iterativa llamando a la función `lookup` sobre cada uno de los componentes del nombre de archivo. Devuelve el i-nodo del último directorio explorado durante la resolución del nombre, es decir, del directorio que contiene el nombre de archivo especificado.

- `_namei`: esta función interna se llama para resolver un nombre de archivo. Llama a la función `dir_namei` para obtener el i-nodo del directorio padre, y llama a la función `lookup` para resolver el nombre de archivo en este directorio. El parámetro `follow_links` especifica el tratamiento a efectuar sobre los enlaces simbólicos: si está posicionado, `_namei` llama luego a `follow_link` para seguir eventualmente un enlace simbólico.
- `lnamei`: esta función interna se llama para resolver un nombre de enlace simbólico. Llama a `getname` para obtener el nombre del enlace, luego a `_namei` para resolver el nombre, posicionando el parámetro `follow_links`.
- `namei`: esta función interna se llama para resolver un nombre de archivo. Llama a `getname` para obtener el nombre del archivo, luego a `_namei` para resolver el nombre, sin posicionar el parámetro `follow_links`.

6.1.8 Gestión de bloqueos

El archivo fuente `fs/locks.c` contiene el código de gestión de bloqueos asociados a los archivos. La variable `file_lock_table` contiene la dirección del primer elemento de la lista de descriptores de bloqueos. Se inicializa al valor `NULL`.

Se definen varias funciones internas de gestión de listas de bloqueos:

- `locks_alloc_lock`: esta función asigna un descriptor de bloqueo, llamando a la función `kmalloc`, y lo inicializa.
- `locks_insert_lock`: esta función inserta un descriptor de bloqueos en la lista de bloqueos asociados a un i-nodo y en la lista global de descriptores de bloqueos.
- `locks_delete_lock`: esta función se llama para suprimir un descriptor de bloqueo. Suprime primero el descriptor de la lista global de bloqueos. Luego explora la lista de bloqueos bloqueados que están asociados al descriptor, y despierta a cada uno de los procesos correspondientes por una llamada a `wake_up`. Finalmente, el descriptor se libera llamando a `kfree`.
- `locks_insert_block`: esta función inserta un descriptor de bloqueos en cola de una lista de bloqueos bloqueados que están asociados a otro bloqueo.
- `locks_delete_block`: esta función suprime un descriptor de una lista de bloqueos bloqueados que están asociados a otro bloqueo.
- `posix_remove_locks`: esta función suprime todos los bloqueos efectuados por un proceso sobre un archivo. Explora la lista de bloqueos y llama a `locks_delete_block` para cada bloqueo creado por el proceso.

- `flock_remove_locks`: esta función suprime todos los bloqueos efectuados sobre un descriptor de archivo. Explora la lista de bloqueos y llama a `locks_delete_block` para cada bloqueo creado sobre el descriptor de archivo especificado, si el número de referencias de este descriptor es igual a 1.
- `posix_make_locks`, `flock_make_lock`: estas dos funciones inicializan un descriptor de bloqueo. El descriptor creado por `posix_make_locks` contiene las direcciones de inicio y fin del bloqueo en el archivo, mientras que el descriptor creado por `flock_make_lock` especifica que el conjunto del archivo está bloqueado.
- `locks_overlap`: esta función devuelve un booleano indicando si dos descriptors de bloqueos sobre el mismo archivo poseen una parte común.
- `locks_conflict`: esta función comprueba si dos descriptors de bloqueos entran en conflicto. Comprueba si los dos bloqueos tienen una parte común (llamando a `locks_overlap`). Si no es así, devuelve el valor 0, si no comprueba si los dos bloqueos son compatibles. Si uno de los bloqueos es de tipo exclusivo (es decir, si su tipo es `F_WRLCK`), los bloqueos son incompatibles, y se devuelve el valor 1. Si no, `locks_conflict` devuelve el valor 0.
- `posix_locks_conflict`, `flock_locks_conflict`: estas funciones comprueban si dos bloqueos entran en conflicto. `posix_locks_conflict` devuelve el valor 0 si los dos bloqueos están vinculados al mismo proceso, y llama a `locks_conflict` en caso contrario. `flock_locks_conflict` devuelve el valor 0 si los dos bloqueos están asociados al mismo descriptor de archivo, y llama a `locks_conflict` en caso contrario.
- `posix_locks_deadlock`: esta función comprueba si la implementación de un bloqueo puede crear un interbloqueo entre dos procesos, especificados por los parámetros `my_task` y `blocked_task`. Explora la lista de descriptors de bloqueos: para cada descriptor, explora la lista de procesos en espera sobre este bloqueo. Si el proceso en espera es `blocked_task` y el bloqueo pertenece a `my_task`, se provocaría un interbloqueo con la creación de un bloqueo, y se devuelve el valor 1. Al final de la exploración de la lista de descriptors de bloqueos, si no se ha detectado ningún interbloqueo, se devuelve el valor 0.

La función `locks_remove_lock` se define también en este archivo. Esta función se llama cuando se cierra un archivo. Llama a la función `posix_remove_locks` o `flock_remove_locks`, según el tipo de bloqueos asociados al archivo.

6.2 El *búfer caché*

6.2.1 Presentación

La gestión del *búfer caché* se implementa en el archivo fuente *fs/buffer.c*. Esta gestión interactúa con la gestión de la memoria presentada en el capítulo 8. El contenido de las memorias intermedias se coloca en páginas de memoria y el estado de las páginas de memoria debe consultarse o modificarse a menudo. Para ello, Linux utiliza la tabla *mem_map* que contiene un descriptor para cada página de memoria. Este descriptor se presenta en detalle en el capítulo 8, sección 5.2, pero es necesario introducir ya ahora ciertos campos utilizados por el *búfer caché*:

- **count**: Este campo contiene el número de referencias a la página.
- **flags**: Este campo contiene el estado de la página.
- **buffers**: Este campo contiene la dirección del primer descriptor de memoria cuyo contenido se sitúa en la página.

Las funciones de gestión del *búfer caché* se descomponen en varias categorías:

- las funciones de gestión de las listas de memorias intermedias;
- las funciones llamadas en la realización de entradas/salidas;
- las funciones que modifican el tamaño del *búfer caché*;
- las funciones de gestión de dispositivos;
- las funciones de servicio que permiten acceder a las memorias intermedias;
- las funciones de reescritura del contenido de las memorias en disco;
- las funciones de gestión de los *clusters*;
- la función de inicialización del *búfer caché*.

Estas funciones se describen en las secciones siguientes.

6.2.2 Gestión de las listas de memorias intermedias

Varias funciones internas permiten gestionar las listas en las que las memorias intermedias se registran. Se utilizan varias variables para mantener las listas:

- `hash_table`: Tabla que contiene los punteros al primer *búfer* de cada lista de *hash*;
- `nr_hash`: Número de listas de *hash*;
- `lru_list`: Tabla que contiene los punteros al primer *búfer* de cada lista LRU;
- `free_list`: Tabla que contiene los punteros al primer *búfer* disponible para cada tamaño de bloque;
- `unused_list`: Puntero al primer búfer no utilizado;
- `reuse_list`: Puntero al primer búfer no utilizado pero que puede reutilizarse;
- `nr_buffers`: Número de búfers asignados;
- `nr_buffers_type`: Tabla que contiene el número de búfers registrados en cada lista;
- `nr_buffers_size`: Tabla que contiene el número de búfers asignados para cada tamaño de bloque.

Las funciones que aseguran la gestión de estas listas son las siguientes:

- `__wait_on_buffer` y `wait_on_buffer`: estas funciones permiten sincronizar varios procesos en modo núcleo que acceden a la misma memoria intermedia.
- `_hashfn` y `hash`: estas funciones implementan la función utilizada para las listas de *hash*.
- `remove_from_hash_queue`: esta función suprime una memoria intermedia de la lista de *hash* correspondiente.
- `remove_from_lru_list`: esta función suprime una memoria intermedia de la lista LRU correspondiente.
- `remove_from_free_list`: esta función suprime una memoria intermedia de la lista de bloques libres.
- `remove_from_queues`: esta función suprime una memoria intermedia de las listas en las que se encuentra, llamando a `remove_from_free_list` o `remove_from_hash_queue`, y luego a `remove_from_lru_list`.
- `put_last_lru`: esta función desplaza una memoria intermedia al fin de su lista LRU.
- `put_last_free`: esta función inserta una memoria intermedia al fin de la lista de bloques libres.

- `insert_into_queues`: esta función inserta una memoria intermedia en una lista. Si la memoria no se utiliza, `put_last_free` se llama para añadirlo al fin de la lista de bloques libres, si no la memoria se inserta en una lista LRU.
- `find_buffer`: esta función efectúa una búsqueda de una memoria intermedia. Explora la lista de hash correspondiente y devuelve la dirección del descriptor, o el valor NULL en caso de error.
- `set_writetime`: esta función posiciona el campo `b_flush_time` del descriptor de búfer a la fecha actual (contenido de la variable `jiffies`), si el contenido de la memoria intermedia se ha modificado. En caso contrario, este campo se pone a cero.
- `refile_buffer`: esta función se llama cuando un proceso libera una memoria intermedia. Examina el estado de la memoria, y la cambia eventualmente de lista LRU. La fecha de última utilización de la memoria (campo `b_lru_time`) se pone a la fecha actual en esta función.
- `put_unused_buffer_head`: esta función se llama para liberar una memoria intermedia que ya no se usa. Añade el búfer a la lista de bloques no utilizados apuntada por `unused_list`, y utiliza la función `wake_up` sobre la variable `buffer_wait` a fin de despertar los procesos en espera de búfer disponible.
- `get_more_buffer_heads`: esta función se llama para asignar descriptores de memorias suplementarias. Repite el proceso hasta poder asignar nuevas memorias. En cada pasada:
 1. comprueba la variable `unused_list` para determinar si existen memorias intermedias sin utilizar; si es así, vuelve a quien llama;
 2. llama a `get_free_page` para obtener una página de memoria;
 3. si la asignación fracasa, llama a `sleep_on` sobre la variable `buffer_wait` a fin de suspender el proceso que llama en espera de una memoria intermedia disponible;
 4. si la asignación tiene éxito, la página asignada se descompone en descriptores de memorias intermedias, y añade cada una de las memorias en la lista de bloques no utilizados apuntada por `unused_list`.
- `recover_reusable_buffer_heads`: esta función inserta las memorias intermedias a reutilizar en la lista de memorias no utilizadas. Explora la lista de memorias a reutilizar, apuntada por `reuse_list`, y llama a `put_unused_buffer_head` para cada búfer.

- `get_unused_buffer_head`: esta función se llama para obtener un descriptor de búfer no utilizado. Llama sucesivamente a `recover_reusable_buffer_heads` y `get_more_buffer_heads`, y devuelve el primer descriptor de la lista de memorias no utilizadas, que es apuntada por `unused_list`.

6.2.3 Entradas/salidas

Las funciones de servicio permiten a las otras partes del núcleo provocar entradas/salidas, utilizando el *búfer caché*. Estas funciones afectan tanto a la creación de memorias intermedias en páginas de memoria, las entradas/salidas sobre éstas, como a las funciones de desbloqueo y desasignación de memorias llamada tras el fin de una entrada/salida.

Estas funciones son:

- `create_buffers`: esta función crea nuevas memorias intermedias en una página de memoria. Asigna descriptors de memorias llamando a `get_unused_buffer_head`, descompone la página en búfers, y coloca su dirección en el campo `b_data` de los descriptors.
- `free_async_buffers`: esta función se llama para liberar las memorias intermedias asociadas a una página de memoria. La lista de estas memorias se explora, y cada memoria intermedia se pone en la lista de memorias a reutilizar, que está apuntada por la variable `reuse_list`.
- `brw_page`: esta función se llama para efectuar una lectura o una escritura de datos en o desde una página de memoria. Asigna primero descriptors de memorias correspondientes al contenido de la página llamando a `create_buffers`, y efectúa luego un bucle para tratar las memorias así creadas.

Cada descriptor de memoria intermedia se inicializa y el indicador `FreeOnIO` se activa, para indicar que se trata de una memoria intermedia temporal que deberá liberarse al final de la entrada/salida. Luego se llama a `get_hash_table` a fin de verificar si otra memoria intermedia en el *búfer* se refiere al mismo bloque en disco. Si es así, la memoria encontrada se utiliza:

- en el caso de una lectura de datos, su contenido se transfiere en la página a leer;
- en el caso de una escritura, la parte correspondiente del contenido de la página se transfiere a la memoria intermedia, y ésta se marca como modificada por una llamada a `mark_buffer_dirty`.

En caso contrario, la memoria intermedia asociada al contenido de la página se guarda en una tabla.

Tras este bucle, la función `ll_rw_block` (véase el capítulo 7, sección 4.2) se llama para lanzar la entrada/salida. Los descriptores de las memorias temporales creadas por `brw_page` se le pasan como parámetro.

- `mark_buffer_uptodate`: esta función se llama al final de una lectura de datos, cuando se ha efectuado la entrada/salida. Indica que el contenido de la memoria intermedia correspondiente está al día, activando el indicador `BH_Uptodate` en el campo `b_state`, y luego explora las memorias intermedias cuyo contenido se sitúa en la misma página de memoria. Si están todas actualizadas, la propia página se marca como actualizada.
- `unlock_buffer`: esta función se llama al final de una entrada/salida para desbloquear una memoria intermedia. Suprime el indicador `BH_Lock`, luego despierta los procesos en espera sobre esta memoria llamando a la función `wake_up`.

En el caso de una memoria intermedia (cuyo indicador `HB_FreeOnIO` se posiciona), `unlock_buffer` explora las otras memorias intermedias situadas en la misma página de memoria. Si todas poseen un número de referencias igual a 1, es decir, si se utilizan únicamente en la página que las contiene, se liberan llamando a `free_async_buffers`.

- `generic_readpage`: esta función se llama para leer el contenido de una página de memoria desde un archivo. Se efectúa un bucle para obtener las direcciones de los bloques que componen el contenido de la página, llamando a la operación `bmap` asociada al i-nodo del archivo. Una vez calculada esta lista de números, la lectura se lanza llamando a `brw_page`.

6.2.4 Modificación del tamaño del *búfer caché*

Bajo Linux, contrariamente a otros sistemas operativos, el tamaño del *búfer caché* no se fija en la inicialización del sistema. El *búfer caché* está inicialmente vacío y cambia de tamaño en el transcurso de la ejecución del sistema. Cuando el núcleo necesita memorias intermedias suplementarias, se asignan nuevas páginas de memoria al *búfer caché* a fin de ampliarlo. Cuando al sistema le falta memoria, puede liberar ciertas memorias intermedias así como las páginas de memoria que las contienen, lo cual reduce el tamaño del *búfer caché*.

Las funciones que modifican el tamaño del *búfer caché* son las siguientes:

- `grow_buffers`: esta función se llama para ampliar el *búfer caché*. Asigna una página de memoria asignando a `__get_free_page`, y llama a `create_buf-`

fers para asignar las memorias intermedias, e inserta las memorias creadas en la lista de memorias no utilizadas.

- `try_to_free_buffer`: esta función se llama a fin de reducir la memoria asignada al *búfer caché*. Explora todas las memorias intermedias contenidas en una página de memoria, y comprueba si se utilizan. Si es así, devuelve el valor 0. En caso contrario, las memorias intermedias se suprimen de las listas llamando a `remove_from_queues` y a `put_unused_buffer_head`. Finalmente, la página que contenía las memorias intermedias se libera llamando a `free_page`, y se devuelve el valor 1.
- `shrink_specific_buffers`: esta función reduce la memoria asignada a las memorias asociadas a un tamaño de bloque especificado. Explora primero la lista de memorias disponibles y comprueba el estado de cada memoria intermedia contenida: si la memoria no se utiliza, `try_to_free_buffer` se llama para intentar liberar todas las memorias contenidas en la misma página de memoria. Tras esta exploración de la lista de memorias disponibles, `shrink_specific_buffers` sigue las listas LRU, comprueba si cada memoria intermedia se ha utilizado recientemente y, si es así, la función `try_to_free_buffer` se llama para intentar liberar la memoria.
- `refill_freelist`: esta función se llama para llenar la lista de memorias intermedias disponibles para un tamaño dado. Comprueba en primer lugar si el número de memorias disponibles para este tamaño es superior a 100. Si es así, considera que hay suficientes memorias disponibles y termina su ejecución. En caso contrario, llama a la función `grow_buffers` a fin de asignar nuevas memorias intermedias. Si las asignaciones son suficientes, se termina.

En el caso en que el *búfer caché* no pueda ampliarse, el tratamiento efectuado por `refill_freelist` es bastante complejo. La función debe seleccionar las memorias intermedias no utilizadas y colocarlas en la lista de memorias intermedias disponibles. Para ello, explora las diferentes listas LRU que contienen las memorias, y por cada lista, selecciona una memoria candidata. Esta memoria debe tener un número de referencias (campo `b_count`) nulo, su contenido no debe haber sido modificado (campo `b_dirty` nulo), y la página que la contiene no debe incluirse en el espacio de direccionamiento de un proceso. Tras la exploración de las listas, se elige una memoria intermedia candidata por cada lista. Se procede seguidamente a una «elección» de la memoria apropiada. Esta elección consiste en elegir la memoria que se ha utilizado más antiguamente, es decir, la que tiene la fecha más antigua en el campo `b_lru_time`.

La memoria elegida se suprime de su lista llamando a `remove_from_queues`, y luego inserta en la lista de memorias disponibles llamando a `put_last_free`.

Se efectúa una búsqueda seguidamente en la lista donde la memoria estaba registrada a fin de elegir otro candidato.

Tras la transferencia de la memoria en la lista de memorias disponibles, `refill_freelist` comprueba si es necesario reiniciar la operación. Si se han creado suficientes memorias intermedias disponibles, la función termina; si no intenta de nuevo ampliar el *búfer caché* llamando a `grow_buffers`, y luego reinicia la selección de una memoria en el punto de «la elección».

6.2.5 Funciones de gestión de dispositivos

Dos funciones permiten actuar sobre las memorias intermedias asociadas a un dispositivo especificado.

La función `invalidate_buffers` se llama para invalidar las memorias intermedias correspondientes a un sistema de archivos especificado. Explora las listas LRU, y actualiza el estado de las memorias correspondiente modificando el campo `b_state`, para indicar que las memorias no son válidas.

La función `set_blocksize` permite especificar el tamaño de los bloques lógicos presentes en un dispositivo. Memoriza este tamaño en la entrada de la tabla `blksize_size` asociada al dispositivo, y luego explora las listas de memorias intermedias. Toda memoria correspondiente al dispositivo cuyo tamaño no es el especificado se suprime de su lista de hash llamando a `remove_from_queue` y se inserta en la lista de memorias no utilizadas.

6.2.6 Funciones de acceso a las memorias intermedias

El módulo de gestión del *búfer caché* ofrece funciones de servicio, utilizables por el SVA y los sistemas de archivos, para acceder a las memorias intermedias:

- `get_hash_table`: esta función se llama para obtener una memoria intermedia existente correspondiente a un sistema de archivos y a un número de bloque. Llama a `find_buffer` para buscar la memoria intermedia. Si se encuentra la memoria correspondiente, su número de usos (campo `b_count`) se incrementa, y se devuelve su dirección; si no `get_hash_table` devuelve el valor `NULL`.
- `getblk`: esta función se llama para obtener una memoria intermedia correspondiente a un dispositivo y a un número de bloque. Primero llama a `get_hash_table` para buscar la memoria intermedia, y devuelve el resultado si la búsqueda es positiva. En caso contrario, es decir, si no existe ninguna memoria

intermedia correspondiente a la petición, se llama a la función `refill_freelist`, y se lanza una nueva búsqueda llamando a `find_buffer`. Si esta nueva búsqueda es positiva, el tratamiento se reinicia. Si no, se asigna un descriptor de la lista de memorias no utilizadas, se inicializa y se inserta en las listas por `insert_into_queues`, devolviendo su dirección.

- `__brelse`: esta función se llama para liberar una memoria intermedia. Ésta se cambia eventualmente de lista LRU llamando a `refile_buffer`, y su número de usos (campo `b_count`) se decrementa.
- `__bforget`: esta función se llama para liberar una memoria intermedia, de la misma manera que `__brelse`, pero la inserta en la lista de memorias no utilizadas.
- `bforget`: esta función simplemente llama a `__bforget`.
- `bread`: esta función se llama para obtener una memoria intermedia correspondiente a un dispositivo y a un número de bloque. Primero llama a `getblk` para obtener el descriptor de la memoria, y eventualmente llama a `ll_rw_block` a fin de leer el contenido de la misma desde el disco. Devuelve la dirección del descriptor, o el valor `NULL` en caso de error.
- `breada`: esta función es similar a `bread`, pero permite activar la lectura anticipada de otros bloques. Llama a `getblk` para obtener el descriptor de memoria especificada, y luego eventualmente a `ll_rw_block` a fin de leer su contenido desde el disco. Para cada bloque especificado, llama a `getblk` para obtener un descriptor de memoria intermedia, y lanza la lectura de todos los bloques llamando a `ll_rw_block`. No espera a la finalización de esta lectura adicional, y devuelve la dirección del descriptor de memoria correspondiente al primer bloque.

6.2.7 Reescritura de las memorias intermedias modificadas

La función `sync_buffers` reescribe las memorias intermedias modificadas en disco. Puede efectuar hasta tres pasadas de exploración de las listas:

- Durante la primera pasada, las memorias intermedias modificadas se reescriben de manera asíncrona en disco llamando a la función `ll_rw_block`. Las memorias bloqueadas en memoria no se tienen en cuenta.
- Durante las pasadas siguientes, `sync_buffers` se pone en espera sobre las memorias intermedias bloqueadas llamando a `wait_on_buffer`.

El número de pasadas efectuadas depende del valor del parámetro `wait`. La primitiva `sync` lo posiciona a 0 para que `sync_buffers` lance las entradas/salidas sin esperar

su fin, mientras que la llamada al sistema *fsync* lo posiciona a 1 para que *sync_buffers* efectúe varias pasadas, a fin de que las memorias intermedias modificadas se reescriban físicamente en disco al final de la función.

Los parámetros del proceso *bdflush* se almacenan en la variable *bdf_prm*. Esta variable contiene los campos siguientes:

tipo	campo	descripción
int	<i>nfract</i>	Porcentaje máximo de memorias intermedias modificadas: cuando éste se sobrepasa, <i>bdflush</i> se activa para reescribir las memorias intermedias en disco
int	<i>ndirty</i>	Número máximo de memorias intermedias a reescribir en cada activación de <i>bdflush</i>
int	<i>nrefill</i>	Número de memorias intermedias a colocar en la lista de memorias intermedias disponibles, cada vez que se reconstruye esta lista
int	<i>nref_dirt</i>	Número de memorias intermedias modificadas a partir del que <i>bdflush</i> debe activarse cuando la lista de memorias intermedias disponibles se reconstruye (este parámetro no se usa en Linux 2.0)
int	<i>clu_nfract</i>	Porcentaje de memorias intermedias a explorar en la búsqueda de clúster disponible (véase la sección 6.2.8) (este parámetro no se usa en Linux 2.0)
int	<i>age_buffer</i>	Tiempo durante el que una memoria intermedia puede conservar sus modificaciones antes de escribirse en disco; este tiempo se expresa en número de ciclos de reloj
int	<i>age_super</i>	Tiempo durante el cual una memoria intermedia con un descriptor de sistema de archivos puede conservar sus modificaciones antes de escribirse en disco; este tiempo se expresa en número de ciclos de reloj
int	<i>lav_const</i>	Constante utilizada para calcular el porcentaje de uso de las memorias intermedias asociadas a los bloques de un cierto tamaño
int	<i>lav_ratio</i>	Este parámetro no se usa en Linux 2.0

Los valores predeterminados de estos parámetros son 60, 500, 64, 256, 15, 30 * HZ, 5 * HZ, 1884 y 2.

Las variables *bdflush_min* y *bdflush_max* definen los valores mínimos y máximos de los parámetros de *bdflush*. Contienen respectivamente los valores 0, 10, 5, 25, 0, 100, 100, 1, 1 y 100, 5000, 2000, 2000, 100, 60000, 60000, 2047, 5.

Varias funciones permiten gestionar el funcionamiento de *bdflush*:

- *wake_up_bdflush*: esta función utiliza la función *wake_up* para despertar al proceso *bdflush*, y espera que termine su tratamiento utilizando *sleep_on* sobre la cola de espera *bdflush_done*.
- *sync_old_buffers*: esta función reescribe el contenido de las memorias intermedias en disco. Primero llama a *sync_supers* y *sync_inodes* para reescribir los descriptors de sistemas de archivos e i-nodos en disco. Luego explora la lista de memorias intermedias modificadas. Para cada memoria, la función *refi-*

`le_buffer` se llama si su contenido no se ha modificado, a fin de desplazar la memoria a la lista correspondiente. Si el contenido de la memoria intermedia se ha modificado, si la memoria no está bloqueada, y si su fecha de escritura se ha alcanzado (es decir, si el valor del campo `s_flush_time` es inferior o igual a la fecha actual, contenida en la variable `jiffies`), la función `ll_rw_block` se llama para reescribir el contenido de la memoria en disco. Finalmente, al fin de la función, las estadísticas de uso de cada tamaño de bloque se recalculan.

- `sys_bdflush`: esta función implementa la llamada al sistema `bdflush`. Permite que el proceso que llama obtenga o modifique uno de los parámetros que controlan la ejecución del proceso `bdflush`.
- `bdflush`: esta función implementa el proceso `bdflush`. Este proceso se crea al inicializar el sistema, y se ejecuta en modo núcleo. Tras la inicialización del descriptor del proceso, `bdflush` efectúa un bucle infinito. A cada pasada, efectúa un tratamiento similar al de `sync_old_buffers`, es decir, guarda en disco el contenido de las memorias intermedias modificadas cuya fecha de reescritura se ha alcanzado, y luego suspende el proceso `bdflush`, llamando a `interruptible_sleep_on`, si el número de memorias modificadas es inferior al porcentaje especificado por el parámetro `nfract` de `bdflush`. El proceso se despertará por las funciones `refill_freelist` y `refile_buffer`, que llamarán a `wake_up_bdflush`, cuando `bdflush` deba ejecutarse de nuevo.

6.2.8 Gestión de *clústers*

Linux utiliza la noción de *clúster* para las entradas/salidas. Un *clúster* es un grupo de memorias intermedias cuyos contenidos son contiguos en memoria, y que corresponden a bloques contiguos en disco. La ventaja de los *clústers* reside en el hecho de que la lectura o la escritura de las memorias intermedias puede efectuarse en una sola entrada/salida, mientras que hay que efectuar varias entradas/salidas (una por bloque) si las memorias intermedias correspondientes a bloques contiguos en disco están dispersas en memoria. Este mecanismo de *clustering* es utilizado por el sistema de archivos Ext2 (véase la sección 6.5.9), así como por las entradas/salidas directas sobre dispositivos accesibles en modo bloque (véase el capítulo 7, sección 4.3). Hay varias funciones disponibles para gestionar los *clústers*:

- `try_to_reassign`: esta función comprueba si todas las memorias intermedias contenidas en una página de memoria están disponibles y las asocia a un nuevo *clúster* si no es así. Explora la lista de memorias contenidas en la página y verifica que cada memoria intermedia está disponible, se suprimen de las listas de hash por una llamada a `remove_from_queues`, sus direcciones en disco (número de dis-

positivo y número de bloque) se modifican, y se registran de nuevo en las listas llamando a `insert_into_queues`.

- `reassign_cluster`: esta función se llama para obtener un nuevo *clúster*. Primero llama a `refill_freelist` a fin de obtener suficientes memorias intermedias disponibles, luego explora la lista de memorias disponibles. Para cada memoria intermedia de la lista, llama a `try_to_reassign` para intentar crear un nuevo *clúster* en la página que contiene esta memoria intermedia.
- `try_to_generate_cluster`: esta función intenta generar un *clúster* en una nueva página de memoria. Asigna una página llamando a `get_free_page`, luego llama a `create_buffers` a fin de crear memorias intermedias que apuntan al contenido de la página de memoria. Utiliza seguidamente `find_buffer` para determinar si memorias intermedias corresponden al dispositivo y a los bloques especificados. Si es así, es imposible crear un *clúster*, las memorias intermedias creadas se liberan llamando a `put_unused_buffer_head`, la página de memoria se libera y se devuelve el valor 0. En el caso contrario, puede crearse un *clúster*. Las memorias intermedias creadas se inicializan y se registran en las listas llamando a `insert_into_queues`.
- `generate_cluster`: esta función se llama para crear un *clúster* por una serie de bloques especificados. Verifica primero que los bloques especificados sean contiguos y que no exista ninguna memoria intermedia referida ya a estos bloques. `maybe_shrink_lav_buffers` se llama a continuación para disminuir el número de memorias intermedias utilizadas si es demasiado importante. Si esta función ha suprimido memorias intermedias, la función `try_to_generate_cluster` se llama para asignar un *clúster*. En caso contrario, o si `try_to_generate_cluster` ha fracasado, la memoria disponible se comprueba; si es limitada, la función `reassign_cluster` se llama para crear el *clúster* a partir de memorias existentes; si no se crea un nuevo *clúster* llamando a `try_to_generate_cluster`.

6.2.9 Inicialización del búfer caché

La función `buffer_init` se llama al arrancar el sistema a fin de inicializar el *búfer caché*. Calcula el número de listas de hash en función del tamaño de memoria disponible, asigna estas listas llamando a `vmalloc`, y las inicializa. Finalmente, llama a `grow_buffers` a fin de asignar algunas memorias intermedias.

6.3 Implementación de las llamadas al sistema

6.3.1 Organización de los archivos fuente

Las llamadas al sistema de manipulación de archivos se implementan en los archivos fuente situados en el directorio *fs*. La tabla siguiente recapitula las llamadas al sistema implementadas en los distintos archivos.

archivo fuente	llamadas al sistema
buffer.c	<i>sync, fsync, fdatasync, bdflush</i>
dquot.c	<i>quotactl</i>
exec.c	<i>exit, brk, uselib</i>
fcntl.c	<i>dup2, dup, fcntl</i>
ioctl.c	<i>ioctl</i>
locks.c	<i>flock</i>
namei.c	<i>mknod, mkdir, rmdir, unlink, symlink, link, rename</i>
noquot.c	<i>quotactl</i>
open.c	<i>statfs, fstatfs, truncate, ftruncate, utime, utimes, acces, chdir, fchdir, chroot, fchmod, chmod, fchown, open, creat, close, whangup</i>
read_write.c	<i>lseek, llseek, read, write, readv, writev</i>
readdir.c	<i>getdents</i>
select.c	<i>select</i> (esta primitiva se detalla en el capítulo 7, sección 4.4)
stat.c	<i>stat, newstat, lstat, newlstat, fstat, newfstat, readlink</i>
super.c	<i>sysfs, ustat, umount, mount</i>

No todas las llamadas al sistema se detallan en esta sección. Los tratamientos de ciertas llamadas al sistema presentan grandes similitudes y sólo se explican algunos ejemplos significativos.

6.3.2 Manipulación de archivos

Las primitivas de manipulación de archivos, implementadas en el archivo fuente *fs/namei.c*, son de relativamente fácil comprensión, y todas siguen el mismo modelo. Sólo las llamadas *mkdir* y *rename* se detallan aquí. La implementación de las otras llamadas al sistema es similar.

La función *sys_mkdir* implementa la llamada al sistema *mkdir*. Llama a *getname* para obtener el nombre del directorio a crear, luego la creación se efectúa llamando a *do_mkdir*. Finalmente, el nombre del directorio se libera por una llamada a *putname*.

La función `do_mkdir` se encarga de la creación del directorio. Llama a `dir_namei` para obtener el i-nodo del directorio padre, y efectúa algunas verificaciones: comprueba que el sistema de archivos no está montado en lectura exclusiva, que el proceso que llama tiene el derecho de crear un subdirectorio, y que una operación `mkdir` está asociada al i-nodo del directorio padre. Una vez efectuadas estas pruebas, se llama a la operación sobre cuotas `initialize` para cargar los descriptores de cuotas asociados al directorio padre. Finalmente, se llama a la operación `mkdir` asociada al i-nodo del directorio padre para proceder a la creación del subdirectorio.

La función `sys_rename` implementa la llamada al sistema *rename*. De la misma manera que `sys_mkdir` llama a `getname` para obtener los nombres de archivos, luego a `do_rename` para proceder al cambio de nombre, y finalmente a `putname` para liberar los nombres de archivos.

La función `do_rename` es algo más compleja que `do_mkdir`, porque debe efectuar más comprobaciones. Llama en primer lugar a `dir_namei` para obtener el i-nodo del directorio padre del archivo a renombrar. Luego, verifica que el proceso que llama posee permisos suficientes sobre el directorio padre, y que la entrada a renombrar no es «.» ni «..». El mismo tratamiento se efectúa sobre el nuevo nombre del archivo. Luego, `do_rename` verifica que los dos directorios se encuentran en el mismo sistema de archivos, y que este último no está montado en lectura exclusiva. La operación sobre cuotas `initialize` asociada al nuevo directorio padre se llama para cargar los descriptores de cuotas. Finalmente, `do_rename` llama a la operación *rename* asociada al anterior directorio padre para efectuar el cambio de nombre.

6.3.3 Gestión de los atributos de archivos

Las llamadas al sistema que permiten modificar los atributos de archivos se implementan en el archivo fuente *fs/open.c*. En razón de su gran similitud, sólo se detallan aquí las llamadas *chown* y *fchown*.

La función `sys_chmod` implementa la primitiva *chmod*. Llama a la función `namei` para obtener el i-nodo a modificar, luego comprueba si es posible modificar este i-nodo; en otras palabras, verifica que el sistema de archivos correspondiente no está montado en lectura exclusiva y que el i-nodo no es inmutable (un i-nodo inmutable no puede modificarse jamás). Luego llama a la función `notify_change` indicándole que el modo y la fecha de última modificación del i-nodo deben modificarse.

La función `sys_fchown`, que implementa la primitiva *fchown* es algo más compleja, porque efectúa comprobaciones suplementarias. Obtiene primero el i-nodo a modificar derreferenciando el puntero contenido en el descriptor de archivo abierto.

Seguidamente, verifica que el i-nodo puede modificarse, e inicializa una variable `newattr` de tipo estructura `iattr` que contiene los nuevos identificadores de usuario y de grupo. Si el propietario se modifica, el modo se actualiza en `newattr` a fin de borrar el bit `setuid`; asimismo, si el grupo se modifica, el modo se actualiza para borrar el bit `setgid`. Si se asocian operaciones de cuotas al i-nodo, la operación `transfer` se llama para tener en cuenta la modificación de propietario y de grupo, y el cambio se efectúa llamando a la función `notify_change`.

El archivo fuente `fs/stat.c` contiene la implementación de las primitivas `stat`, `lstat` y `fstat`. Las funciones que las implementan obtienen el i-nodo del archivo afectado, y devuelven las informaciones que contiene en la memoria intermedia cuya dirección se pasa como parámetro. Las funciones `cp_old_stat` y `cp_new_stat` efectúan la copia de las informaciones.

6.3.4 Entrada/salida sobre archivos

Las primitivas de entrada/salida se implementan en varios archivos fuente:

- `fs/namei.c` contiene la función `open_namei` que contiene la mayor parte del código de la llamada `open`;
- `fs/open.c` contiene las funciones `sys_open` y `sys_close` que implementan las primitivas `open` y `close`;
- `fs/read_write.c` contiene las funciones `sys_lseek`, `sys_llseek`, `sys_read`, `sys_write`, `sys_readv` y `sys_writev` que implementan las primitivas `lseek`, `llseek`, `read`, `write`, `readv` y `writev`.

La función `sys_open` es muy simple: llama a `getname` para obtener el nombre del archivo a abrir, abre el archivo con una llamada a la función `do_open`, y libera el nombre del archivo llamando a `putname`.

La función `do_open` efectúa la apertura propiamente dicha del archivo. Obtiene primero un descriptor de archivo abierto, llamando a la función `get_empty_filp`, y lo inicializa. Seguidamente llama a la función `open_namei` para obtener el i-nodo correspondiente al archivo, luego llama a la función `get_write_access` para verificar que el archivo puede abrirse en escritura si el modo de apertura lo especifica. La etapa siguiente consiste en inicializar el campo `f_op` a partir de las operaciones sobre archivos especificadas por el i-nodo, y llamar a la operación `open` si está definida. Finalmente, la dirección del descriptor de archivo abierto se inserta en la tabla de archivos abiertos por el proceso habitual.

La función `open_namei` implementa una gran parte de la llamada al sistema `open`. Ante todo llama a `dir_namei` para obtener el i-nodo del directorio padre. Si se especifica la opción `O_CREAT`, se llama a la función `lookup` y se devuelve un error si el archivo existe y si la opción `O_EXCL` también se especifica; luego se llama a la operación sobre cuotas `initialize` y a la operación `create` asociadas al i-nodo del directorio, para crear el archivo y obtener su i-nodo. Si no se especifica la opción `O_CREAT`, el i-nodo del archivo se obtiene por una llamada a la operación `lookup` asociada al directorio. La función `follow_link` se llama seguidamente para seguir un eventual enlace simbólico, y se efectúan pruebas de conformidad del modo de apertura respecto al i-nodo. A continuación, si se especifica la opción `O_TRUNC`, el contenido del archivo se suprime: el acceso en escritura se obtiene llamando a `get_write_access`, la función `locks_verify_locked` se usa para comprobar que no hay ningún bloqueo asociado al i-nodo, la función `do_truncate` se llama para liberar el contenido del archivo, y el acceso en escritura se obtiene llamando a la función `put_write_access`. Finalmente, el i-nodo del archivo se devuelve a quien ha llamado.

La función `sys_close` es muy simple: verifica que el número de descriptor de entrada/salida pasado como parámetro es válido, y se llama a la función `close_fp` para efectuar el cierre del archivo.

La función `close_fp` empieza por obtener el i-nodo correspondiente al descriptor de archivo abierto, llama a la función `locks_remove_locks` para suprimir los bloqueos asociados al i-nodo, y libera el descriptor del archivo llamando a `fput`.

Las funciones `sys_lseek` y `sys_llseek` modifican la posición actual de un archivo. Verifican ante todo que el número del descriptor de archivo es válido. Si una operación `lseek` está asociada al descriptor de archivo, se llama; si no la posición actual se modifica directamente cambiando el valor del campo `f_pos` del descriptor de archivo.

La función `sys_read` implementa la lectura de datos desde un archivo. Adquiere primero el descriptor de archivo llamando a `fget`, luego verifica que el modo de apertura del archivo permite la lectura, y que una operación `read` está asociada al archivo. Si estas pruebas dan positivo, la función `locks_verify_area` se llama para verificar que la sección a leer no está bloqueada. Seguidamente, la dirección de la memoria intermedia proporcionada por el proceso que llama se valida por una llamada a `verify_area`, y la lectura se efectúa llamando a la operación `read` asociada al archivo. Finalmente, el descriptor de archivo se libera por una llamada a `fput`.

La función `sys_write` es muy similar a `sys_read`, excepto que efectúa una escritura de datos en lugar de una lectura. La única diferencia notable es que `sys_write` borra los bits `setuid` y `setgid` del i-nodo, si los datos se escriben por un proceso que no posee los derechos de superusuario.

Las primitivas *readv* y *writew* se implementan en la función *do_readv_writew*, y las funciones *sys_readv* y *sys_writew* efectúan únicamente las pruebas de validez del número de descriptor de entrada/salida llamando a *do_readv_writew* para efectuar la entrada/salida. La función *do_readv_writew* verifica primero sus argumentos validando el vector de memorias intermedias que se le pasa. Luego comprueba la existencia de bloqueos en la sección del archivo afectado llamando a *locks_verify_area*, después efectúa un bucle de lectura o escritura de datos, utilizando las memorias intermedias especificadas en el vector pasado como parámetro, llamando a la operación *read* o *write* asociada al descriptor de archivo.

6.3.5 Lectura de directorio

El archivo fuente *fs/readdir.c* contiene las funciones *old_readdir* y *sys_getdents* que implementan las primitivas de lectura de entradas de directorios *readdir* y *getdents*.

Estas dos funciones son muy similares: verifican la validez del número de descriptor de entrada/salida, utilizan la función *verify_area* para validar la dirección de la memoria intermedia pasada como parámetro, y llaman a la operación *readdir* asociada al descriptor de archivo. La diferencia entre las dos funciones se basa en el parámetro *filldir* pasado a la operación *readdir*: *old_readdir* especifica *fillonedir*, que copia una sola entrada de directorio en la memoria intermedia, mientras que *sys_getdents* especifica *filldir*, que copia entradas de directorio en la memoria intermedia hasta que esté lleno.

6.3.6 Gestión de bloqueos

Las llamadas de gestión de bloqueos asociados a los archivos se implementan en el archivo fuente *fs/locks.c*. Se encuentran varias funciones:

- *flock_lock_file*: esta función crea un bloqueo para un archivo entero. Explora la lista de bloqueos asociados al i-nodo correspondiente, y comprueba si un bloqueo existe ya. Si es así, se devuelve el valor 0. Si un bloqueo cuyo tipo es diferente ya existe, se suprime llamando a *locks_delete_lock*. Se asigna seguidamente un nuevo descriptor de bloqueo por *locks_alloc_lock*. Se efectúa un bucle de exploración de descriptors de bloqueos asociados: si existe un conflicto entre un bloqueo existente y el nuevo bloqueo, el proceso que llama se pone en espera de la supresión del primer bloqueo. Cuando todos los conflictos se han resuelto, o en caso de ausencia de conflicto, el nuevo descriptor se inserta en la lista de bloqueos por *locks_insert_lock*.

- `posix_lock_file`: esta función crea un bloqueo asociado a una parte de un archivo. Efectúa primero un bucle sobre la lista de bloqueos asociados al i-nodo comprobando los conflictos y los riesgos de interbloqueo. En caso de interbloqueo, se devuelve el error `EDEADLK`. En caso de conflicto, el proceso que llama se suspende, en espera de la supresión del bloqueo existente. Tras esta verificación, la lista de bloqueos asociados al i-nodo se explora de nuevo para determinar si el nuevo bloqueo puede fusionarse con un bloqueo existente, comparando cada bloqueo puesto por el proceso actual con el nuevo bloqueo:
 - Si los dos bloqueos son del mismo tipo, si poseen una parte común o son adyacentes, se fusionan en un solo bloqueo. El bloqueo existente se suprime por `locks_delete_lock`, y el nuevo descriptor se modifica a fin de incluir la zona correspondiente al antiguo bloqueo.
 - Si los dos bloqueos son de tipo diferente, y si el nuevo incluye el antiguo, los procesos en espera sobre éste se despiertan, y el antiguo bloqueo se suprime.
 - Si los dos bloqueos son de tipo diferente, y si el antiguo incluye al nuevo, el antiguo bloqueo se descompone en dos: una parte se coloca antes del nuevo bloqueo, y una parte después.
 - Finalmente, si los dos bloqueos son de tipo diferente, y si poseen una parte común, las coordenadas del antiguo bloqueo se modifican.

En cada modificación de bloqueo existente, los procesos en espera se despiertan, porque el cambio del bloqueo puede autorizarles a proseguir su ejecución.

- `sys_flock`: esta función implementa la primitiva *flock*. Controla la validez de sus parámetros, y llama a `flock_lock_file` para crear el bloqueo.
- `fcntl_getlk`: esta función implementa el tratamiento de la petición `F_GETLK` de la primitiva *fcntl*. Se controla primero la validez de sus parámetros, y efectúa un bucle de exploración de la lista de bloqueos asociados al i-nodo. Por cada bloqueo existente, llama a `posix_locks_conflict` a fin de determinar si el bloqueo especificado sería origen de un conflicto. Si es así, se devuelve el descriptor del bloqueo existente.
- `fcntl_setlk`: esta función trata las peticiones `F_SETLK` y `F_SETLKW` de la primitiva *fcntl*. Se controla primero la validez de sus parámetros, crea un descriptor de bloqueo llamando a `posix_make_lock`, y llama a `posix_lock_file` para asociar el bloqueo al archivo.

6.3.7 Gestión del búfer caché

Las llamadas al sistema relacionadas con el *búfer caché* se implementan en el archivo fuente *fs/buffer.c*. El tratamiento de las reescrituras de memorias intermedias se descompone en varias funciones:

- **sync_dev**: esta función reescribe los bloques modificados que se asocian a un dispositivo. Para provocar las reescrituras, llama sucesivamente a las funciones **sync_buffers**, **sync_supers**, **sync_inodes**, **sync_buffers** para tratar de nuevo las memorias intermedias que hayan sido modificadas por **sync_inodes** y **sync_dquot**s.
- **fsync_dev**: esta función es similar a **sync_dev**, pero se pone en espera de la reescritura de todas las memorias intermedias modificadas. Por tanto, no termina su ejecución hasta que todas las memorias intermedias modificadas se han reescrito en disco.
- **sys_sync**: esta función implementa la llamada al sistema *sync*. Llama a **fsync_dev** especificando 0 como número de dispositivo a fin que los bloques pertenecientes a todos los dispositivos se reescriban.
- **sys_fsync**: esta función implementa la llamada al sistema *fsync*. Comprueba la validez del número de descriptor de entrada/salida que se le pasa como parámetro, y llama a la operación **fsync** asociada al descriptor de archivo si existe.
- **sys_fdatasync**: esta función implementa la llamada al sistema *fdatasync*. Su tratamiento es idéntico al de **sys_fsync**.

6.3.8 Gestión de los sistemas de archivos

Las llamadas al sistema de gestión de sistemas de archivos se implementan en el archivo fuente *fs/super.c*. Las funciones definidas son las siguientes:

- **sys_sysfs**: esta función implementa la primitiva *sysfs*. Según el valor de su parámetro **option**, llama a **fs_index**, **fs_name** o **fs_maxindex**.
- **read_super**: esta función interna se llama para leer e inicializar el superbloque de un sistema de archivos a montar. Obtiene el descriptor del tipo de sistemas de archivos llamando a **get_fs_type**, asigna un descriptor de superbloque en la tabla **super_blocks**, lo inicializa, y llama a la función **read_super** asociada al tipo de sistema de archivos especificados a fin de leer el superbloque desde el disco.
- **do_umount**: esta función se llama para desmontar un sistema de archivos.

Si el sistema de archivos especificado es el sistema de archivos raíz, desactiva las cuotas llamando a `quota_off`, llama a `fsync_dev` para reescribir los bloques modificados, y modifica las opciones de montaje por una llamada a `remount_sb` a fin de hacer el sistema de archivos accesible en lectura exclusiva.

En caso contrario, las cuotas se desactivan llamando a `quota_off`, luego se llama la función `fs_may_mount` para verificar que es posible desmontar el sistema de archivos. Los i-nodos correspondientes al directorio raíz y al punto de montaje son liberados por una llamada a `iput`. Si el superbloque ha sido modificado, se reescribe en disco llamando a la operación `write_super`. Finalmente, la función `put_super` se llama para liberar el superbloque, y el descriptor de sistema de archivos montado correspondiente se libera por una llamada a `remove_vfsmnt`.

- `sys_umount`: esta función implementa la llamada al sistema `umount`. Verifica primero que el proceso que llama posee los derechos del superusuario, y convierte el nombre especificado en un número de dispositivo. Para ello, convierte el nombre en i-nodo llamando a `namei`, y luego comprueba el tipo del i-nodo: si es un archivo especial, representa el dispositivo que contiene el sistema de archivos; si no es el punto de montaje.

Una vez obtenido el número de dispositivo, se llama a la función `do_umount` para desmontar el sistema de archivos.

- `do_mount`: esta función se llama para montar un sistema de archivos. Verifica primero la validez de las opciones de montaje, y convierte el nombre del punto de montaje en i-nodo llamando a `namei`. Efectúa seguidamente comprobaciones de validez sobre el i-nodo: no debe ser un punto de montaje ya utilizado, y debe corresponder a un directorio. La función `fs_may_mount` se llama también para verificar que el dispositivo puede montarse.

Cuando se han efectuado estas pruebas, el superbloque del sistema de archivos se carga en memoria llamando a `read_super`, se asigna un descriptor de sistema de archivos montado y se llama a `add_vfsmnt`, y el descriptor de superbloque se actualiza.

- `do_remount_sb`: esta función interna se llama para modificar las opciones de montaje de un sistema de archivos. Se verifica la validez de las nuevas opciones de montaje, y llama a la operación `remount_fs` asociada al sistema de archivos si existe. Finalmente, actualiza las opciones de montaje en el descriptor de superbloque y en el descriptor de sistema de archivos montado.
- `do_remount`: esta función interna se llama para modificar las opciones de montaje de un sistema de archivos. Convierte el nombre del punto de montaje especifi-

cado en i-nodo llamando a `namei`, y llama a `do_remount_sb` para modificar las opciones de montaje.

- `copy_mount_options`: esta función interna copia las opciones de montaje (parámetro `data` de la llamada al sistema `mount`) desde el espacio de direccionamiento del proceso actual en el espacio de direccionamiento del núcleo.
- `sys_mount`: esta función implementa la llamada al sistema `mount`. Verifica en primer lugar que el proceso que llama posee los derechos de usuario. Si se especifica la opción `MS_REMOUNT`, la función `do_remount` se llama para modificar las opciones de montaje.

En el caso en que `MS_REMOUNT` no se especifique, el descriptor del tipo de sistema de archivos especificado se obtiene llamando a `get_fs_type`. Si este tipo requiere un dispositivo, el nombre del dispositivo se convierte en i-nodo llamando a `namei`, se efectúan pruebas de validez sobre el i-nodo, y se llama a la operación sobre el archivo `open` asociada para inicializar el dispositivo. En el caso contrario, se obtiene un identificador de dispositivo «ficticio» llamando a la función `get_unnamed_dev`.

Para terminar, `sys_mount` copia las opciones de montaje en el espacio de direccionamiento del núcleo llamando a `copy_mount_options`, luego llama a `do_mount` para proceder al montaje del sistema de archivos.

Las llamadas al sistema `statfs` y `fstatfs` que permiten obtener informaciones sobre un sistema de archivos se implementan en el archivo fuente `fs/open.c` por las funciones `sys_statfs` y `sys_fstatfs`. Estas dos nociones convierten su primer parámetro en i-nodo, bien llamando a la función `namei`, o bien derreferenciando el descriptor de entrada/salida especificado, y verifican que una operación `statfs` se asocia al sistema de archivos que contiene el i-nodo, y devuelven el resultado de esta operación.

6.3.9 Manipulación de descriptores de entrada/salida

Las primitivas de manipulaciones de descriptores de entrada/salida, `dup`, `dup2` y `fcntl` se implementan en el archivo fuente `fs/fcntl.c`. Las funciones definidas son las siguientes:

- `dupfd`: esta función duplica un descriptor de entrada/salida. Verifica primero la validez del número de descriptor que se pasa como parámetro, busca un descriptor no utilizado, y copia las informaciones desde el primer descriptor. Finalmente, incrementa el número de utilizaciones del descriptor de archivo correspondiente.
- `sys_dup2`: esta función implementa la llamada al sistema `dup2`. Verifica la validez del número de descriptor de entrada/salida deseado, lo cierra si está abierto lla-

mando a la función `sys_close`, y llama a `dupfd` para efectuar la duplicación de descriptor, precisándole el número de descriptor a utilizar.

- `sys_dup`: esta función implementa la llamada al sistema `dup`. Llama a la función `dupfd`, sin precisarle el nuevo número del descriptor.
- `sys_fcntl`: esta función implementa la llamada al sistema `fcntl`. Verifica la validez del número de descriptor de entrada/salida especificado y efectúa tratamientos diferentes según el valor del parámetro `cmd` que se le pasa:
 - `F_DUPFD`: llama a `dupfd` para duplicar el descriptor de entrada/salida;
 - `F_GETFD`: devuelve el bit del campo `close_on_exec` correspondiente al descriptor;
 - `F_SETFD`: activa el bit del campo `close_on_exec` correspondiente al descriptor;
 - `F_GETFL`: devuelve el campo `f_flags` del descriptor de archivo;
 - `F_SETFL`: llama a la operación `fsync` asociada al archivo si el indicador `FASYNC` se modifica, y modifica el valor del campo `f_flags` del descriptor de archivo;
 - `F_GETLK`: llama a `fcntl_getlk`;
 - `F_SETLK`: llama a `fcntl_setlk`;
 - `F_SETLKW`: llama a `fcntl_setlk`.

6.3.10 Herencia de descriptores

Al crear un nuevo proceso con la llamada al sistema `fork`, el contexto del proceso padre se duplica. Este contexto incluye los descriptores de archivos. Las funciones `copy_fs` y `copy_files`, definidas en el archivo fuente `kernel/fork.c`, efectúan esta duplicación, y se citan en el capítulo 4, sección 6.2.1.

Si se crea un proceso clon, y las opciones de clonado incluyen `CLONE_FS`, la función `copy_fs` incrementa el número de referencias del descriptor apuntado por el campo `fs` del descriptor de proceso actual. Si no, asigna un descriptor llamando a `kma-lloc`, copia en él el contenido del descriptor asociado al proceso padre, e incrementa el número de referencias a los i-nodos que representan los directorios raíz y actual.

Si se crea un proceso clon, y las opciones de clonado incluyen `CLONE_FILES`, la función `copy_files` incrementa el número de referencias de la tabla de descriptors de archivos abiertos, apuntado por el campo `files` del descriptor de proceso actual. Si no, asigna una tabla de descriptors de archivos abiertos por el proceso hijo, llamando a `kmallocc`, y copia en ella los descriptors de archivos abiertos del proceso padre. El número de referencias de cada descriptor copiado se incrementa.

Al terminar un proceso, su contexto se libera. Las funciones `_exit_fs`, `close_files` y `__exit_files`, definidas en el archivo fuente `kernel/exit.c`, efectúan esta liberación.

La función `__exit_fs` decrementa el número de referencias del descriptor apuntado por el campo `fs` del descriptor de proceso especificado. Si este número llega a nulo, la función `iput` se llama para liberar los directorios raíz y actual asociados al proceso (campos `root` y `pwd`), luego el descriptor se libera llamando a la función `kfree`.

La función `close_files` cierra los archivos abiertos por un proceso. Efectúa un bucle sobre la tabla de descriptors de archivos abiertos por el proceso y llama a la función `close_fp` para cada archivo abierto. La función `__exit_files` decrementa el número de referencias de la tabla de descriptors de archivos abiertos, apuntada por el campo `files` del descriptor de proceso. Si este número llega a nulo, se llama a la función `close_files` para cerrar todos los archivos abiertos, y la tabla de descriptors se libera por una llamada a `kfree`.

6.3.11 Cambio de directorio

Las llamadas al sistema de cambio de directorio, `chdir`, `fchdir` y `chroot` se implementan en el archivo fuente `fs/open.c`. Estas primitivas obtienen el i-nodo del directorio especificado, ya sea llamando a la función `namei` o bien derreferenciando el descriptor de entrada/salida pasado como parámetro, y luego memorizan este i-nodo en el descriptor del proceso actual.

La dirección del descriptor del i-nodo del directorio actual se almacena en el campo `current->fs->pwd`, mientras que la dirección del directorio raíz se almacena en `current->fs->root`.

6.3.12 Gestión de las cuotas de disco

Al igual que las funciones internas de gestión y las operaciones sobre cuotas, la llamada al sistema `quotactl` se implementa en el archivo fuente `fs/dquot.c`. El tratamiento de esta primitiva se descompone en varias funciones:

- `set_dqblk`: esta función trata las opciones `Q_SETQUOTA`, `Q_SETUSE` y `Q_SETQLIM`. Primero llama a la función `dqget` para obtener el descriptor de cuota afectado, modifica su contenido, lo marca como modificado, y lo libera llamando a `dqput`.
- `get_quota`: esta función implementa la opción `Q_GETQUOTA`. Obtiene el descriptor de cuota afectado llamando a `dqget`, copia las informaciones que contiene en el espacio de direccionamiento del proceso que llama por `memcpy_tofs`, y libera el descriptor llamando a `dqput`.
- `get_stats`: esta función implementa la opción `Q_GETQUOTA`. Copia el contenido de la variable `dqstats` en el espacio de direccionamiento del proceso por `memcpy_tofs`.
- `add_dquot_ref`: esta función, definida en el archivo fuente *fs/file_table.c*, se llama al activar las cuotas en un sistema de archivos. Explora la lista de descriptors de archivos abiertos y, por cada archivo abierto en escritura sobre el sistema de archivos especificado, llama a la operación `initialize`.
- `reset_dquot_ref`: esta función, definida en el archivo fuente *fs/file_table.c*, se llama al desactivar las cuotas de un sistema de archivos. Explora la lista de descriptors de archivos abiertos, y, por cada archivo abierto en escritura en el sistema de archivos especificado, asigna el valor `NODQUOT` en el campo `i_dquot` del i-nodo asociado, y desactiva el indicador `S_WRITE` del campo `i_flags` a fin de indicar que el i-nodo no posee descriptor de cuota asociado.
- `quota_off`: esta función implementa la opción `Q_QUOTAOFF`. Busca el descriptor de archivos montado llamando a `lookup_vfsmnt`, y pone el valor `NULL` en el campo `dq_op` del descriptor de superbloque asociado. Seguidamente se llama a las funciones `reset_dquot_ref` e `invalidate_dquots` para liberar los descriptors asociados al sistema de archivos. Finalmente, se cierra el archivo de definición llamando a `close_fp`.
- `quota_on`: esta función implementa la opción `Q_QUOTAON`. Busca el descriptor de archivos montado llamando a `lookup_vfsmnt`, y verifica que las cuotas no están ya activas en este sistema de archivos. Si es así, se devuelve el error `EBUSY`. `quota_on` simula luego la llamada al sistema `open` sobre el archivo de definición de las cuotas: se llama a `open_namei` para obtener el i-nodo del archivo, y luego se inicializa un descriptor de archivo abierto, obtenido llamando a `get_empty_filp`. El archivo se abre llamando a la operación de archivo `open`, y el descriptor de archivo se memoriza en el descriptor de sistema de archivos montado. Finalmente, se llama a la función `add_dquot_ref` para tener en cuenta las cuotas para todos los descriptors de archivos abiertos desde este sistema de archivos.

- `sys_quactotl`: esta función implementa la primitiva *quotactl*. Según el valor de sus parámetros, llama a la función `quota_on`, `quota_off`, `get_quota`, `sync_quotas`, `get_stats` o `set_dqblk`.

6.4 Sistemas de archivos soportados

Linux se desarrolló originariamente bajo Minix [Tanenbaum 1987] y las primeras versiones del sistema sólo soportaban este sistema de archivos, cuyas limitaciones son importantes: tamaño limitado a 64 MB, nombres de archivo limitados a 14 caracteres. A fin de superar estas limitaciones, se han desarrollado otros varios sistemas de archivos:

- *Extended File System*, que extendía las posibilidades del sistema de archivos Minix, pero ofrecía un rendimiento mediocre;
- *Xia File System*, basado fuertemente en el sistema de archivos Minix, que extendía sus posibilidades ofreciendo un rendimiento razonable;
- *Second Extended File System* [Card et al. 1994], segunda versión del *Extended File System*, que extiende las posibilidades ofreciendo un buen rendimiento. Este sistema de archivos se inspira en la versión presente en BSD Unix [McKusick et al. 1984], pero posee numerosas extensiones originales.

Las características generales de estos sistemas de archivos son las siguientes:

	Minix	Ext FS	Ext2 FS	Xia FS
Tamaño máximo	64 MB	2 GB	4 TB	2 GB
Tamaño máximo de un archivo	64 MB	2 GB	2 GB	64 MB
Tamaño máximo de nombre de archivo	14 c	255 c	255 c	248 c
Soporte de las tres fechas	No	No	Sí	Sí
Posibilidades de extensión	No	No	Sí	No
Tamaño de bloque variable	No	No	Sí	No
Mantenido	Sí	No	Sí	No

Además de estos sistemas de archivos nativos, Linux incluye el soporte de sistemas de archivos que ofrece una cierta compatibilidad con otros sistemas operativos. Se trata de MS/DOS, Windows 95, OS/2, Unix System V, BSD Unix, soporte de NFS (*Networked File System*) que permite compartir archivos en la red, y un tipo de sistema de archivos particular, llamado */proc*, que permite acceder a los datos mantenidos por el núcleo.

En el resto de este capítulo se detallan la estructura y la implementación de los sistemas de archivos Ext2 y */proc*.

6.5 Implementación del sistema de archivos Ext2

6.5.1 Características de Ext2

El sistema de archivos Ext2 ofrece funcionalidades estándar. Soporta los archivos Unix (archivos regulares, directorios, archivos especiales, enlaces simbólicos) y ofrece funcionalidades avanzadas:

- Pueden asociarse atributos a los archivos para modificar el comportamiento del núcleo; los atributos reconocidos son los siguientes:
 - supresión segura: cuando el archivo se suprime, su contenido se destruye previamente con datos aleatorios;
 - *undelete*: cuando el archivo se suprime, se guarda automáticamente a fin de poder restaurarlo ulteriormente (aún no se ha implementado);
 - compresión automática: la lectura y la escritura de datos en el archivo da lugar a una compresión al vuelo (aún no se ha implementado en el núcleo estándar);
 - escrituras síncronas: toda modificación sobre el archivo se escribe de manera síncrona en disco;
 - inmutable: el archivo no puede modificarse ni suprimirse;
 - adición exclusiva: el archivo sólo puede modificarse si se ha abierto en modo adición, y no puede suprimirse.
- Compatibilidad con las semánticas de Unix System V Release 4 o BDS: una opción de montaje permite elegir el grupo asociado a los nuevos archivos: la semántica BSD especifica que el grupo se hereda desde el directorio padre, mientras que SVR4 utiliza el número de grupo primario del proceso que llama.
- Enlaces simbólicos «rápidos»: ciertos enlaces simbólicos no utilizan bloque de datos: el nombre del archivo destino está contenido directamente en el i-nodo en disco, lo que permite economizar el espacio en disco y acelerar la resolución de estos enlaces evitando una lectura de bloque.
- El estado de cada sistema de archivos se memoriza: cuando el sistema de archivos se monta, se marca como inválido hasta que se desmonte. El verificador de la estructura, *e2fsck*, utiliza este estado para acelerar las verificaciones cuando no son necesarias.

- Un contador de montaje y una demora máxima entre dos verificaciones pueden utilizarse para forzar la ejecución de *e2fsck*.
- El comportamiento del código de gestión puede adaptarse en caso de error: puede mostrar un mensaje de error, «remontar» el sistema de archivos en lectura exclusiva a fin de evitar una corrupción de los datos, o provocar un error del sistema.

Además, Ext2 incluye numerosas optimizaciones. En las lecturas de datos, se efectúan lecturas anticipadas. Ello significa que el código de gestión pide la lectura no sólo del bloque que necesita, sino también de otros bloques consecutivos. Esto permite cargar en memoria bloques que se usarían en las entradas/salidas siguientes. Este mecanismo se utiliza también en las lecturas de entradas de directorio, ya sean explícitas (por la primitiva *readdir*) o implícitas (en la resolución de nombres de archivos en la operación sobre el i-nodo *lookup*).

Las asignaciones de bloques e i-nodos también se han optimizado. Se usan grupos de bloques para agrupar los i-nodos emparentados así como sus bloques de datos. Un mecanismo de preasignación permite también asignar bloques consecutivos a los archivos: cuando debe asignarse un bloque, se reservan hasta 8 bloques consecutivos. De este modo, las asignaciones de bloques siguientes ya se han satisfecho y el contenido de archivos tiende a escribirse en bloques contiguos, lo que acelera su lectura, especialmente gracias a las técnicas de lectura anticipada.

6.5.2 Estructura física de un sistema de archivos Ext2

Un sistema de archivos, de tipo Ext, debe estar presente sobre un dispositivo físico (disquete, disco duro...) y el contenido de este dispositivo se descompone lógicamente en varias partes, como lo muestra la figura 6.7.

Sector de boot	Conjunto de bloques 1	Conjunto de bloques 2	...	Conjunto de bloques 3
----------------	-----------------------	-----------------------	-----	-----------------------

FIG. 6.7 – Estructura de un sistema de archivos Ext2

El sector de boot contiene el código máquina necesario para cargar el núcleo en el arranque del sistema, y cada uno de los grupos de bloques se descompone a su vez en varios elementos como muestra la figura 6.8:

- una copia del superbloque: esta estructura contiene las informaciones de control del sistema de archivos y se duplica en cada grupo de bloques para permitir paliar fácilmente una corrupción del sistema de archivos;

- una tabla de descriptores: estos últimos contienen las direcciones de bloques que contienen las informaciones cruciales, como los bloques de *bitmap* y la tabla de i-nodos; también se duplican en cada grupo de bloques;
- un bloque de *bitmap* para los bloques: este bloque contiene una tabla de bits: a cada bloque del grupo se le asocia un bit indicando si el bloque está asignado (el bit está entonces a 1) o disponible (el bit está a 0);
- una tabla de i-nodos: estos bloques contienen una parte de la tabla de i-nodos del sistema de archivos;
- bloques de datos: el resto de los bloques del grupo se utiliza para almacenar los datos contenidos en los archivos y los directorios.

Super-bloque	Descriptores	Bitmap bloques	Bitmap i-nodos	Tabla de i-nodos	Bloque de datos
--------------	--------------	----------------	----------------	------------------	-----------------

FIG. 6.8 – Estructura de un grupo de bloques

Un sistema de archivos se organiza en archivos y directorios. Un directorio es un archivo de tipo particular, que contiene entradas. Cada una de las entradas de directorio contiene varios campos:

- el número de i-nodo correspondiente al archivo;
- el tamaño de la entrada en bytes;
- el número de caracteres que componen el nombre del archivo;
- el nombre del archivo.

La figura 6.9 representa un directorio que contiene las entradas «.», «..», «arch1», «nombre_de_archivo_largo», y «f2».

i1	12	01	.
i2	12	02	..
i3	16	05	arch1
i4	28	19	nombre_archivo_largo
i5	12	02	f2

FIG. 6.9 – Estructura de un directorio

6.5.3 El superbloque

El superbloque contiene las informaciones de control del sistema de archivos. Se sitúa al principio del sistema de archivos en disco (en el desplazamiento 1024) y se duplica

en cada grupo de bloques para permitir su restauración en caso de corrupción de la copia primaria.

La estructura `ext2_super_block` define el formato del superbloque. Se define en el archivo `<linux/ext2_fs.h>` y contiene los campos siguientes:

tipo	campo	descripción
_u32	<code>s_inodes_count</code>	Número total de i-nodos
_u32	<code>s_blocks_count</code>	Número total de bloques
_u32	<code>s_r_blocks_count</code>	Número de bloques reservados al superusuario
_u32	<code>s_free_blocks_count</code>	Número de bloques libres
_u32	<code>s_free_inodes_count</code>	Número de i-nodos libres
_u32	<code>s_first_data_block</code>	Número del primer bloque de datos
_u32	<code>s_log_block_size</code>	Tamaño lógico de los bloques
_u32	<code>s_blocks_per_group</code>	Número de bloques por grupo
_u32	<code>s_frags_per_group</code>	Número de fragmentos por grupo
_u32	<code>s_inodes_per_group</code>	Número de i-nodos por grupo
_u32	<code>s_mtime</code>	Fecha del último montaje del sistema de archivos
_u32	<code>s_wtime</code>	Fecha de la última escritura del superbloque
_u16	<code>s_mnt_count</code>	Número de montajes del sistema de archivos
_s16	<code>s_max_mnt_count</code>	Número máximo de montajes
_u16	<code>s_magic</code>	Firma del sistema de archivos
_u16	<code>s_state</code>	Estado del sistema de archivos
_u16	<code>s_errors</code>	Comportamiento del sistema de archivos en caso de errores
_u16	<code>s_minor_rev_level</code>	Número de revisión
_u32	<code>s_lastcheck</code>	Fecha de la última verificación del sistema de archivos
_u32	<code>s_checkinterval</code>	Tiempo máximo entre dos verificaciones
_u32	<code>s_creator_os</code>	Identificador del sistema operativo bajo el cual se ha creado el sistema de archivos
_u16	<code>s_def_resuid</code>	Identificador del usuario que puede usar los bloques reservados al superusuario de modo predeterminado
_u16	<code>s_def_resgid</code>	Identificador del grupo que puede usar los bloques reservados al superusuario de modo predeterminado

6.5.4 Los descriptores de grupo de bloques

Cada grupo de bloques contiene una copia del superbloque así como una copia de descriptores de grupos. Estos descriptores contienen las coordenadas de las estructuras de control presentes en cada grupo.

La estructura `ext2_group_desc` define el formato de un descriptor de grupo. Se define en el archivo `<linux/ext2_fs.h>` y contiene los campos siguientes:

tipo	campo	descripción
_u32	<code>bg_block_bitmap</code>	Dirección del bloque de <i>bitmap</i> para los bloques de este grupo
_u32	<code>bg_inode_bitmap</code>	Dirección del bloque de <i>bitmap</i> para los i-nodos de este grupo
_u32	<code>bg_inode_table</code>	Dirección del primer bloque de la tabla de i-nodos en este grupo

<code>_u16</code>	<code>bg_free_blocks_count</code>	Número de bloques libres en este grupo
<code>_u16</code>	<code>bg_free_inodes_count</code>	Número de i-nodos libres en este grupo
<code>_u16</code>	<code>bg_used_dirs_count</code>	Número de directorios asignados en este grupo
<code>_u16</code>	<code>bg_pad</code>	No utilizado
<code>_u32[3]</code>	<code>bg_reserved</code>	Campo reservado para futura extensión

6.5.5 Estructura de un i-nodo

La tabla de i-nodos se descompone en varias partes: cada parte está contenida en un grupo de bloques. Esto permite utilizar estrategias de asignación particulares: cuando un bloque debe asignarse, el núcleo intenta asignarlo en el mismo grupo que su i-nodo a fin de minimizar el desplazamiento de las cabezas de lectura/escritura en la lectura del archivo.

La estructura `ext2_inode` define el formato de un i-nodo. Se declara en el archivo `<linux/ext2_fs.h>` y contiene los campos siguientes:

tipo	campo	descripción
<code>_u16</code>	<code>i_mode</code>	Modo del i-nodo
<code>_u16</code>	<code>i_uid</code>	Identificador del propietario
<code>_u32</code>	<code>i_size</code>	Tamaño del archivo en bytes
<code>_u32</code>	<code>i_atime</code>	Fecha de último acceso al archivo
<code>_u32</code>	<code>i_ctime</code>	Fecha de última modificación del i-nodo
<code>_u32</code>	<code>i_mtime</code>	Fecha de última modificación del contenido del archivo
<code>_u32</code>	<code>i_dtime</code>	Fecha de supresión del archivo
<code>_u16</code>	<code>i_gid</code>	Identificador de grupo
<code>_u16</code>	<code>i_links_count</code>	Número de enlaces asociados al i-nodo
<code>_u32</code>	<code>i_blocks</code>	Número de bloques de 512 bytes asignados al i-nodo
<code>_u32</code>	<code>i_flags</code>	Atributos asociados al archivo
<code>_u32</code>	<code>i_reserved1</code>	Campo reservado para extensión futura
<code>_u32 [EXT2_N_BLOCKS]</code>	<code>i_block</code>	Direcciones de bloques de datos asignados al i-nodo
<code>_u32</code>	<code>i_version</code>	Número de versión asociado al i-nodo
<code>_u32</code>	<code>i_file_acl</code>	Dirección del descriptor de la lista de control de acceso asociada al archivo (no está implementado en Linux 2.0)
<code>_u32</code>	<code>i_dir_acl</code>	Dirección del descriptor de la lista de control de acceso asociada a un directorio (no está implementado en Linux 2.0)
<code>_u16</code>	<code>i_pad1</code>	No utilizado
<code>_u32[2]</code>	<code>i_reserved2</code>	Campo reservado para extensión futura

El campo `i_block` contiene las direcciones de bloques de datos asociadas al i-nodo. Esta tabla se estructura según el método clásico de Unix [Bach 1993, McKusick *et al.* 1996, Goodheart and Cox 1994, Vahalia 1996]:

- los primeros doce elementos (valor de la constante `EXT2_NDIR_BLOCKS`) de la tabla contienen las direcciones de bloques de datos:

- el puesto EXT2_IND_BLOCK contiene la dirección de un bloque que contiene a su vez la dirección de los bloques de datos siguientes;
- el puesto EXT2_DIND_BLOCK contiene la dirección de un bloque que contiene la dirección de bloques que contienen la dirección de los bloques de datos siguientes;
- el puesto EXT2_TIND_BLOCK contiene la dirección de un bloque que contiene la dirección de bloques que apuntan a su vez a bloques indirectos.

Este mecanismo de direccionamiento se ilustra en la figura 6.10 (limitándose a dos niveles de indirección por razones de claridad).

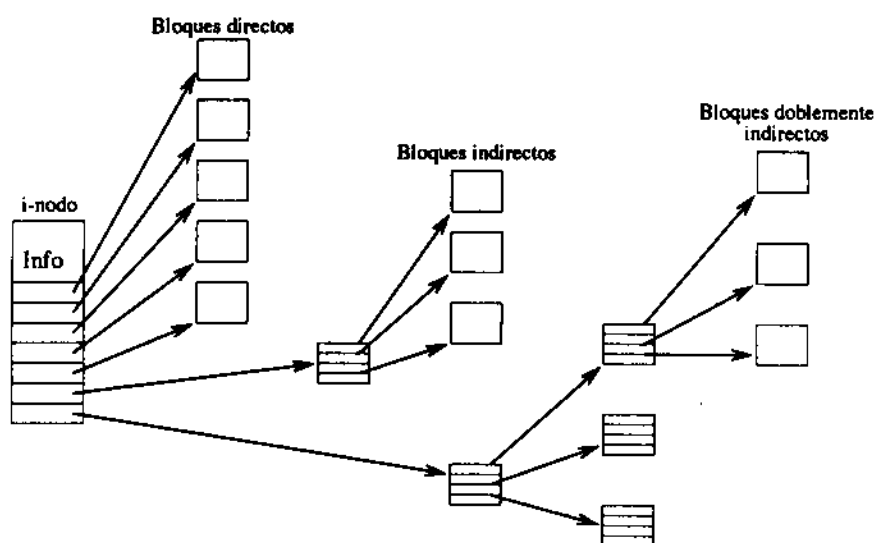


FIG. 6.10 – Punteros contenidos en un i-nodo

6.5.6 Entrada de directorio

Los directorios se componen de bloques de datos, como los archivos regulares. Sin embargo, estos bloques se estructuran lógicamente en una serie de entradas. La estructura `ext2_dir_entry` define el formato de estas entradas. Se declara en el archivo `<linux/ext2_fs.h>` y contiene los campos siguientes:

tipo	campo	descripción
<code>_u32</code>	<code>inode</code>	Número del i-nodo del archivo
<code>_u16</code>	<code>rec_len</code>	Tamaño en bytes de la entrada de directorio
<code>_u16</code>	<code>name_len</code>	Número de caracteres que componen el nombre del archivo
<code>char[]</code>	<code>name</code>	Nombre del archivo

6.5.7 Operaciones vinculadas al sistema de archivos

Las operaciones relacionadas con el superbloque de un sistema de archivos Ext2 se implementan en el archivo fuente *fs/ext2/super.c*. La variable `ext2_sops` contiene los punteros a las funciones que aseguran estas operaciones.

Las funciones `ext2_error`, `ext2_panic` y `ext2_warning` son llamadas por el código de gestión del sistema de archivos cuando se detecta un error. Muestran un mensaje de error o de advertencia con la identificación del dispositivo afectado y la función interna que haya detectado el error llamando a la función `printk`.

Se utilizan varias funciones internas para el montaje de un sistema de archivos:

- `parse_options`: esta función analiza las opciones de montaje especificadas. Analiza la cadena de caracteres pasada como parámetro e inicializa las opciones de montaje.
- `ext2_setup_super`: esta función inicializa el descriptor de superbloque a partir del superbloque del sistema de archivos leído desde el disco.
- `ext2_check_descriptors`: esta función verifica la validez de los descriptors de conjuntos leídos desde el disco. Para cada descriptor, verifica que los bloques de *bitmap* y la tabla de i-nodos están contenidos en el grupo.
- `ext2_commit_super`: esta función se llama para guardar las modificaciones efectuadas en el superbloque. Marca la memoria intermedia que contiene el superbloque como modificada, llamando a `mark_buffer_dirty`. De este modo, el contenido de la memoria intermedia se reescribirá en disco en el próximo guardado del *búfer caché*.

La función `ext2_read_super` implementa la operación sobre el sistema de archivos `read_super`. Se llama al montar un sistema de archivos. Empieza por analizar las opciones de montaje llamando a `parse_options`, luego lee el superbloque desde el disco, y verifica su validez. Inicializa seguidamente el descriptor de superbloque. Luego se asigna una tabla de punteros para los descriptors de grupos, cada descriptor se carga en una memoria intermedia, llamando a la función `bread`, y se verifica la validez de estos descriptors por `ext2_check_descriptors`. Finalmente, se lee el i-nodo de la raíz del sistema de archivos en memoria llamando a `iget`, y se llama a `ext2_setup_super` para terminar la inicialización del descriptor de superbloque.

La función `ext2_write_super` implementa la operación `write_super`, que actualiza los campos `s_mtime` (fecha de última modificación del superbloque) y

`s_state` para indicar que el sistema de archivos está montado; luego llama a la función `ext2_commit_super`.

La función `ext2_write_super` implementa la operación sobre sistema de archivos `remount_fs`. Llama a `parse_options` para decodificar las nuevas opciones de montaje, actualiza el descriptor de superbloque, y llama a `ext2_commit_super` para indicar que el superbloque se ha modificado.

La función `ext2_put_super` implementa la operación sobre sistema de archivos `put_super`. Se llama cuando un sistema de archivos se desmonta. Libera las memorias intermedias que contienen los descriptors del sistema de archivos, llamando a la función `brelse`, y libera los punteros a esas memorias llamando a `kfree_s`. Seguidamente, libera las memorias intermedias asociadas a los bloques de *bitmap* cargados en memoria. Finalmente, libera la memoria intermedia que contiene el superbloque del sistema de archivos.

Para terminar, la función `ext2_statfs` implementa la operación sobre sistema de archivos `statfs`: copia las estadísticas de uso del sistema de archivos desde el descriptor de superbloque en la variable pasada como parámetro.

6.5.8 Asignación y liberación de bloques e i-nodos

Las funciones de asignación y liberación de bloques e i-nodos se implementan en los archivos fuente `fs/ext2/balloc.c` y `fs/ext2/ialloc.c`.

Estos módulos definen funciones internas:

- `get_group_desc`: esta función devuelve el descriptor correspondiente a un grupo de bloques;
- `read_block_bitmap`, `read_inode_bitmap`: estas funciones cargan en memoria un bloque de *bitmap*.
- `load_block_bitmap`, `load_inode_bitmap`: estas funciones se llaman para obtener un bloque de *bitmap*. Mantienen un caché LRU de los bloques cargados en las tablas `s_block_bitmap` y `s_inode_bitmap`, y llaman a `read_block_bitmap` y `read_inode_bitmap` para cargar los bloques en memoria.

La liberación de bloques se efectúa por la función `ext2_free_blocks`. Esta función verifica que los números de bloques a liberar son válidos, luego carga el descriptor y el bloque de *bitmap* del grupo afectado llamando a `get_group_desc` y `load_block_bitmap`. Seguidamente, el bit correspondiente a cada bloque a librar se pone a 0 en el bloque de *bitmap*, se llama a la operación sobre cuota

`free_block` asociada al i-nodo por cada bloque, y se incrementa el número de bloques libres. Finalmente, el superbloque y el bloque de *bitmap* se marcan como modificados llamando a `mark_buffer_dirty`.

La función `ext2_new_block` implementa la asignación de bloque. Esta función acepta como parámetro `goal`, que especifica el número de bloque a asignar si está disponible. Tras haber cargado el bloque de *bitmap*, `ext2_new_block` comprueba si el bloque especificado está disponible. Si es así, se elige. Si no, se hace una búsqueda de un bloque libre en las cercanías de este bloque y, si la búsqueda fracasa, se busca un byte a cero en el bloque de *bitmap* del grupo, es decir, que se buscan 8 bloques libres consecutivos o, al menos, un solo bloque disponible. Tras estas distintas comprobaciones, si no se ha encontrado ningún bloque disponible, `ext2_new_block` explora todos los grupos de bloques para encontrar un bloque disponible. Cuando se ha encontrado un bloque, la operación sobre cuota `alloc_block` se llama para verificar que el proceso que llama puede asignar el bloque, y el bit correspondiente del bloque *bitmap* se pone a 1, para indicar que el bloque ya no está disponible.

Tras haber asignado un bloque, `ext2_new_block` intenta proceder a una preasignación de bloques consecutivos. Efectúa un bucle de asignación de los bloques siguientes llamando a la operación sobre cuota `alloc_block` y marcando los bits correspondientes en el bloque de *bitmap* a 1. Finalmente, se asigna una memoria intermedia para el bloque llamando a `getblk`, su contenido se pone a 0, y se actualizan las informaciones de descriptores de grupos y del superbloque.

La liberación de un i-nodo se efectúa por la función `ext2_free_inode`. Esta función verifica la validez del i-nodo a liberar, y carga el descriptor y el bloque de *bitmap* del grupo afectado llamando a `get_group_desc` y `load_inode_bitmap`. Seguidamente, el bit correspondiente al i-nodo a liberar se pone a 0 en el bloque de *bitmap*, el número de i-nodos libres se incrementa, y se llama a la operación sobre cuota `free_inode` asociada al i-nodo. Finalmente, el superbloque y el bloque de *bitmap* se marcan como modificados llamando a `mark_buffer_dirty`.

La función `ext2_new_inode` implementa la asignación de i-nodo. El algoritmo de asignación depende del tipo del i-nodo:

- Si hay que asignar un directorio, se efectúa una búsqueda para elegir el grupo de bloques que posee un número de i-nodos libres superior a la media, y en el que el número de bloques libres sea máximo;
- en los demás casos, `ext2_new_inode` intenta utilizar el conjunto de grupos que contienen el i-nodo del directorio padre. Si no es posible, se efectúa una búsqueda cuadrática de un grupo que contenga i-nodos libres. Si esta búsqueda fracasa, se utiliza finalmente una búsqueda lineal, grupo por grupo.

Una vez elegido el grupo de bloques a utilizar, el bloque de *bitmap* se carga en memoria llamando a `load_inode_bitmap`, y se busca el primer bit a 0, correspondiente a un i-nodo no utilizado; se pone el bit a 1, y el número de i-nodos disponibles se decrementa. El descriptor de i-nodo se inicializa seguidamente y se inserta en las listas de hash llamando a `insert_inode_hash`. Finalmente, si hay asociadas operaciones sobre cuotas al superbloque del sistema de archivos, las operaciones `initialize` y `alloc_inode` se llaman para tener en cuenta la asignación.

6.5.9 Gestión de los i-nodos en disco

El archivo fuente `fs/ext2/inode.c` contiene las funciones de gestión de i-nodos en disco.

La operación sobre i-nodo `bmap` implementa varias funciones:

- `inode_bmap`: esta función devuelve la dirección de un bloque contenido en el i-nodo.
- `block_bmap`: esta función devuelve la dirección de un bloque obtenida en una tabla contenida en un bloque de datos. Se utiliza para acceder a las direcciones de bloques indirectos.
- `ext2_bmap`: esta función implementa la operación `bmap`. Obtiene la dirección del bloque de datos especificado llamando a `inode_bmap` y `block_bmap` para efectuar las diversas indirecciones.

Hay varias funciones relacionadas con la asignación de bloques:

- `ext2_discard_prealloc`: esta función se llama al cerrar un archivo. Libera los bloques asignados previamente por `ext2_new_block` sin usar aún llamando a `ext2_free_blocks`.
- `ext2_alloc_block`: esta función se llama para asignar un bloque. Gestiona la preasignación: si un bloque ha sido preasignado y corresponde al bloque especificado por el parámetro `goal`, se utiliza; si no los bloques preasignados se liberan llamando a `ext2_discard_prealloc`, y se llama a `ext2_new_block` para asignar un bloque.
- `inode_getblk`: esta función se llama para obtener una memoria intermedia que contiene un bloque cuya dirección se almacena en el i-nodo. Utiliza `getblk` para obtener la memoria intermedia, si el bloque está ya asignado. Si el bloque no está asignado y si quien llama ha especificado una creación de bloque, se llama a la función `ext2_alloc_block` para obtener un nuevo bloque.

- `block_getblk`: esta función se llama para obtener una memoria intermedia que contenga un bloque cuya dirección se almacena en una tabla contenida en un bloque de datos. Es muy similar a `inode_getblk`.
- `ext2_getblk`: esta función se llama para obtener una memoria intermedia conteniendo un bloque asociado a un i-nodo. De manera similar a `ext2_bmap`, llama a `inode_getblk` y a `block_getblk` para efectuar las diversas indirecciones.
- `block_getcluster`: esta función intenta crear un *clúster* referido a varios bloques directos de un i-nodo. Comprueba que los bloques especificados son contiguos en disco, y llama `generate_cluster` para crear el *clúster*.
- `ext2_getcluster`: esta función intenta crear un *clúster* referido a varios bloques de un i-nodo. Efectúa un tratamiento muy similar al de `ext2_getblk`, excepto que llama a `block_getcluster` en lugar de `block_getblk`.
- `ext2_bread`: esta función se llama para leer un bloque de datos asociado a un i-nodo. Llama a `ext2_getblk` para obtener la memoria intermedia correspondiente, luego lee el contenido del bloque llamando a `ll_rw_block` si el contenido de la memoria no está actualizado.

La función `ext2_read_inode` implementa la operación sobre sistema de archivos `read_inode`. Verifica primero la validez del número de i-nodo a cargar, luego calcula el número del grupo que contiene dicho i-nodo, y lee el bloque de la tabla de i-nodos correspondiente llamando a la función `bread`. El contenido del i-nodo en disco se copia seguidamente en el descriptor de i-nodo, y el campo `i_op` (puntero a las operaciones vinculadas al i-nodo) se inicializa según su tipo.

La función `ext2_update_inode` reescribe un i-nodo en disco. Verifica primero la validez del número de i-nodo a escribir, luego calcula el número del grupo que contiene este i-nodo, y lee el bloque de la tabla de i-nodos correspondiente llamando a la función `bread`. El contenido del i-nodo en disco se modifica seguidamente en la memoria intermedia a partir del descriptor del i-nodo y la memoria se marca como modificada llamando a `mark_buffer_dirty`. Finalmente, si el i-nodo debe reescribirse inmediatamente en disco, se llama a la función `ll_rw_block` para proceder a la escritura física.

La función `ext2_write_inode` implementa la operación `write_inode`. Llama a la función `ext2_update_inode` especificando que la reescritura del i-nodo no debe efectuarse inmediatamente. La función `ext2_sync_inode` efectúa la misma tarea pero exige una reescritura inmediata al llamar a `ext2_update_inode`.

La función `ext2_put_inode` implementa la operación sobre sistema de archivos `put_inode`. Libera los bloques preasignados llamando a `ext2_discard_pre-`

lloc, luego comprueba el número de enlaces para determinar si el archivo se ha suprimido. Si es así, la fecha de supresión del i-nodo se actualiza, el i-nodo se reescribe en disco llamando a `ext2_update_inode`, el contenido del archivo se libera llamando a `ext2_truncate`, y finalmente se llama a `ext2_free_inode` para liberar el i-nodo.

6.5.10 Gestión de directorios

El archivo fuente *fs/ext2/dir.c* contiene las funciones de gestión de directorios:

- `ext2_chack_dir_entry`: esta función se llama para controlar la validez de una entrada de directorio. Si se detecta un error, muestra un mensaje llamando a la función `ext2_error`.
- `ext2_readdir`: esta función implementa la operación `readdir`. Utiliza la función `ext2_bread` para leer cada bloque que compone el directorio, e implementa una estrategia de lectura anticipada. Cada bloque leído se descompone en entradas de directorios que se colocan en la memoria intermedia utilizando el puntero a función `filldir` pasado como parámetro (este parámetro contiene la dirección de las funciones `fillonedir` o `filldir`, definidas en *fs/readdir.c*, como se explica en la sección 6.3.5). `ext2_readdir` utiliza los números de versión asociados a los descriptors del archivo abierto y del i-nodo correspondiente: si estos números son diferentes, significa que el directorio se ha modificado (al menos un archivo ha sido añadido o suprimido), y se efectúa un bucle de resincronización para colocar el puntero de posición actual al principio de una entrada válida.

Otras funciones de gestión de directorios se definen también en el archivo fuente *fs/ext2/namei.c*:

- `ext2_match`: esta función interna compara un nombre de archivo especificado con el nombre contenido en una entrada de directorio.
- `ext2_find_entry`: esta función se llama para buscar una entrada en un directorio. Explora el contenido del directorio obteniendo cada bloque por una llamada a `ext2_getblk`. Se utiliza una estrategia de lectura anticipada. Cada bloque se descompone en una entrada de directorio y se llama a la función `ext2_match` por cada entrada para comparar el nombre que contiene con el nombre buscado. Si se encuentra el nombre, se devuelve la dirección de la entrada.
- `ext2_lookup`: esta función implementa la operación `lookup`. Efectúa primero una búsqueda del nombre de archivo en el caché de nombres llamando a `dcache_lookup`. Si el nombre se encuentra en el caché, el i-nodo correspondiente se lee llamando a `iget`, y se devuelve. Si no, se busca el nombre en el directorio llamando a la fun-

ción `ext2_find_entry`, el resultado se añade en el caché de nombre por `dca-che_add`, el i-nodo se carga en memoria llamando a `iget`, y se devuelve.

- `ext2_add_entry`: esta función se llama para crear una nueva entrada en un directorio. Efectúa primero una exploración del directorio leyendo cada bloque llamando a `ext2_bread`, y buscando una entrada utilizable, es decir, una entrada disponible de tamaño suficiente o una entrada utilizada que pueda descomponerse en dos entradas. Una vez se encuentra una entrada utilizable, se inicializa con el nombre de archivo especificado.
- `ext2_delete_entry`: esta función se llama para suprimir una entrada de directorio. Libera la entrada:
 - poniendo a 0 el número de i-nodo de la entrada si se trata de la primera entrada de un bloque;
 - fusionando esta entrada con la anterior en el caso contrario.

Otras numerosas funciones, que implementan las operaciones sobre i-nodos, se definen en `fs/ext2/namei.c`: `ext2_create`, `ext2_link`, `ext2_mkdir`, `ext2_mknod`, `ext2_rename`, `ext2_rmdir`, `ext2_symlink` y `ext2_unlink`. Estas funciones son relativamente simples, porque se limitan a encadenar las llamadas de las otras funciones internas, y su funcionamiento no se detalla aquí.

6.5.11 Entradas/salidas sobre archivos

Sólo la operación sobre archivo `write` se implementa específicamente. La lectura de datos se efectúa llamando a la función `generic_file_read`, que se detalla en el capítulo 8, sección 6.8, debido a sus interacciones con los mecanismos de gestión de la memoria.

La función `ext2_file_write`, definida en el archivo fuente `fs/ext2/file.c`, implementa la operación `write`, que verifica sus argumentos y efectúa un bucle de escritura: mientras queden datos por escribir, obtiene una memoria intermedia llamando a `ext2_getblk`, copia una parte de los datos a escribir en la memoria, y marca la memoria como modificado llamando a `mark_buffer_dirty`. Una vez escritos todos los datos, el descriptor de archivo y el i-nodo se actualizan.

La función `ext2_release_file` implementa la operación sobre archivo `release`. Llama a `ext2_discard_prealloc` para liberar los bloques preasignados por `ext2_new_block` en el último cierre del archivo.

- *rtc*: informaciones relacionadas con el reloj en tiempo real;
- *stat*: estadísticas diversas sobre las operaciones efectuadas por el núcleo (tiempo de procesador consumido, número de entradas/salidas en disco, número de cargas de páginas en memoria, número de interrupciones de hardware tratadas, número de cambios de contexto efectuados, fecha y hora de arranque del sistema, y número total de procesos creados);
- *smpt*: informaciones relacionadas con el tratamiento multiprocesadores;
- *uptime*: tiempo pasado desde el arranque del sistema;
- *version*: versión del núcleo.

Además de estos archivos, el directorio */proc* contiene varios directorios:

- *net*: archivos que contienen las informaciones relacionadas con los protocolos de red;
- *scsi*: archivos que contienen las informaciones relacionadas con los gestores de dispositivos SCSI;
- *sys*: archivos que contienen las informaciones relacionadas con las variables del núcleo gestionadas por la primitiva *sysctl* (véase el capítulo 13, sección 2.4);
- un directorio por proceso existente en el sistema: el nombre de este directorio es el número del proceso, y este directorio contiene los archivos siguientes:
 - *cmdline*: lista de argumentos del proceso;
 - *cwd*: enlace al directorio actual del proceso;
 - *environ*: lista de variables de entorno del proceso;
 - *exe*: enlace al archivo binario ejecutado por el proceso;
 - *fd*: directorio que contiene enlaces a los archivos abiertos por el proceso;
 - *maps*: lista de zonas de memoria contenidas en el espacio de direccionamiento del proceso;
 - *mem*: contenido del espacio de direccionamiento del proceso;
 - *root*: enlace al directorio raíz del proceso;
 - *stat*, *statm*, *status*: estado del proceso.
- *self*: enlace con el directorio correspondiente al proceso actual.

6.6 Implementación del sistema de archivos */proc*

6.6.1 Presentación

El sistema de archivos */proc* es algo particular: no da acceso a datos almacenados en disco, sino que hace accesibles, en forma de archivos virtuales, ciertas informaciones gestionadas por el núcleo.

Los archivos accesibles en el sistema de archivos */proc* son los siguientes:

- *cmdline*: argumentos pasados al núcleo en el arranque del sistema;
- *cpuinfo*: descripción del procesador o procesadores utilizados;
- *devices*: lista de gestores de dispositivos incluidos en el núcleo;
- *dma*: lista de canales DMA utilizados por los gestores de dispositivos;
- *filesystems*: lista de tipos de sistemas de archivos soportados por el núcleo;
- *interrupts*: lista de interrupciones de hardware utilizadas por los gestores de dispositivos;
- *ioports*: lista de puertos de entrada/salida utilizados por los gestores de dispositivos;
- *kcore*: memoria asignada al núcleo;
- *kmsg*: últimos mensajes mostrados por el núcleo;
- *ksyms*: lista de símbolos del núcleo utilizables por módulos (véase el capítulo 12, sección 3.3);
- *loadavg*: carga del sistema;
- *locks*: lista de bloqueos asociados a los archivos;
- *meminfo*: estado de ocupación de la memoria central;
- *modules*: lista de módulos cargados en el núcleo;
- *mounts*: lista de sistemas de archivos montados;
- *pci*: lista de dispositivos conectados en el bus PCI;
- *profile*: informaciones de *profiling* del núcleo, utilizadas para determinar el tiempo pasado en ejecutar cada una de las funciones;

6.6.2 Entradas de */proc*

Las entradas del directorio */proc* (archivos o directorios) se gestionan de manera dinámica: una lista de descriptores es mantenida en memoria por el núcleo, y se explora accediendo al contenido de */proc*.

La estructura `proc_dir_entry`, definida en el archivo `<linux/proc_fs.h>`, representa el tipo de estos descriptores. Contiene los campos siguientes:

tipo	campo	descripción
unsigned short	low_ino	Número de i-nodo asociado a la entrada
unsigned short	namelen	Tamaño del nombre de la entrada
const char *	name	Nombre de la entrada
mode_t	mode	Tipo y derechos de acceso
nlink_t	nlink	Número de enlaces
uid_t	uid	Identificador del usuario propietario
gid_t	gid	Identificador del grupo de usuarios asociado
unsigned long	size	Tamaño en bytes
struct	ops	Operaciones relacionadas con la entrada
inode_operations *		
int (*)(char *, char **, off_t, int, int)	get_info	Puntero a la función llamada en una lectura
void (*)(struct	fill_inode	Puntero a la función encargada de inicializar los atributos de la entrada (tipo, derechos de acceso, usuario y grupo propietarios)
inode)		
struct	next	Puntero al descriptor de la entrada siguiente
proc_dir_entry *		
struct	parent	Puntero al descriptor del directorio padre
proc_dir_entry *		
struct	subdir	Puntero al descriptor de la primera entrada del directorio
proc_dir_entry *		
void *	data	Datos privados asociados a la entrada

Los números de i-nodos se asignan estáticamente a las entradas de */proc*. El archivo de cabecera define varios tipos en este sentido:

- `root_directory_inos`: números de i-nodos asociados a las entradas de */proc*, comprendidos entre 1 y 127;
- `net_directory_inos`: números de i-nodos asociados a las entradas de */proc/net*, comprendidos entre 128 y 255;
- `scsi_directory_inos`: números de i-nodos asociados a las entradas de */proc/scsi*, comprendidos entre 256 y 511.

Además, las constantes `PROC_DYNAMIC_FIRST` y `PROC_NDYNAMIC` definen los números de i-nodos que pueden asignarse de manera dinámica.

Los números de i-nodos asociados a las entradas de los directorios correspondientes a los procesos se calculan según el número del proceso. Este número se desplaza 16 bits hacia la izquierda para generar un número básico, y el tipo `pid_directory_inos` define un número a añadir a esta base para obtener el número de i-nodo. Por ejemplo, el número de i-nodo del archivo *root* contenido en el directorio correspondiente al proceso de número *p* será $p * 65536 + 6$, porque la constante `PROC_PID_ROOT` tiene el valor 6.

6.6.3 Operaciones sobre sistema de archivos

Las operaciones relacionadas con el sistema de archivos se implementan en el archivo fuente *fs/proc/root.c*.

La función `proc_get_inode` se encarga de inicializar el contenido de un descriptor de i-nodo: llama a `iget` para cargar el i-nodo, e inicializa algunos de sus campos desde el descriptor de la entrada correspondiente. En esta función, la dirección del descriptor de entrada se guarda en el campo `generic_ip` del descriptor de i-nodo a fin de poder acceder en la ejecución de las operaciones sobre i-nodos.

La función `proc_read_super` se llama en el montaje del sistema de archivos */proc*. Inicializa el descriptor de archivos y llama a la función `proc_get_inode` para inicializar el i-nodo correspondiente a la raíz del sistema de archivos.

La función `proc_put_super` se llama al desmontar el sistema de archivos. Se limita a poner a cero el campo `s_dev` del descriptor afectado para indicar que el sistema de archivos ya no está montado.

La función `proc_read_inode` se llama para leer el contenido de un i-nodo. Calcula un número de proceso desplazando el número de i-nodo de 16 bits hacia la derecha, y efectúa una búsqueda del descriptor de proceso correspondiente en la tabla `task`.

Tras esta búsqueda, pueden presentarse tres casos:

1. El número de proceso no se ha encontrado.
2. El número de i-nodo corresponde a una entrada estática de */proc*. El campo `i_op` del descriptor del i-nodo se inicializa entonces con la dirección de las operaciones sobre i-nodos específicos del archivo (por ejemplo, `proc_kmsg_inode_operations` para el archivo */proc/kmsg*).
3. El número de proceso ha sido hallado: los 8 bits de menor peso del número de i-nodo se utilizan entonces para determinar cuál es la entrada del directorio correspondiente al proceso. El campo `i_op` del descriptor del i-nodo se inicia-

liza con la dirección de las operaciones sobre i-nodos específicos del archivo (por ejemplo, `proc_mem_inode_operations` para el archivo *mem*).

6.6.4 Gestión de directorios

El archivo fuente *fs/proc/root.c* contiene las funciones que permiten gestionar el contenido de los directorios.

La variable global `proc_root` contiene el descriptor de la raíz del sistema de archivos.

La función `proc_register` registra una nueva entrada en un directorio, y la función `proc_unregister` suprime una entrada borrando su descriptor de la lista.

La función `proc_register_dynamic` atribuye dinámicamente un número de i-nodo a un descriptor y guarda la entrada correspondiente en la lista.

La inicialización de la lista de archivos y directorios situados en */proc* se efectúa por `proc_root_init`. Esta función llama a la función `proc_register` para registrar las entradas contenidas en */proc*.

La función `proc_lookup` efectúa la exploración de un directorio. Explora la lista encadenada de las entradas registradas en el directorio y compara el nombre especificado con el nombre de archivo contenido en cada descriptor. `proc_root_lookup` efectúa la búsqueda de un nombre de entrada en el directorio raíz. Llama a `proc_lookup` y comprueba su resultado. Si `proc_lookup` ha encontrado la entrada especificada (por tanto, si el nombre se refiere a una entrada estática registrada llamando a `proc_register`), se devuelve el resultado. En el caso contrario, el nombre especificado debe representar un directorio correspondiente a un proceso. El nombre se convierte en número de proceso, se efectúa una búsqueda del proceso correspondiente en la tabla `task`, y se construye el número de i-nodo correspondiente a partir del número de proceso.

La función `proc_readdir` se llama para obtener una lista de entradas contenidas en un directorio. Se basa en el valor del campo `f_pos` del descriptor de archivo correspondiente al directorio: este campo contiene el índice de la entrada a devolver. `proc_readdir` explora la lista de entradas registradas en el directorio, y llama a la función especificada por el parámetro `filldir`, a fin de colocar los nombres de las entradas en la memoria intermedia proporcionada por el usuario. La función `proc_root_readdir` implementa la llamada `readdir` para el directorio raíz. Si la posición en el directorio es inferior a `FIRST_PROCESS_ENTRY`, llama a `proc_readdir` para obtener los nombres de entradas registradas. Luego explora la

tabla *task*, convierte los números de procesos en cadenas de caracteres, y las coloca en la memoria intermedia proporcionada por la llamada a la función especificada por el parámetro *filldir*.

Se utiliza un solo descriptor para referenciar todos los directorios correspondientes a procesos: la variable *proc_pid* definida en el archivo fuente *fs/proc/base.c*. Su contenido se inicializa por la función *proc_base_init*, llamada por *proc_root_init*. Esta función llama a *proc_register* para registrar las entradas *cmdline*, *cwd*, *environ*, *exe*, *fd*, *maps*, *mem*, *root*, *stat*, *statm* y *status*.

6.6.5 Operaciones sobre i-nodos y sobre archivos

El sistema de archivos */proc* utiliza los punteros a las operaciones sobre i-nodos y sobre archivos abiertos para diferenciar los tratamientos asociados a las entradas. La mayor parte de los tratamientos se implementa en el archivo fuente *fs/proc/array.c*.

La función *array_read* implementa la operación sobre archivo *read*. Asigna primero una página de memoria, que servirá de memoria intermedia llamando a *__get_free_page*, y obtiene el descriptor de entrada asociada al i-nodo. Seguidamente, llama a la operación *get_info* asociada a la entrada, o la función *fill_array* si no se ha definido ninguna operación. Finalmente, copia el resultado devuelto en la memoria intermedia proporcionada por el proceso que llama.

La función *fill_array* se llama para leer el contenido de un archivo, cuando la entrada correspondiente no posee operación *get_info*. Llama a *get_process_array* si el i-nodo está contenido en un directorio correspondiente a un proceso, o *get_root_array* en caso contrario. Estas dos funciones comprueban el número de i-nodo del archivo, y llaman a una función que se encarga de convertir los datos internos del núcleo en una cadena de caracteres.

Entradas/salidas

Primitivas detalladas

ioctl, mknod, select

1 Conceptos

El acceso a los dispositivos se efectúa mediante archivos especiales. Un archivo especial aparece en el árbol de la misma manera que un archivo regular, pero no se le asigna ningún espacio en disco. A cada archivo especial corresponde un controlador de dispositivo cuyo código se integra al núcleo.

Una vez que un proceso ha abierto un archivo especial, sus peticiones de lectura y escritura no se transmiten al sistema de archivos, sino al controlador de dispositivo correspondiente. Este último efectúa las entradas/salidas físicas sobre el dispositivo, generalmente dialogando con su controlador, cuando el proceso utiliza las llamadas al sistema `read` y `write`.

Existen dos tipos de archivos especiales:

- Los archivos especiales en modo bloque: corresponden a dispositivos estructurados en bloques, como los discos, a los que se accede proporcionando un número de bloque a leer o escribir. Las entradas/salidas se efectúan mediante las funciones del *búfer caché*.
- Los archivos especiales en modo carácter: corresponden a dispositivos no estructurados, como los puertos serie y paralelo, sobre los que se puede leer y escribir los datos byte a byte, generalmente de manera secuencial.

Cada archivo especial se caracteriza por tres atributos:

- su tipo (bloque o carácter):
- su número mayor: este número identifica el controlador que gestiona el dispositivo;
- su número menor: este número permite al controlador conocer el dispositivo físico sobre el cual debe actuar.

Los archivos especiales se encuentran generalmente en el directorio `/dev`, del que damos un extracto de la lista en la figura 7.1.

```
...
brw-r----- 1 root disk 3, 0 Oct 3 1993 hda
brw-r----- 1 root disk 3, 1 Oct 3 1993 hda1
brw-r----- 1 root disk 3, 2 Oct 3 1993 hda2
brw-r----- 1 root disk 3, 3 Oct 3 1993 hda3
brw-r----- 1 root disk 3, 4 Oct 3 1993 hda4
brw-r----- 1 root disk 3, 5 Oct 3 1993 hda5
brw-r----- 1 root disk 3, 5 Oct 3 1993 hda6
...
crw--w--w- 1 card users 4, 0 Oct 3 1993 tty0
crw--w--w- 1 card tty 4, 1 May 16 12:53 tty1
crw--w--w- 1 root wheel 4, 2 May 16 12:53 tty2
crw--w--w- 1 root wheel 4, 3 May 16 12:53 tty3
crw--w--w- 1 root wheel 4, 4 May 16 12:53 tty4
cew--w--w- 1 root wheel 4, 5 May 16 12:19 tty5
...
```

FIG. 7.1 – Lista de archivos especiales en el directorio `/dev`

2 Llamadas al sistema

2.1 Creación de un archivo especial

La llamada al sistema `mknod` permite crear un archivo especial. Su sintaxis es la siguiente:

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

```
int mknod (const char *pathname, mode_t mode, dev_t dev);
```

El parámetro `pathname` especifica el nombre de archivo a crear, `mode` indica los permisos y el tipo de archivo a crear, y `dev` contiene el identificador del dispositivo correspondiente al archivo especial. El archivo `<sys/stat.h>` define constantes utilizables para el tipo de archivo:

constante	significado
S_IFREG	Archivo regular
S_IFCHR	Archivo especial en modo carácter
S_IFBLK	Archivo especial en modo bloque
S_IFIFO	Tubería con nombre (véase el capítulo 10)

El identificador del dispositivo se compone de sus números mayor y menor. Varias macroinstrucciones, definidas en `<sys/sysmacros.h>`, permiten manipular este identificador:

macroinstrucción	significado
major	Devuelve el número mayor correspondiente a un identificador de dispositivo
minor	Devuelve el número menor correspondiente a un identificador de dispositivo
makedev	Devuelve el identificador de dispositivo correspondiente a un número mayor y a un número menor

`mknod` devuelve el valor 0 en caso de éxito o el valor -1 en caso de error. Los errores posibles son los siguientes:

error	significado
EACCES	El proceso que llama no tiene los derechos necesarios para crear el archivo especificado por <code>pathname</code>
EXIST	<code>pathname</code> especifica un nombre de archivo existente
EINVAL	<code>mode</code> contiene un tipo no válido
EFAULT	<code>pathname</code> contiene una dirección no válida
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENAMETOOLONG	<code>pathname</code> especifica un nombre de archivo demasiado largo
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOSPC	El sistema de archivos está saturado
ENOTDIR	Uno de los componentes de <code>pathname</code> , utilizado como nombre de directorio, no es un directorio
EPERM	<code>mode</code> especifica un tipo distinto de S_IFIFO y el proceso que llama no posee los derechos del superusuario
EROFS	El sistema de archivos es de lectura exclusiva

2.2. Entradas/salidas sobre dispositivos

Las entradas/salidas sobre dispositivos se efectúan por las mismas primitivas que las utilizadas para leer y escribir datos en archivos regulares. La apertura de un dispositivo se efectúa por la primitiva `open` especificando el nombre de un archivo especial. El núcleo devuelve entonces un descriptor de entradas/salidas correspondiente al disposi-

tivo y el proceso que llama puede acceder por las llamadas al sistema *read*, *write* y *lseek* (si el dispositivo soporta el acceso directo). Tras la utilización, puede utilizarse la primitiva *close* para cerrar el dispositivo.

2.3 Multiplexado de entradas/salidas

Al acceder a un dispositivo o a varios dispositivos, un proceso puede quedar en espera de datos. La primitiva *select* permite efectuar un multiplexado sobre varios descriptors de entradas/salidas. Su sintaxis es la siguiente:

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

int select (int n, fd_set *readfs, fd_set *writefds, fd_set
            *exceptfds, struct timeval *timeout);
```

select pone al proceso actual en espera de cambio sobre varios grupos de descriptors:

- *readfs* contiene la lista de descriptors desde los que el proceso que llama desea leer datos;
- *writefds* contiene la lista de descriptors en los que el proceso que llama desea escribir datos;
- *exceptfds* contiene la lista de descriptors para los que el proceso actual desea estar informado de cambios excepcionales.

El parámetro *n* debe contener el número del descriptor de entradas/salidas más elevado presente en uno de los grupos. El parámetro *timeout* especifica la demora de espera. Si contiene el valor *NULL*, el proceso que llama se suspende hasta que inter venga un cambio que afecte a uno de los descriptors.

Se proporcionan varias macroinstrucciones para gestionar los grupos de descriptors:

macroinstrucción	significado
FD_ZERO	Inicialización a cero de un grupo
FD_CLR	Supresión de un descriptor en un grupo
FD_SET	Adición de un descriptor en un grupo
FD_ISSET	Comprobación de presencia de un descriptor en un grupo

Al volver de la llamada *select*, los grupos de descriptors se modifican para indicar las condiciones detectadas. Se devuelve el número de descriptors afectados.

En caso de error, *select* devuelve el valor -1 y *errno* toma uno de los valores siguientes:

<i>error</i>	<i>significado</i>
EINVAL	Se ha especificado un descriptor de entrada/salida incorrecto en uno de los grupos
EINTR	Se ha recibido una señal durante la espera
EINVAL	n contiene un valor negativo
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos

El programa siguiente utiliza la primitiva *select* para esperar la pulsación de al menos un carácter sobre la entrada estándar durante 5 segundos.

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

void main (void)
{
    fd_set      readfds;
    struct timeval  timeout;
    char        c;
    int         r;

    /* Inicialización del grupo de descriptores */
    FD_ZERO (&readfds);
    FD_SET (0, &readfds);
    /* Espera durante 5 segundos */
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;

    /* Espera de un carácter durante 5 segundos */
    r = select (1, &readfds, NULL, NULL, &timeout);

    /* Comprobación de presencia de un carácter */
    if (r == 1) {
        /* Lectura del carácter */
        read (0, &c, sizeof (c));
        printf ("Carácter leído: %c\n", c);
    } else
        printf ("Ningún carácter escrito durante 5 segundos\n");
}
```

2.4 Operación de control sobre un dispositivo

Las entradas/salidas sobre dispositivos se efectúan gracias a las primitivas estándar. Pero existe una llamada al sistema, llamada *ioctl*, permitiendo modificar los parámetros de un dispositivo correspondiente a un descriptor de entradas/salidas. La sintaxis de la llamada al sistema *ioctl* es la siguiente:

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

int ioctl (int fd, int cmd, char *arg);
```

La primitiva *ioctl* permite modificar el estado del dispositivo asociado al descriptor especificado por el parámetro *fd*. El parámetro *cmd* indica la operación a efectuar, y *arg* apunta a una variable cuyo tipo depende de la operación a efectuar.

Los códigos correspondientes a las operaciones de control se definen en el archivo de cabecera *<sys/ioctl.h>*, que incluye numerosos archivos de cabecera. Es imposible detallar aquí las operaciones de control debido a su extrema diversidad. Se aconseja el estudio de los archivos de cabecera del núcleo para escribir un programa que trabaje con dispositivos.

En caso de éxito, *ioctl* devuelve el valor 0. En caso de error, se devuelve el valor -1 y *errno* toma uno de los valores siguientes:

<i>error</i>	<i>significado</i>
EBADF	El descriptor de entrada/salida no es válido
ENOTTY	La operación especificada no puede aplicarse al dispositivo correspondiente al descriptor <i>fd</i>
EINVAL	La operación especificada por <i>cmd</i> o el argumento <i>arg</i> no es válido

Según la operación efectuada y el dispositivo afectado, pueden devolverse otros errores de diferentes naturalezas.

3 Presentación general de la implementación

3.1 Dispositivos soportados por el núcleo

Linux actualiza la lista de controladores de dispositivos soportados, tanto si estos gestores se han incluido en la compilación del núcleo como si se han cargado en forma de módulos.

Para ello, utiliza dos tablas: `blkdevs` y `chrdevs`, que contienen respectivamente los descriptors de dispositivos en modo bloque y los de los dispositivos en modo carácter. La estructura `device_struct`, definida en el archivo fuente `fs/devices.c`, caracteriza el tipo de cada una de las entradas de estas tablas. Contiene los campos siguientes:

tipo	campo	descripción
<code>const char*</code>	<code>name</code>	Nombre del dispositivo gestionado
<code>struct file_operations *</code>	<code>fops</code>	Operaciones sobre archivos asociados al dispositivo

3.2 Entradas/salidas en disco

Las entradas/salidas efectuadas en dispositivos en modo bloque se satisfacen por medio del *búfer caché*: los bloques se cargan en memorias intermedias llamando a las funciones `bread` y `breada`. Estas últimas llaman a un módulo de entradas/salidas a fin de leer o escribir bloques en disco.

La función de este módulo consiste en mantener la lista de peticiones de entradas/salidas en curso y satisfacerlas. Cuando uno o más bloques deben leerse o escribirse en disco, la petición de entradas/salidas se añade a la lista correspondiente al dispositivo físico. Esta lista es explorada por el controlador de dispositivo que efectúa las entradas/salidas una a una.

A fin de optimizar los tiempos de desplazamiento de las cabezas de lectura/escritura en disco, la lista de peticiones se mantiene ordenada por el número de sector físico. De este modo, las peticiones que respectan a sectores próximos se efectúan minimizando el desplazamiento de las cabezas, según el principio del ascensor [Silberschatz y Calvin 1994].

La estructura `request`, definida en el archivo `<linux/blkdev.h>`, especifica el tipo de los elementos de la lista de peticiones. Estos contienen los campos siguientes:

tipo	campo	descripción
<code>volatile int</code>	<code>rq_status</code>	Estado de la petición (ver más adelante)
<code>kdev_t</code>	<code>rq_dev</code>	Identificador del dispositivo
<code>int</code>	<code>cmd</code>	Mandato: READ o WRITE
<code>int</code>	<code>errors</code>	Número de errores físicos detectados en la entrada/salida
<code>unsigned long</code>	<code>sector</code>	Sector de inicio
<code>unsigned long</code>	<code>nr_sectors</code>	Número de sectores a leer o escribir
<code>unsigned long</code>	<code>current_nr_sectors</code>	Número de sectores restantes para leer o escribir
<code>char *</code>	<code>buffer</code>	Dirección de los datos a leer o escribir

<code>struct semaphore *</code>	<code>sem</code>	Semáforo utilizado para la sincronización de los accesos concurrentes a esta petición
<code>struct buffer_head *</code>	<code>bh</code>	Puntero a la memoria intermedia que contiene los datos a leer o escribir
<code>struct buffer_head *</code>	<code>bhtail</code>	Puntero a la última memoria intermedia de la petición
<code>struct request *</code>	<code>next</code>	Puntero a la petición siguiente

El campo `rq_status` puede tomar los valores siguientes:

<i>opción</i>	<i>significado</i>
<code>RQ_INACTIVE</code>	Elemento de la lista disponible
<code>RQ_ACTIVE</code>	Elemento de la lista ocupado
<code>RQ SCSI_BUSY</code>	Petición en curso
<code>RQ SCSI_DONE</code>	Petición satisfecha

4 Presentación detallada de la implementación

4.1 Gestión de los dispositivos efectuados

El archivo fuente `fs/devices.c` contiene las funciones de gestión de dispositivos soportados. Este módulo mantiene la lista de controladores presentes en el núcleo.

Se utilizan dos tablas para mantener la lista de dispositivos soportados: `blkdevs` contiene los descriptores de dispositivos en modo bloque, y `chrdevs` contiene los descriptores de dispositivos en modo carácter.

Las funciones `register_blkdev` y `register_chrdev` permiten registrarse a los controladores de dispositivos. Añaden un descriptor de dispositivos en la tabla correspondiente: `blkdevs` para los dispositivos accesibles en modo bloque o `chrdevs` para los dispositivos accesibles en modo carácter. Las funciones `unregister_blkdev` y `unregister_chrdev` permiten a un controlador de dispositivo darse de baja, por ejemplo en el caso de un módulo de gestión de dispositivo que se suprime del núcleo en curso de ejecución, modificando la entrada correspondiente en la tabla `blkdevs` o `chrdevs`.

Las funciones `get_blkfops` y `get_chrfops` devuelven un puntero a las operaciones sobre archivos asociados a un dispositivo.

La apertura de dispositivo se gestiona por las funciones `blkdev_open` y `chrdev_open`. Éstas obtienen las operaciones sobre archivos asociados al dispositivo llamando a `get_blkfops` o `get_chrfops`, y efectúan la apertura llamando a la ope-

ración open. La función `blkdev_release` se llama en el último cierre de un dispositivo en modo bloque: obtiene las operaciones sobre archivos asociadas al dispositivo llamando a `get_blkfops`, y llama a la operación `release`.

Este módulo exporta diversas variables:

- `def_blk_fops`: operaciones sobre archivos asociados a los dispositivos en modo bloque;
- `blkdev_inode_operations`: operaciones sobre i-nodos asociados a los dispositivos en modo bloque;
- `def_chr_fops`: operaciones sobre archivos asociados a los dispositivos en modo carácter;
- `chrdev_inode_operations`: operaciones sobre i-nodos asociados a los dispositivos en modo carácter.

4.2 Entradas/salidas de disco

Las funciones de gestión de las listas de peticiones de entradas/salidas de disco se implementan en el archivo fuente `drivers/block/ll_rw_blk.c`.

Se definen varias funciones:

- `get_request`: esta función busca un descriptor de petición libre en la lista global. El descriptor encontrado se marca seguidamente como activo e inicializado.
- `__get_request_wait`, `get_request_wait`: el papel de estas dos funciones es similar al de `get_request`, excepto que ponen al proceso que llama en espera si no hay libre ningún descriptor de petición.
- `add_request`: esta función se llama para añadir una petición de entrada/salida a la lista correspondiente a un dispositivo, manteniendo la lista ordenada.
- `make_request`: esta función se llama para crear una petición de entradas/salidas. Primero intenta fusionar la petición con una petición ya grabada en la lista, si los números de sector son adyacentes. Si no es posible, llama a `__get_request_wait` para obtener un descriptor de petición, lo inicializa, y lo inserta en la lista del dispositivo correspondiente llamando a `add_request`.
- `ll_rw_block`: esta función la llama el *búfer caché*, los sistemas de archivos, y el módulo de entradas/salidas de bloques, a fin de efectuar una entrada/salida. Verifica la validez de sus argumentos, y crea una petición de entrada/salida llamando a

`make_request`. La memoria intermedia correspondiente se bloquea mientras la entrada/salida no se haya efectuado y se desbloqueará al final de la entrada/salida. De este modo, el proceso que llama puede ponerse en espera del fin de la entrada/salida llamando a `wait_on_buffer`.

4.3 Entradas/salidas sobre dispositivos en modo bloque

Las funciones de lectura y escritura de datos sobre dispositivos accesibles en modo bloque se implementan en el archivo fuente `fs/block_dev.c`. Estas funciones utilizan las primitivas del *búfer caché* para acceder a las memorias intermedias asociadas a los dispositivos. Las funciones `block_write`, `block_read` y `block_sync` implementan las operaciones sobre archivos `write`, `read` y `fsync` asociados a los dispositivos en modo bloque.

La función `block_write` implementa la escritura de datos sobre un dispositivo. Genera *clústers* para todos los bloques a escribir llamando a `generate_cluster` y utiliza un mecanismo de lectura anticipada para cargar en memoria los bloques siguientes. Copia los datos a escribir desde la dirección especificada por quien llama en las memorias intermedias creadas, y escribe estas memorias por grupos llamando a `ll_rw_block`.

La función `block_read` implementa la lectura de datos desde un dispositivo, utilizando un mecanismo de lectura anticipada. Utiliza la función `generate_cluster` para generar *clústers* correspondientes a los bloques a leer, y crea una sola petición de entrada/salida para todos los bloques llamando a `ll_rw_block`. Efectúa seguidamente un bucle por todas las memorias intermedias cuya lectura se ha activado: espera el fin de la lectura del bloque y copia su contenido a la dirección proporcionada por quien llama.

Las dos funciones son bastante complejas porque utilizan a la vez la lectura anticipada y los *clústers*.

La función `block_fsync` reescribe los datos modificados asociados a un dispositivo. Provoca la reescritura de las memorias intermedias llamando a `fsync_dev`.

4.4 Multiplexado de entradas/salidas

4.4.1 Principio

El multiplexado de entradas/salidas por la primitiva `select` se implementa en el archivo fuente `fs/select.c`. Su principio es relativamente simple: para cada descriptor de

entrada/salida especificado, se efectúa una comprobación para determinar si es posible la entrada/salida. Si no es así, el proceso actual se coloca en la cola de espera correspondiente al archivo.

La lista de colas de espera sobre las que está suspendido el proceso actual se memoriza en una tabla de multiplexado. Cada entrada de la tabla se define por la estructura `select_table_entry`, definida en el archivo de cabecera `<linux/wait.h>`:

tipo	campo	descripción
<code>struct wait_queue</code>	<code>wait</code>	Entrada correspondiente al proceso actual en la cola de espera
<code>struct wait_queue **</code>	<code>wait_address</code>	Dirección de la cola de espera en la que está registrado el proceso actual

La estructura `select_table_struct` define el formato de los descriptores de la tabla de multiplexado, que contiene los campos siguientes:

tipo	campo	descripción
<code>int</code>	<code>nr</code>	Número de entradas en la tabla
<code>struct select_table_entry *</code>	<code>entry</code>	Dirección de la página de memoria que contiene la tabla de multiplexado

Al ejecutarse la primitiva `select`, el proceso actual se registra en las colas de espera correspondientes a los descriptores proporcionados, y la tabla de multiplexado contiene la lista de estas colas de espera. De este modo, cuando el proceso actual se suspende, puede ser despertado por la llamada a las funciones `wake_up` y `wake_up_interruptible` sobre cualquiera de las colas de espera. De este modo, el proceso se despierta cuando es posible una entrada/salida sobre uno de los descriptores de archivos especificados.

A fin de implementar la demora proporcionada en la llamada de la primitiva `select`, puede posicionarse el campo `timeout` del descriptor de proceso actual. Así se implementa una suspensión con demora de la forma descrita en el capítulo 4, sección 6.1.4.

4.4.2 Funciones de utilidad

La función `select_wait`, definida en el archivo de cabecera `<linux/sched.h>`, añade una entrada a la tabla de multiplexado, y registra esta entrada en la cola de espera especificada.

La función `free_wait` suprime todas las entradas de una tabla de multiplexado de las colas de espera en las que se han registrado. Explora toda la tabla, y llama a `remove_wait_queue` para cada entrada.

La función `check` se llama para verificar si una entrada/salida es posible sobre el archivo especificado. Llama a la operación `select` asociada al archivo.

4.4.3 La operación sobre archivo *select*

La operación sobre archivo `select` debe comprobar si la entrada/salida especificada es posible sobre el archivo indicado. Si es así, debe devolver el valor 1; si no debe añadir una entrada en la tabla de multiplexado llamando a `select_wait`, y devolver el valor 0.

A título de ejemplo, estudiemos el funcionamiento de la función `random_select`, que implementa la operación `select` para el generador de números aleatorios. Esta función se implementa en el archivo fuente `drivers/char/random.c`, y su código es el siguiente:

```
int random_select (struct inode *inode, struct file *file, int
                  sel_type, select_table *wait)
{
    switch (sel_type) {
    case SEL_IN:
        if (random_state.entropy_count >= 8)
            return 1;
        select_wait(&random_wait, wait);
        break;
    case SEL_OUT:
        if (random_state.entropy_count < WAIT_OUTPUT_BITS)
            return 1;
        select_wait(&random_wait, wait);
        break;
    }
    return 0;
}
```

Esta función distingue dos tipos de acceso:

- el acceso en lectura (`SEL_IN`): si al menos están disponibles ocho números aleatorios, es posible la lectura, y `random_select` devuelve el valor 1;
- el acceso en escritura (`SEL_OUT`): si la memoria de números aleatorios no está saturada, la escritura es posible y `random_select` devuelve el valor 1.

En el caso en que la entrada/salida no sea posible, se llama a `select_wait` para crear una entrada en la tabla de multiplexado y registrar dicha entrada en la cola de espera `random_wait`; `random_select` devuelve entonces el valor 0.

La cola de espera `random_wait` se utiliza de manera interna por el gestor de números aleatorios. Cuando se generan nuevos números aleatorios, o cuando se libera espacio en la memoria intermedia, las funciones internas llaman a `wake_up_interruptible` para despertar los procesos en espera.

4.4.4 Implementación de la primitiva *select*

La función `do_select` implementa el multiplexado propiamente dicho. Verifica primero la validez de sus parámetros, controlando los descriptors de archivos especificados, y luego inicializa una tabla de multiplexado. Posiciona seguidamente el estado del proceso actual a `TASK_INTERRUPTIBLE`, y efectúa un bucle sobre todos los descriptors especificados. Para cada descriptor, se llama la función `check` a fin de determinar si se posible la entrada/salida. Si es así, se incrementa el número de descriptors utilizables. Tras este bucle, se comprueba el número de descriptors utilizables: si es nulo, y el proceso no ha recibido ninguna señal, `schedule` se llama para provocar una ordenación, y vuelve a empezar la exploración de los descriptors especificados. Si el número de descriptors no es nulo, o si el proceso actual ha recibido una señal, el tratamiento se para: la tabla de multiplexado se libera, el estado del proceso se pone a `TASK_RUNNING`, y `do_select` devuelve el número de descriptors calculados.

La primitiva *select* se implementa por `sys_select`. Esta función verifica la validez de sus parámetros, convierte la demora especificada en número de ciclos de reloj, y llama a `do_select` para proceder al multiplexado propiamente dicho. Tras esta llamada, coloca el resultado del multiplexado en los parámetros proporcionados por el proceso que llama.

4.5 Gestión de las interrupciones

El controlador de un dispositivo transmite generalmente una interrupción al procesador cuando cambia de estado, por ejemplo cuando termina una entrada/salida o cuando se produce un error físico. El controlador del dispositivo debe reaccionar en la recepción de este tipo de interrupción.

Linux contiene un módulo de gestión de interrupciones. Este módulo de bajo nivel se ocupa de la programación física del controlador de interrupciones y ofrece funciones que permiten manipular fácilmente las interrupciones. El archivo fuente `arch/i386/kernel/irq.c` contiene las funciones de este módulo.

Las funciones `disable_irq` y `enable_irq` permiten respectivamente inhibir y activar una interrupción de hardware. Impiden toda interrupción, ocultan o hacen visi-

ble la interrupción deseada programando el controlador de interrupciones, y restablece el estado del procesador.

Las funciones de desvío de interrupción son muy similares a las funciones de gestión de las señales (véase capítulo 5): un controlador de dispositivo puede especificar una función a ejecutar en la recepción de una interrupción de hardware. Cuando se recibe esta interrupción, todas las funciones asociadas se llaman una por una para tratar el evento.

Las funciones `do_IRQ` y `do_fast_IRQ` se llaman al recibir una interrupción de hardware. Exploran la lista de funciones de tratamiento de la interrupción, y llaman a cada una de estas funciones.

La función `request_irq` permite a un controlador de dispositivo asociar una función de tratamiento a una interrupción de hardware: un descriptor de interrupción se asigna, se inicializa y se añade a la lista de funciones de tratamiento de la interrupción.

La función `free_irq` permite a un controlador de dispositivo desactivar su función de tratamiento de una interrupción de hardware. Explora la lista correspondiente a la interrupción, suprime el descriptor correspondiente y lo libera.

4.6 Gestión de los canales DMA

Al usar la transferencia de datos en modo DMA (*Direct Memory Access*), el controlador de dispositivo lee o escribe los datos directamente en memoria central, utilizando una dirección física que se le transmite con la petición de entrada/salida.

A cada controlador de dispositivo que funciona en modo DMA se le asocia un canal DMA. Para dialogar con el controlador, el gestor de dispositivo debe reservar el canal, a fin de ser el único que lo utilice.

Linux proporciona un mecanismo de reserva y liberación de canales DMA. El archivo fuente `kernel/dma.c` contiene las funciones correspondientes.

Los canales DMA reservados se almacenan en la tabla `dma_chan_busy`. Cada elemento de la tabla es una estructura (estructura `dma_chan`) y contiene los campos siguientes:

tipo	campo	descripción
int	lock	Booleano que indica si el canal está reservado
const char *	device_id	Puntero a una cadena que identifica el gestor de dispositivo que ha reservado el canal

La función `request_dma` permite a un gestor de dispositivos la reserva de un canal DMA. Verifica la validez del número de canal, y comprueba si el canal está ya reservado. Si es así, se devuelve el error `EBUSY`; si no se memoriza el nombre del gestor, y se devuelve el valor 0.

La función `free_dma` libera un canal DMA (por ejemplo, cuando un gestor de dispositivo cargado en forma de módulo se descarga de la memoria). Comprueba la validez del número de canal, y marca el canal como disponible modificando el puesto correspondiente de `dma_chan_busy`.

El código del sistema de archivos `/proc` llama a una tercera función, `get_dma_list`, que devuelve la representación ASCII de los puestos reservados de la tabla `dma_chan_busy`, bajo la forma siguiente:

2: floppy
4: cascade
6: aha1542

4.7 Acceso a los puertos de entrada/salida

El diálogo entre un gestor de dispositivos y el controlador correspondiente se efectúa utilizando puertos de entrada/salida. Estos puertos se pueden direccionar con las instrucciones de ensamblador `in` y `out`.

Al estar los gestores de dispositivos escritos en lenguaje C, Linux ofrece diversas funciones, definidas en el archivo de cabecera `<asm/io.h>`, que permiten acceder a los puertos de entrada/salida:

- `outb`: escritura de un byte en un puerto;
- `inb`: lectura de un byte desde un puerto;
- `outw`: escritura de una palabra de 16 bits en un puerto;
- `inw`: lectura de una palabra de 16 bits desde un puerto;
- `outl`: escritura de una palabra de 32 bits en un puerto;
- `inl`: lectura de una palabra de 32 bits desde un puerto;
- `outsb`: escritura de una serie de bytes en un puerto;
- `insb`: lectura de una serie de bytes desde un puerto;

- `outsw`: escritura de una serie de palabras de 32 bits en un puerto;
- `insw`: lectura de una serie de palabras de 32 bits desde un puerto;
- `outs1`: escritura de una serie de palabras de 32 bits en un puerto;
- `ins1`: lectura de una serie de palabras de 32 bits desde un puerto.

Existen variantes de cada una de estas funciones, con el sufijo `_p`, para provocar una espera de unos ciclos de reloj tras la lectura o la escritura en el puerto.

4.8 Ejemplo de dispositivo en modo bloque: el disco en memoria

4.8.1 Presentación

El archivo fuente *drivers/block/rd.c* contiene la implementación del gestor de disco en memoria, el cual permite reservar un área de memoria y utilizarla como un disco virtual.

La constante `NUM_RAMDISKS`, que posee el valor 16, especifica el número de discos de memoria que pueden ser gestionados por este módulo. Las variables `rd_length` y `rd_blocksize` contienen el tamaño en bytes y el tamaño de los bloques relacionados con cada uno de los discos de memoria.

4.8.2 Acceso al contenido del disco en memoria

La función `rd_request` se llama para efectuar una entrada/salida sobre el contenido de un disco en memoria: ésta se encarga de acceder al contenido del disco. El gestor utiliza un método original para almacenar el contenido del disco en memoria: en lugar de declarar un área de memoria con el contenido del disco y copiar su contenido en las memorias intermedias en cada lectura, se utiliza el propio contenido de las memorias intermedias.

Estas memorias se proporcionan en operaciones de lectura y escritura por las funciones `block_read` y `block_write`:

- En una lectura, si existe una memoria intermedia correspondiente al bloc a leer, la función `block_read` utiliza directamente su contenido; si no se asigna una memoria intermedia y `rd_request` inicializa su contenido a cero.

- En una escritura, `block_write` transmite una memoria intermedia a `rd_request`. Esta última función posiciona el indicador `BH_Protected` para indicar que la memoria intermedia está en uso y no debe liberarse en ningún caso.

De este modo, el contenido del disco de memoria se almacena en las memorias intermedias:

- No se asocia ninguna memoria intermedia a los bloques que no han sido leídos, y se devuelve una memoria intermedia inicializada a cero en caso de lectura del bloque.
- Se asocia una memoria intermedia protegida a cada bloque que haya sido escrito, y su contenido se utiliza en la lectura del bloque.

4.8.3 Operaciones sobre archivos

Las operaciones sobre archivos `read` y `write` se implementan por medio de las funciones `block_read` y `block_write`, descritas en la sección 4.3. La función `rd_request` se llama para realizar las entradas/salidas.

La función `rd_ioctl` implementa la operación de archivo `ioctl`. Permite invalidar las memorias intermedias asociadas a un disco de memoria, u obtener el tamaño de un disco en memoria.

La operación `open` se implementa por medio de la función `rd_open`. Esta función inicializa el disco de memoria si su contenido debe cargarse (véase la sección 4.8.5).

La variable `fs_fops` contiene las direcciones de estas funciones.

4.8.4 Inicialización

La función `rd_init` se llama al iniciarse el sistema.

Esta función registra ante todo el gestor llamando a `register_blkdev` a la que pasa `fd_fops` como parámetro, a fin de indicarle las operaciones de archivo asociadas al disco de memoria. También inicializa el campo `request_fn` del puesto de la tabla `blk_dev` correspondiente al disco de memoria, para que la función `re_request` se llame en el tratamiento de una petición de entrada/salida. Finalmente, inicializa los descriptores de discos de memoria gestionados, modificando el contenido de las tablas `rd_length` y `rd_blocksizes`.

4.8.5 Carga de disco de memoria

La descripción presentada anteriormente corresponde al caso de un disco de memoria no inicializado: en el acceso al contenido del disco, las memorias intermedias correspondientes se crean automáticamente. Linux permite también inicializar el contenido de un disco de memoria desde un dispositivo. Esto resulta particularmente útil en el caso de un disquete de arranque, con un sistema de archivos que se carga en el disco de memoria creado al iniciar el sistema.

La carga del contenido de discos de memoria se efectúa cuando el núcleo se compila con la opción `CONFIG_BLK_DEV_INITRD`. En este caso, se llama a la función `initrd_load` en la inicialización del sistema. Ésta llama a `re_load_image` para cargar el contenido del disco de memoria. En caso contrario, se llama a `rd_load`, y esta función sólo llama a `rd_load_image` si la raíz del sistema de archivos está asociada al lector de disquetes.

La función `rd_load_image` carga el contenido de un disco de memoria desde un dispositivo. Se llama a la función `blk_open` para abrir el dispositivo especificado y el disco de memoria, identifica el tipo de la imagen a cargar llamando a `identify_ramdisk_image`, y procede a la carga en memoria, llamando a la operación `read` asociada al dispositivo fuente y la operación `write` asociada al disco de memoria, por cada bloque. Al terminar la carga, las memorias intermedias asociadas al dispositivo fuente se invalidan por una llamada a `invalidate_buffers`, y el dispositivo se cierra llamando a la operación de archivo `release` asociada.

La función `identify_ramdisk_image` se utiliza para identificar el tipo de contenido del dispositivo a cargar en el disco de memoria. Ésta reconoce los sistemas de archivos de tipo Minix y Ext2, así como las imágenes comprimidas, creadas por el mandato `gzip`, y devuelve el número de bloques contenidos en la imagen.

En el caso de un disco de memoria cuyo contenido se carga, se asigna una zona de memoria en la inicialización del núcleo (en el archivo fuente `arch/i386/kernel/setup.c`) y sus coordenadas están contenidas en las variables `initrd_start` (dirección de inicio de la zona) e `initrd_end` (dirección del fin de la zona).

En el caso de la carga de un disco de memoria, la función `rd_open` efectúa un tratamiento particular: incrementa el número de referencias al disco de memoria, y devuelve la variable `initrd_fops`, indicando que deben usarse funciones especiales en el acceso al contenido del disco.

La operación `read` se implementa entonces por la función `intrd_read`. Esta función copia el contenido de la zona de memoria reservada para el disco de memoria en la memoria intermedia proporcionada por el usuario, utilizando `memcpy_tofs`.

La función `intrd_release`, que implementa la operación `release`, decrementa el número de referencias al disco de memoria. Si este número llega a nulo, la zona de memoria asociada se libera llamando a `free_page` para cada página.

4.9 Ejemplo de dispositivo en modo carácter: la impresora

4.9.1 Funcionamiento

El gestor del puerto paralelo sobre arquitectura x86, implementado en el archivo fuente `drivers/char/lp.c`, es particularmente interesante, ya que puede utilizar dos métodos diferentes para dialogar con el puerto:

- las interrupciones: para señalar un evento, el puerto envía una interrupción que es tratada por el gestor;
- la exploración (*polling*): en lugar de utilizar el sistema de interrupciones, el gestor efectúa bucles de espera comprobando el registro de estado del puerto. Este método permite evitar la producción de numerosas interrupciones (una por carácter enviado a la impresora) y puede resultar más eficaz. Las operaciones de guardado y restauración del contexto del proceso actual, en el tratamiento de una interrupción, pueden ser costosas en el caso de un dispositivo que genere numerosas interrupciones.

4.9.2 Descripción de los puertos

Los puertos paralelos gestionados se definen por descriptores. Estos últimos se agrupan en la tabla `lp_table`. La estructura `lp_struct`, definida en el archivo de cabecera `<linux/lp.h>`, especifica el tipo de cada uno de los elementos. Contiene los campos siguientes:

tipo	campo	descripción
int	base	Dirección de base de los puertos de entrada/salida utilizados
unsigned int	irq	Número de la interrupción utilizada
int	flags	Estado de la impresora conectada
unsigned int	chars	Número de intentos a efectuar para imprimir un carácter
unsigned int	time	Duración de la suspensión para una espera, expresada en ciclos de reloj
unsigned int	wait	Número de bucles de espera a efectuar antes de que la impresora tenga en cuenta un carácter

<code>struct wait_queue *</code>	<code>lp_wait_q</code>	Cola de espera utilizada para esperar la llegada de una interrupción
<code>char *</code>	<code>lp_buffer</code>	Puntero a una memoria intermedia que contiene los caracteres a imprimir
<code>unsigned int</code>	<code>lastcall</code>	Fecha de la última escritura en la impresora
<code>unsigned int</code>	<code>runchars</code>	Número de caracteres escritos en la impresora sin provocar suspensión
<code>unsigned int</code>	<code>waittime</code>	Campo no utilizado
<code>struct lp_stats</code>	<code>stats</code>	Estadísticas sobre el uso de la impresora

4.9.3 Funciones de gestión de la impresora con exploración

La función `lp_char_polled` implementa el envío de un carácter en modo *polling*. Primero efectúa un bucle de espera hasta que la impresora esté a punto para recibir un carácter. Durante este bucle, se llama a la función `schedule`, si la variable `need_resched` está posicionada, a fin de evitar el bloqueo de un proceso más prioritario por una espera activa. Cuando la impresora está a punto, se le envía el carácter. Esta función devuelve el valor 1 si el carácter ha podido enviarse a la impresora, y el valor 0 en caso contrario.

La función `lp_write_polled` se llama para imprimir una serie de caracteres. Efectúa un bucle sobre cada carácter a imprimir. El carácter se envía llamando a `lp_char_polled`, y se verifica el resultado. Si el carácter se ha imprimido correctamente, `lp_write_polled` pasa al carácter siguiente. En caso de error, se muestra un mensaje llamando a `printk`, y el proceso se duerme por un tiempo determinado, modificando su estado a `TASK_INTERRUPTIBLE`, activando el campo `timeout` del descriptor de proceso, y llamando a la función `schedule` para provocar una ordenación. Tras cada suspensión, `lp_write_polled` comprueba si ha recibido alguna señal para el proceso durante su sueño. Si es así, se devuelve el número de caracteres imprimidos (o el error `EINTR` si aún no se ha imprimido ningún carácter). Si no, se suspende de nuevo durante cierto tiempo antes de volver al bucle de envío de caracteres. Al finalizar su ejecución, la función devuelve el número de caracteres que se han enviado a la impresora.

4.9.4 Funciones de gestión de la impresora con interrupciones

La función `lp_char_interrupt` implementa el envío de un carácter utilizando las interrupciones. Efectúa un bucle durante el cual espera que la impresora esté a punto para recibir un carácter y le envía el carácter. Esta función devuelve el valor 1 si el carácter se ha podido enviar a la impresora, y el valor 0 en caso contrario.

La función `lp_interrupt` se llama cuando se recibe una interrupción. Efectúa una búsqueda en la tabla `lp_table` para determinar a qué impresora se refiere la inte-

rupción, y llama a la función `wake_up` para despertar a los procesos en espera sobre el campo `lp_wait_q` del descriptor correspondiente.

La función `lp_write_interrupt` se llama para imprimir una serie de caracteres. Efectúa un bucle sobre cada carácter a imprimir. El carácter se envía llamando a `lp_char_interrupt`, y se comprueba el resultado. Si el carácter se ha imprimido correctamente, `lp_write_interrupt` pasa al carácter siguiente. En caso de error, se muestra un mensaje llamando a `printk`, y el proceso se duerme por un tiempo determinado, modificando su estado a `TASK_INTERRUPTIBLE`, posicionando el campo `timeout` del descriptor de proceso, y llamando a la función `interruptible_sleep_on` sobre el campo `lp_wait_q` del descriptor de la impresora. De este modo, el proceso será despertado por la expiración del tiempo especificado o por la llegada de una interrupción (porque `lp_interrupt` despierta los procesos en espera sobre `lp_wait_q`). Tras esta suspensión, `lp_write_interrupt` comprueba si se ha recibido una señal para el proceso durante su sueño. Si es así, se devuelve el número de caracteres imprimidos (o el error `EINTR` si no se ha imprimido aún ningún carácter). Si no, vuelve al bucle de envío de caracteres. Al final de su ejecución, la función devuelve el número de caracteres que se han enviado a la impresora.

4.9.5 Operaciones de entrada/salida sobre archivo

La función `lp_write` implementa la operación sobre archivo `write`. Ésta llama a la función `lp_write_interrupt` o a `lp_write_polled` según el modo de gestión de la impresora.

La función `lp_lseek` implementa la operación sobre archivo `lseek`, y devuelve el error `ESPIPE`.

La función `lp_open` implementa la operación sobre archivo `open`. Primero verifica que la impresora especificada existe, y que no está ya abierta por otro proceso, y luego la inicializa. Si la impresora debe gestionarse por interrupciones, se asigna una memoria intermedia llamando a `kmalloc`, y su dirección se guarda en el campo `lp_buffer` del descriptor de impresora, y luego `lp_open` llama a `request_irq` para especificar que la función `lp_interrupt` debe llamarse al recibir una interrupción relacionada con el puerto paralelo en cuestión. Finalmente, la impresora se marca como ocupada.

La función `lp_release` implementa la operación sobre archivo `release`. Si la impresora estaba gestionada por interrupciones, se llama a `free_irq` para suprimir la llamada a `lp_interrupt` en la recepción de una interrupción, y luego la memo-

ria intermedia apuntada por el campo `lp_buffer` del descriptor de impresora se libera llamando a `kfree_s`. Finalmente, la impresora se marca como disponible.

La función `lp_ioctl` implementa la operación sobre archivo `ioctl`. Ésta permite consultar y modificar los parámetros relacionados con la impresora.

La variable `lp_fops` contiene las direcciones de estas funciones.

4.9.6 Funciones de inicialización

La función `lp_probe` comprueba la presencia de un puerto paralelo específico. Si el puerto existe, inicializa su descriptor.

La función `lp_init` se llama al inicializarse el sistema, o en la carga del gestor en forma de módulo. Primero registra el gestor llamando a `register_chrdev` a la que transmite la variable `lp_fops`, para indicarle las operaciones asociadas al archivo. Luego efectúa un bucle de reconocimiento de los puertos posibles llamando a `lp_probe`.

Gestión de la memoria

Primitivas detalladas

brk, calloc, free, malloc, mlock, mlockall, mmap, mprotect, mremap, msync, munlock, munlockall, munmap, realloc, sbrk

1 Conceptos básicos

1.1 Espacio de direccionamiento de un proceso

A todo proceso se le asocia un espacio de direccionamiento que representa las zonas de memoria asignadas al proceso. Este espacio de direccionamiento incluye:

- el código del proceso;
- los datos del proceso, que se descompone en dos segmentos, por una parte *data*, que contiene las variables inicializadas, y por otra parte *bss* que contiene las variables no inicializadas;
- el código y los datos de las bibliotecas compartidas utilizadas por el proceso;
- la pila utilizada por el proceso.

En la arquitectura x86, Linux asigna tres gigabytes a este espacio de direccionamiento. El gigabyte restante se reserva para la memoria utilizada por el núcleo (el propio código de Linux así como los datos que manipula).

Los tres gigabytes disponibles se descomponen en regiones de memoria utilizables por el proceso.

El programa *MuestraDirecciones.c* muestra la dirección de diversas variables y funciones.

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

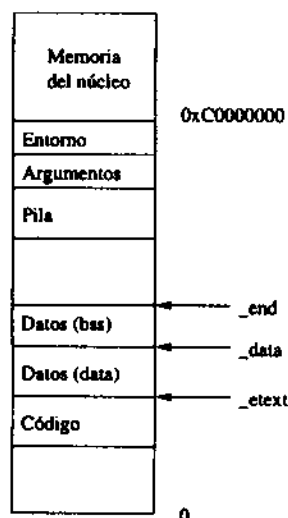
int i;      /* Variable no inicializada (segmento BSS) */
int j = 2;  /* Variable inicializada (segmento DATA) */
extern int _end;
extern int _etext;    /* Fin del segmento de código */
extern int _edata;    /* Fin del segmento de datos */
extern int __bss_start; /* Inicio del segmento BSS */
extern char **entorno; /* Puntero al entorno */

void main (int argc, char *argv[])
{
    int k;
    printf ("Dirección de la función main = %09lx\n", main);
    printf ("Dirección del símbolo _etext = %09lx\n", &_etext);
    printf ("Dirección de la variable j = %09lx\n", &j);
    printf ("Dirección del símbolo _edata = %09lx\n", &_edata);
    printf ("Dirección del símbolo __bss_start = %09lx\n",
            &__bss_start);
    printf ("Dirección de la variable i = %09lx\n", &i);
    printf ("Dirección del símbolo _end = %09lx\n", &_end);
    printf ("Dirección de la variable k = %09lx\n", &k);
    printf ("Dirección del primer argumento = %09lx\n", argv[0]);
    printf ("Dirección de la primera variable = %09lx\n",
            entorno [0]);
    exit (0);
}
```

La visualización de este programa es la siguiente:

```
Dirección de la función main = 008048474
Dirección del símbolo _etext = 008048564
Dirección de la variable j = 00804971c
Dirección del símbolo _edata = 0080497d8
Dirección del símbolo __bss_start = 0080497d8
Dirección de la variable i = 0080497dc
Dirección del símbolo _end = 0080497e0
Dirección de la variable k = 0bffff890
Dirección del primer argumento = 0bffff9a7
Dirección de la primera variable = 0bffff9b9
```

Esta visualización permite deducir la ubicación de las regiones de memoria utilizadas por el proceso, como se esquematiza en la figura 8.1.

FIG. 8.1 – *Espacio de direccionamiento de un proceso*

1.2 Asignación de memoria

Cuando un proceso empieza su ejecución, sus segmentos poseen un tamaño fijo. Sin embargo, existen funciones de asignación y liberación de memoria, que permiten a un proceso manipular variables cuyo número o tamaño no es conocido en el momento de la compilación.

Las asignaciones y liberaciones se efectúan modificando el tamaño del segmento de datos del proceso. Cuando debe asignarse un dato, el segmento de datos se aumenta en el número de bytes necesario y el dato puede almacenarse en el espacio de memoria así asignado.

Cuando un dato situado al final del segmento de datos deja de usarse, su liberación consiste simplemente en reducir el tamaño del segmento.

2 Llamadas al sistema de base

2.1 Cambio del tamaño del segmento de datos

Un proceso puede modificar el tamaño de su segmento de datos. Para ello, Linux proporciona la llamada al sistema *brk*:

```
#include <unistd.h>

int brk (void *end_data_segment);
```

El parámetro `end_data_segment` especifica la dirección de fin del segmento de datos. Debe ser superior a la dirección de fin del segmento de código e inferior en 16 kilobytes a la dirección de fin del segmento de pila. En caso de éxito, `brk` devuelve el valor 0. En caso de fallo, se devuelve el valor -1 y la variable `errno` toma el valor `ENOMEM`.

Una función de biblioteca permite también al proceso actual modificar el tamaño de su segmento de datos:

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

void *sbrk (ptrdiff_t increment);
```

El parámetro `increment` especifica el número de bytes a añadir al segmento de datos, o a sustraer si `increment` es negativo. La función `sbrk` devuelve la nueva dirección de fin de segmento de datos, o el valor -1 en caso de fallo. En este último caso, la variable `errno` toma el valor `ENOMEM`.

2.2 Funciones de asignación y liberación de memoria

Aunque es posible gestionar dinámicamente la memoria por medio de las funciones `brk` y `sbrk`, es relativamente pesado proceder de este modo. Aunque la asignación de memoria se facilita, porque basta con aumentar el tamaño del segmento de datos, la liberación es más ardua, porque es necesario tener en cuenta las zonas de memoria utilizadas para disminuir el tamaño del segmento de datos cuando es necesario. Por esta razón, se utilizan generalmente funciones de asignación y liberación proporcionadas por la biblioteca estándar.

Estas funciones utilizan de manera interna `brk` y `sbrk` para asignar y liberar zonas de memoria, y gestionan la estructuración de los bloques de memoria.

Sus prototipos son los siguientes:

```
#include <stdlib.h>

void *malloc (size_t size);
void *calloc (size_t nmemb, size_t size);
void *realloc (void *ptr, size_t size);
void free (void *ptr);
```

La función *malloc* permite asignar una nueva zona de memoria. El parámetro *size* especifica el tamaño en bytes del bloque a asignar. *malloc* devuelve la dirección de la zona asignada, o bien el valor NULL en caso de fallo. El contenido del bloque asignado es indeterminado.

La función *calloc* permite asignar también una zona de memoria, pero está prevista para asignar una matriz. El parámetro *nmemb* especifica el número de elementos de la matriz, y *size* indica el tamaño de cada uno de los elementos de la matriz, expresado en bytes. El tamaño del bloque a asignar es pues $nmemb * size$. Al igual que *malloc*, *calloc* devuelve la dirección de la zona asignada, o bien el valor NULL en caso de error. El contenido del bloque asignado se inicializa a cero.

La función *realloc* modifica el tamaño de una zona de memoria. El parámetro *ptr* especifica la dirección de un bloque de memoria, asignado mediante una llamada a *malloc* o *calloc*, y *size* indica su nuevo tamaño. El contenido de la zona de memoria se copia en el nuevo bloque asignado. La dirección de la nueva zona asignada se devuelve por *realloc*, y puede ser diferente de la dirección especificada por *ptr*. En caso de error, se devuelve el valor NULL, y la zona apuntada por *ptr* no se libera.

Finalmente, la función *free* libera una zona de memoria. El parámetro *ptr* especifica la dirección de un bloque de memoria asignado por una llamada a *malloc*, *calloc* o *realloc*. Tras la ejecución de *free*, la zona de memoria se libera y *ptr* ya no contiene una dirección válida.

3 Conceptos avanzados

3.1 Regiones de memoria

Como se expone en la sección 1.1, el espacio de direccionamiento de un proceso se compone de varias regiones de memoria. Cada región de memoria se caracteriza por varios atributos:

- sus direcciones de inicio y de fin;
- los derechos de acceso que tiene asociados;
- el objeto asociado (por ejemplo, un archivo ejecutable que contiene el código ejecutado por el proceso).

Las regiones de memoria contenidas en el espacio de direccionamiento de un proceso pueden determinarse mostrando el contenido del archivo *maps*, situado en el directorio de cada proceso en el sistema de archivos */proc*, por ejemplo:

```
bbj>>cat /proc/self/maps
08048000-0804a000 r-xp 00000000 03:02 7914
0804a000-0804b000 rw-p 00001000 03:02 7914
0804b000-08053000 rwxp 00000000 00:00 0
40000000-40005000 rwxp 00000000 03:02 18336
40005000-40006000 rw-p 00004000 03:02 18336
40006000-40007000 rw-p 00000000 00:00 0
40007000-40009000 r--p 00000000 03:02 18255
40009000-40082000 r-xp 00000000 03:02 18060
40082000-40087000 rw-p 00078000 03:02 18060
40087000-400b9000 rw-p 00000000 00:00 0
bffff000-c0000000 rwxp fffff000 00:00 0
```

Los dos primeros campos de cada línea representan las direcciones de principio y de fin de la región. El campo siguiente expresa los derechos de acceso asociados (el carácter «p» indica que la región puede ser compartida con otros procesos). Los campos siguientes indican las informaciones referidas al objeto asociado a la región de memoria: el desplazamiento del inicio de la región en el objeto, el número de dispositivo que contiene el objeto y el número de i-nodo del objeto.

En el ejemplo presentado, las tres primeras regiones corresponden al programa ejecutado (segmento de código, segmento de datos inicializados y segmento de datos no inicializados). Las tres regiones siguientes corresponden al cargador de programas *ld-linux.so.1* (segmento de código, segmento de datos inicializados y segmento de datos no inicializados). La región siguiente corresponde al archivo */usr/share/locale/fr_FR/LC_CTYPE* que se carga en memoria automáticamente al arrancar el programa. Las tres regiones siguientes corresponden a la biblioteca C compartida, */lib/libc.so.5.3.12* (segmento de código, segmento de datos inicializados y segmento de datos no inicializados). Finalmente, la última región corresponde al segmento de pila utilizado por el proceso.

3.2 Protección de la memoria

Según el tipo de informaciones contenidas en memoria, se asocia una protección diferente a cada región de memoria perteneciente al espacio de direccionamiento de un proceso.

Los derechos de acceso a la memoria son gestionados directamente por el procesador: a cada página de memoria se le asocia una protección, y el procesador verifica la validez de cada acceso efectuado por el proceso actual.

Linux permite a un proceso modificar las protecciones de memoria asociadas a algunas de las regiones comprendidas en su espacio de direccionamiento. Ofrece varios tipos de derechos de acceso:

- **PROT_NONE**: la región de memoria se marca como inaccesible;
- **PROT_READ**: el proceso actual puede leer los datos contenidos en la región;
- **PROT_WRITE**: el proceso actual puede modificar los datos contenidos en la región;
- **PROT_EXEC**: el proceso actual puede ejecutar el código contenido en la región.

Cuando un proceso modifica los derechos de acceso asociados a una región de memoria, el núcleo modifica las informaciones de protección asociadas a las páginas de memoria correspondientes. Los controles de acceso son efectuados por el procesador, que desencadena una interrupción en caso de acceso incorrecto. Esta interrupción es gestionada por Linux, que envía la señal **SIGSEGV** al proceso culpable.

Hay que destacar que, en la arquitectura x86, no todos los controles de acceso son implementados por el procesador. En los procesadores x86, las protecciones **PROT_WRITE** y **PROT_EXEC** implican **PROT_READ**. Es por tanto imposible autorizar la modificación de datos o la ejecución de código, impidiendo su acceso en lectura.

3.3 Bloqueo de zonas de memoria

Linux utiliza la carga de páginas bajo petición. Esto significa que las páginas que forman parte del espacio de direccionamiento de un proceso no se cargan todas en memoria al arrancar el proceso. Cuando el proceso efectúa un acceso a memoria en una página no cargada, el procesador desencadena una interrupción. Esta interrupción es gestionada por Linux, que puede reaccionar de distintas maneras:

- enviar la señal **SIGSEGV** al proceso actual, si la página de memoria referenciada no forma parte del espacio de direccionamiento del proceso, es decir, si el proceso ha utilizado una dirección de memoria inválida;
- asignar una página vacía si, por ejemplo, la página de memoria correspondiente forma parte del segmento de pila;
- cargar el contenido de la página, por ejemplo, desde la memoria secundaria, desde el archivo ejecutable que contiene el programa en curso de ejecución.

De manera simétrica, Linux puede elegir liberar una página de memoria. Si el núcleo necesita memoria central, selecciona una o más páginas, las escribe eventualmente en memoria secundaria (si han sido modificadas) y las libera.

Un proceso puede verse suspendido por el núcleo cuando este último debe cargar páginas o escribirlas en memoria secundaria. Este comportamiento es incompatible con los procesos en tiempo real, que no deben ser interrumpidos ni suspendidos. Asimismo, Linux permite a un proceso privilegiado bloquear ciertas páginas en memoria.

Diversas llamadas al sistema permiten a un proceso especificar que sus páginas no deben ser descartadas de la memoria. Una vez las páginas especificadas están presentes en memoria, el núcleo no las tiene en cuenta cuando necesita liberar memoria. De este modo, el proceso no puede ser suspendido en espera de la carga de una página que habría sido descartada de la memoria.

3.4 Proyección de archivos en memoria

El espacio de direccionamiento de un proceso contiene de modo predeterminado sus segmentos de código, de datos y de pila, así como los segmentos de código y de datos de las bibliotecas compartidas que utiliza.

Un proceso puede crear nuevas regiones de memoria en su espacio de direccionamiento para acceder al contenido de archivos. Esta operación se llama proyección de archivo en memoria (*file mapping*). Cuando un proceso proyecta el contenido de un archivo en memoria, se crea una nueva región de memoria en su espacio de direccionamiento y el contenido del archivo se hace accesible en esta región. También es posible acceder directamente al contenido del archivo utilizando lecturas y escrituras directas en memoria, sin utilizar las primitivas *read* y *write*.

Este método es particularmente interesante cuando un proceso manipula un archivo constituido por una serie de elementos del mismo tipo. Al utilizar las primitivas de entrada/salida sobre archivos, el proceso debería utilizar la llamada al sistema *lseek* para posicionarse sobre el elemento que desea manipular, y luego leer o escribir el dato por *read* o *write*. Si el archivo se proyecta en memoria, el proceso puede utilizar la dirección de la zona correspondiente como dirección de base, y utilizar un índice para acceder al elemento deseado, de la misma manera que para una variable de tipo matriz.

Están disponibles varios tipos de proyecciones:

- las proyecciones compartidas: varios procesos proyectan el contenido de un archivo en sus espacios de direccionamiento, y toda modificación efectuada por un proceso es inmediatamente visible para los otros;
- las proyecciones privadas: las modificaciones que efectúa el proceso sobre el contenido del archivo son privadas, y no son visibles por los otros procesos que han proyectado el mismo archivo en su espacio de direccionamiento;

- las proyecciones anónimas: el proceso, en realidad, ha creado una zona de memoria que no corresponde al contenido de un archivo. En los accesos en lectura a la zona de memoria, Linux asigna páginas inicializadas a cero. Las páginas modificadas se guardan en el *swap*.

3.5 Dispositivos de *swap*

Cuando el núcleo necesita memoria, puede eliminar páginas en memoria. Si el contenido de estas páginas ha sido modificado, es necesario guardarlas en disco: una página correspondiente a un archivo proyectado en memoria se reescribe en el archivo, y una página correspondiente a los datos se guarda en un dispositivo de *swap*.

Un dispositivo de *swap* puede ser un dispositivo en modo bloque, por ejemplo una partición en disco, o un disco normal. Previamente debe inicializarse mediante el mandato *mkswap*.

Linux es capaz de utilizar varios dispositivos de *swap*. Cuando una página debe guardarse, los dispositivos de *swap* activo se exploran para encontrar un lugar donde escribir la página. La activación de un dispositivo de *swap* se efectúa llamando a la función de sistema *swapon*.

Contrariamente a la mayor parte de sistemas, Linux no se limita a la activación de dispositivos de *swap*. Un dispositivo activo puede desactivarse, sin tener que reiniciar el sistema. Al desactivarlo, todas las páginas guardadas en el dispositivo se vuelven a cargar en memoria. La llamada al sistema *swapoff* efectúa esta desactivación.

4 Llamadas al sistema complementarias

4.1 Protección de páginas de memoria

Un proceso puede modificar los derechos de acceso asociados a una parte de la memoria que se le asigna. Para ello, Linux proporciona la primitiva *mprotect*, cuya sintaxis es la siguiente:

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

```
int mprotect (caddr_t addr, size_t len, int prot);
```


La llamada al sistema modifica las protecciones de acceso asociadas a la zona de memoria especificada. El parámetro *addr* representa la dirección de la zona de memoria a modificar, y *len* indica el tamaño de la zona, expresado en bytes. El parámetro *prot* especifica los derechos de acceso a posicionar. *mprotect* devuelve el valor 0 en caso de éxito, o el valor -1 en caso de error. En este último caso, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EACCESS	La protección especificada por <i>prot</i> no es aplicable
EFAULT	La dirección especificada por <i>addr</i> no forma parte del espacio de direccionamiento del proceso que llama
EINVAL	<i>addr</i> , <i>len</i> o <i>prot</i> contiene un valor no válido
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos

Diferentes constantes, definidas en el archivo `<sys/mman.h>`, representan las diferentes protecciones posibles:

<i>constante</i>	<i>significado</i>
PROT_NONE	La zona se marca como inaccesible
PROT_READ	La zona se marca como accesible en lectura
PROT_WRITE	La zona se marca como accesible en escritura
PROT_EXEC	La zona se marca como accesible en ejecución

Tras esta llamada al sistema, si el proceso actual intenta acceder a la zona de manera incompatible con las protecciones posicionadas, por ejemplo si intenta escribir datos en una zona accesible en lectura exclusiva, se le envía la señal SIGSEGV.

4.2 Bloqueo de páginas de memoria

Un proceso privilegiado puede bloquear páginas en memoria. Linux ofrece varias llamadas al sistema para ello:

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

int mlock (const void *addr, size_t len);
int munlock (const void *addr, size_t len);
int mlockall (int flags);
int munlockall (void);
```

La primitiva *mlock* permite bloquear una zona en memoria. El parámetro *addr* especifica la dirección de inicio de la zona, y *len* indica su tamaño en bytes. Tras esta llamada al sistema, la zona de memoria queda residente en memoria y no se transferirá a la memoria secundaria.

La llamada al sistema *munlock* permite desbloquear una zona en memoria. El parámetro *addr* especifica la dirección de inicio de la zona, y *len* indica su tamaño en bytes.

La llamada al sistema *mlockall* bloquea en memoria todas las páginas comprendidas en el espacio de direccionamiento del proceso actual, es decir, las páginas de los segmentos de código, de datos y de pila, así como las bibliotecas compartidas, los segmentos de memoria compartidos, y los archivos proyectados en memoria. El parámetro *flag* especifica las modalidades del bloqueo, y se expresa según las constantes siguientes, definidas en el archivo `<sys/mman.h>`:

constante	significado
MCL_CURRENT	Todas las páginas cargadas en memoria en el espacio de direccionamiento del proceso actual están bloqueadas
MCL_FUTURE	Todas las páginas que se cargarán en memoria en el futuro en el espacio de direccionamiento del proceso actual se bloquearán

La primitiva *munlockall* desbloquea todo el espacio de direccionamiento del proceso actual, y anula el efecto de las llamadas al sistema *mlock* y *mlockall* anteriores.

Todas estas primitivas devuelven el valor 0 en caso de éxito. En caso de error, se devuelve el valor -1 y la variable *errno* puede tomar los valores siguientes:

error	significado
EINVAL	<i>flags</i> contiene un valor no válido
ENOMEM	La memoria disponible no permite el bloqueo
EPERM	El proceso que llama no posee los privilegios necesarios para bloquear páginas de memoria

4.3 Proyección en memoria

Linux proporciona varias llamadas al sistema que permiten proyectar el contenido de archivos en memoria:

```
#include <unistd.h>
#include <sys/mman.h>
void *mmap (void *start, size_t len, int prot, int flags, int
            fd, off_t offset);
int munmap (void *start, size_t len);
void *mremap (void *old_start, size_t old_len, size_t new_len,
              unsigned long flags);
```

La llamada al sistema *mmap* proyecta el contenido de un archivo en memoria, en el espacio de direccionamiento del proceso actual. El parámetro *start* especifica la dirección de la zona de memoria donde el contenido del archivo debe ser accesible. Esta

dirección es únicamente una indicación dada al sistema, que puede decidir utilizar otra de inicio. El parámetro *len* indica el número de bytes a proyectar en memoria. Las protecciones en memoria a aplicar se definen por el parámetro *prot*, mediante las constantes detalladas en la sección 4.1. El parámetro *flag* especifica las modalidades de la protección. Finalmente, *fd* indica el descriptor del archivo, cuyo contenido debe proyectarse en memoria, y *offset* especifica a partir de qué byte el contenido del archivo debe ser accesible. La dirección a partir de la cual el contenido del archivo es accesible se devuelve por *mmap*. En caso de error, *mmap* devuelve el valor -1.

Pueden utilizarse varias constantes para el parámetro *flag*, definidas en el archivo `<sys/mman.h>`:

constante	significado
MAP_FIXED	La dirección de inicio debe imperativamente corresponder al parámetro <i>addr</i> . Si esta dirección no se puede usar, <i>mmap</i> devuelve un error
MAP_SHARED	La proyección en memoria se comparte con todos los otros procesos que hayan proyectado el archivo en memoria. Toda modificación efectuada por un proceso es inmediatamente visible por los otros
MAP_PRIVATE	La proyección en memoria sólo afecta al proceso actual. Toda modificación efectuada por el proceso actual no será visible por los otros procesos que han proyectado el archivo en memoria
MAP_ANONYMOUS	La proyección en memoria no afecta a ningún archivo. La primitiva <i>mmap</i> se llama para crear una nueva región de memoria cuyo contenido se inicializará a cero
MAP_DENYWRITE	Todo intento de acceso en escritura al archivo por un proceso devolverá el error ETXTBSY
MAP_LOCKED	La región creada debe bloquearse en memoria

En caso de error, la variable *errno* puede tomar los valores siguientes:

error	significado
EACCES	El tipo de proyección o las proyecciones de acceso son incompatibles con el modo de apertura del archivo
EAGAIN	El archivo está bloqueado o una cantidad demasiado importante de páginas están bloqueadas en memoria
EBADF	El descriptor de entradas/salidas especificado no es válido
EINVAL	<i>start</i> , <i>len</i> u <i>offset</i> contiene un valor no válido (por ejemplo una dirección que no está alineada a una frontera de página)
ENOMEM	No existe bastante memoria disponible
ETXTBSY	La opción MAP_DENYWRITE se ha especificado pero el archivo está abierto en escritura

La primitiva *munmap* suprime la proyección en memoria de un archivo. El parámetro *start* especifica la dirección de la zona de memoria correspondiente y *len* indica su tamaño, expresado en bytes. En caso de éxito, se devuelve el valor 0. En caso de error, *munmap* devuelve el valor -1, y la variable *errno* toma el valor **EINVAL**.

La llamada al sistema *mremap* modifica el tamaño de una zona de memoria. El parámetro *old_start* especifica la dirección de inicio de la zona, cuyo tamaño en bytes se indica en *old_len*. El nuevo tamaño de la zona se transmite en el parámetro *new_len*. El parámetro *flags* especifica las modalidades de la modificación. Linux 2.0 sólo proporciona una opción, *MREMAP_MAYMOVE*, que indica que el núcleo está autorizado para modificar la dirección de inicio de la zona. La primitiva *mremap* devuelve la dirección de la zona de memoria, que puede ser diferente del valor transmitido en *old_start*, o bien se devuelve el valor *NULL* en caso de error. En este último caso, la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EAGAIN	La región de memoria está bloqueada y no puede desplazarse
EFAULT	La región de memoria especificada por <i>old_start</i> y <i>old_len</i> no forma parte del espacio de direccionamiento del proceso que llama
EINVAL	<i>old_start</i> , <i>old_len</i> o <i>new_len</i> contiene un valor no válido (por ejemplo una dirección que no está alineada en una frontera de página)
ENOMEM	El tamaño de la región de memoria no puede aumentarse, y <i>MREMAP_MAYMOVE</i> no se ha especificado

El programa *EjemploMmap.c* utiliza la primitiva *mmap* para acceder al contenido de un archivo. Abre un archivo, utiliza *mmap* para proyectar su contenido en memoria, y efectúa un bucle de visualización de cada byte contenido en el archivo.

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

void main (int argc, char *argv[])
{
    int          fd;
    int          i;
    struct stat   st;
    char         *addr;

    /* Control de argumentos */
    if (argc != 2) {
        fprintf (stderr, "Uso: %s nombre_de_archivo\n", argv[0]);
        exit (1);
    }
    /* Obtención del tamaño del archivo */
    if (stat (argv[1], &st) == -1) {
        perror ("stat");
        exit (2);
    }
}
```

```
/* Apertura del archivo */
fd = open (argv[1], O_RDONLY);
if (fd == -1) {
    perror ("open");
    exit (3);
}
/* Proyección del archivo en memoria */
addr = (char*) mmap (NULL, st.st_size, PROT_READ,
                     MAP_SHARED, fd, (off_t) 0);
if (addr == NULL) {
    perror ("mmap");
    (void) close (fd);
    exit (4);
}
/* Cierre del archivo */
close (fd);
/* Bucle de visualización del contenido del archivo */
for (i = 0; i < st.st_size; i++)
    putchar (addr[i]);
/* Liberación del archivo */
if (munmap (addr, st.st_size) == -1) {
    perror ("munmap");
    (void) close (fd);
    exit (5);
}
exit (0);
}
```

4.4 Sincronización de páginas en memoria

La llamada al sistema *msync* provoca la reescritura en disco de datos contenidos en una región de memoria. Su prototipo es el siguiente:

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

int msync (const void *start, size_t len, int flags);
```

La llamada al sistema *msync* provoca la reescritura en disco de las modificaciones efectuadas en el contenido de un archivo proyectado en memoria. El parámetro *start* especifica la dirección de inicio de la zona a reescribir, cuyo tamaño en bytes se indica en el parámetro *len*. Las modalidades de actualización se especifican en el parámetro *flags*. La primitiva *msync* devuelve el valor 0 en caso de éxito, o el valor -1 en caso de error.

Diversas constantes permiten modificar el comportamiento de *msync*:

constante	significado
MS_ASYNC	La reescritura se efectúa de manera asíncrona: cuando la llamada al sistema termina, las reescrituras han sido lanzadas, pero no está garantizado que los datos hayan sido ya escritos en disco
MS_SYNC	La reescritura se efectúa de manera síncrona: cuando termina la llamada al sistema, las modificaciones se han escrito físicamente en el disco
MS_INVALIDATE	Las otras proyecciones en memoria del mismo archivo no son válidas, de manera que se provoca la relectura de los datos cuando accede un proceso. De este modo, las modificaciones efectuadas son visibles por los otros procesos

En caso de error, la variable *errno* puede tomar los valores siguientes:

error	significado
EFAULT	La región de memoria especificada por <i>start</i> y <i>len</i> no forma parte del espacio de direccionamiento del proceso que llama
EINVAL	<i>start</i> no contiene una dirección alineada en una frontera de página, o <i>flags</i> contiene un valor no válido

4.5 Gestión de los dispositivos de *swap*

La llamada al sistema *swapon* permite activar un dispositivo de *swap*. Su prototipo es el siguiente:

```
#include <unistd.h>
#include <linux/swap.h>
```

```
int swapon (const char *pathname, int swapflags);
```

La primitiva *swapon* activa el dispositivo cuyo nombre se especifica en el parámetro *pathname*. Éste puede ser un dispositivo en modo bloque, o un archivo regular. En los dos casos, debe haber sido inicializado por el mandato *mkswap*. El parámetro *swapflags* permite indicar la prioridad del dispositivo. Cuando debe guardarse una página, Linux explora la lista de dispositivos de *swap* y utiliza el que tiene mayor prioridad y que posee páginas sin asignar. El valor de *swapflags* se expresa añadiendo la prioridad deseada (un entero entre 1 y 32767) a la constante *SWAP_FLAG_PREFER*.

En caso de éxito se devuelve el valor 0. En caso de error, *swapon* devuelve el valor -1 y la variable *errno* puede tomar los valores siguientes:

error	significado
EFAULT	<i>pathname</i> contiene una dirección no válida
EINVAL	<i>pathname</i> no se refiere a un dispositivo en modo bloque ni a un archivo regular
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos

ENAMETOOLONG	pathname especifica un nombre de archivo demasiado largo
ENOENT	pathname se refiere a un nombre de archivo inexistente
ENOMEM	El núcleo no ha podido asignar memoria para sus descriptores internos
ENOTDIR	Uno de los componentes de pathname, utilizado como nombre de directorio, no lo es
EPERM	El proceso que llama no posee los privilegios necesarios, o el número máximo de dispositivos activos se ha alcanzado ya

La desactivación de un dispositivo de *swap* se efectúa llamando a la primitiva *swapoff*. Su prototipo es el siguiente:

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

```
int swapoff (const char *pathname);
```

El parámetro *pathname* especifica el nombre del dispositivo de *swap* a desactivar. En caso de éxito, *swapoff* devuelve el valor 0; si no devuelve el valor -1 y la variable *errno* puede tomar los valores siguientes:

<i>error</i>	<i>significado</i>
EFAULT	pathname contiene una dirección no válida
EINVAL	pathname no se refiere a un dispositivo de <i>swap</i> activo
ELOOP	Se ha encontrado un ciclo de enlaces simbólicos
ENAMETOOLONG	pathname especifica un nombre de archivo demasiado largo
ENOENT	pathname se refiere a un nombre de archivo inexistente
ENOMEM	La memoria disponible es demasiado restringida para recargar todas las páginas contenidas en el dispositivo de <i>swap</i>
ENOTDIR	Uno de los componentes de pathname, utilizado como nombre de directorio, no lo es
EPERM	El proceso que llama no posee los privilegios necesarios

5 Presentación general de la implementación

5.1 Gestión de las tablas de página

5.1.1 Segmentación

En ciertas arquitecturas, el acceso a la memoria se efectúa utilizando segmentos. Una dirección se compone de un identificador de segmento, y de un desplazamiento en el segmento. El procesador combina la dirección del segmento y el desplazamiento para obtener una dirección virtual.

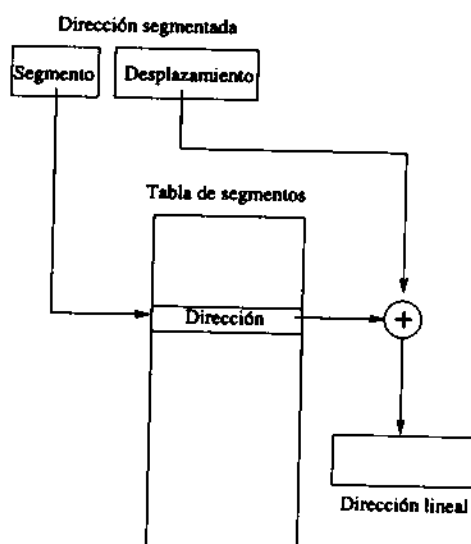


FIG. 8.2 – Segmentación de memoria

En la arquitectura x86 se definen varios registros de segmentos. Contienen punteros a descriptores de segmento que definen la protección asociada al segmento, así como la dirección de inicio del segmento en memoria.

La figura 8.2 ilustra la conversión de una dirección compuesta de un identificador de segmento y de un desplazamiento en el segmento.

En la arquitectura x86, Linux utiliza este mecanismo de segmentación para separar las zonas de memoria asignadas al núcleo y a los procesos. Dos segmentos se refieren a los tres primeros gigabytes del espacio de direccionamiento de los procesos y su contenido puede leerse y modificarse en modo usuario y en modo núcleo. Dos segmentos se refieren al cuarto gigabyte del espacio de direccionamiento y su contenido puede leerse y modificarse en modo núcleo únicamente.

De este modo, el código y los datos del núcleo se protegen de los accesos erróneos o mal intencionados por parte de procesos en modo usuario.

En modo usuario, los registros de segmento *cs* y *ds* apuntan a los dos segmentos de usuario. En modo núcleo, *cs* y *ds* apuntan a los dos segmentos del núcleo. La modificación del valor de los registros de segmentos se efectúa en el cambio de modo de ejecución, cuando un proceso pasa al modo núcleo para ejecutar una llamada al sistema, por ejemplo. Además, este paso al modo núcleo provoca la modificación del registro de segmento *fs*. Este registro apunta al segmento de datos del proceso que llama, a fin de permitir al núcleo leer y escribir en su espacio de direccionamiento, mediante funciones especializadas.

5.1.2 Paginación

Linux utiliza los mecanismos de memoria virtual proporcionados por el procesador sobre el que se ejecuta. Las direcciones manipuladas por el núcleo y los procesos son direcciones virtuales y el procesador efectúa una conversión para transformar una dirección virtual en dirección física en memoria central.

El mecanismo de conversión es el siguiente: una dirección de memoria se descompone en dos partes, un número de página y un desplazamiento en la página. El número de página se utiliza como índice en una tabla, llamada tabla de páginas, lo que proporciona una dirección física de página en memoria central. A esta dirección se le añade el desplazamiento para obtener la dirección física de la palabra de memoria en concreto.

La figura 8.3 representa esta conversión.

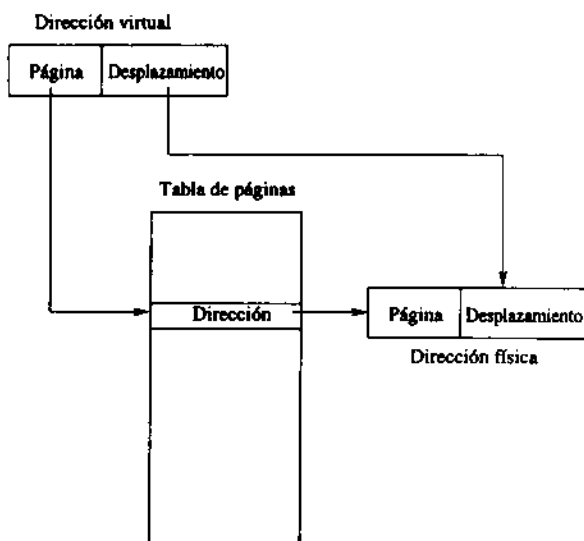


FIG. 8.3 – Conversión de dirección virtual en dirección física

Debido al tamaño del espacio de memoria direccionable por los procesadores, la tabla de páginas raramente se implementa en forma de una sola tabla contigua en memoria. Como la tabla de páginas debe ser residente en memoria, necesitaría un exceso de memoria únicamente para esta tabla. Por ejemplo, los procesadores de la arquitectura x86 pueden direccionar cuatro gigabytes, el tamaño de las páginas de memoria es de cuatro kilobytes, y cada entrada de la tabla ocupa cuatro bytes; con tales procesadores, una tabla de páginas completa utilizaría 1048576 entradas, ocupando cuatro megabytes de memoria.

Por esta razón, la tabla de páginas a menudo se descompone en varios niveles, con un mínimo de 2:

- un catálogo de tabla de páginas contiene las direcciones de las páginas que contienen partes de dicha tabla;
- las partes utilizadas de la tabla de páginas se cargan en memoria.

La figura 8.4 representa la conversión de dirección en el caso de la arquitectura x86, que utiliza una tabla de páginas de dos niveles.

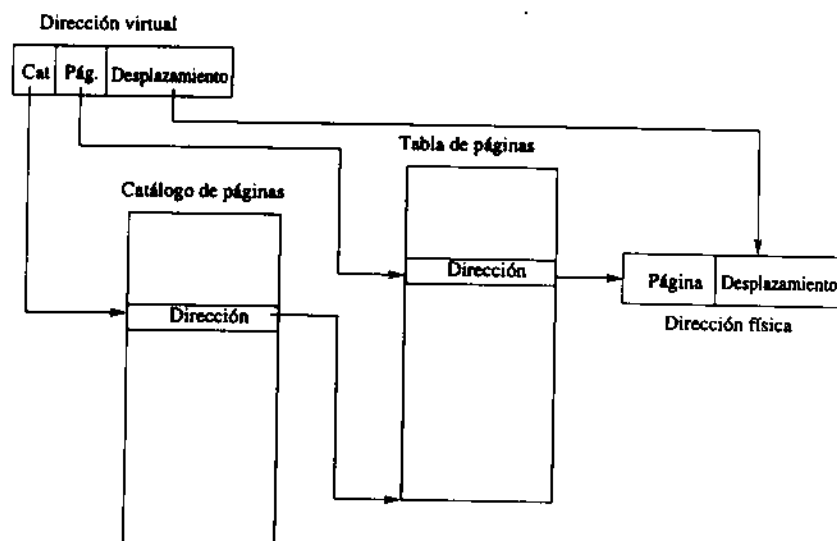


FIG. 8.4 – Tabla de páginas de dos niveles

El interés de esta tabla de páginas por niveles se basa en que la tabla de páginas no necesita cargarse completamente en memoria. Si un proceso utiliza 6 megabytes de memoria en un procesador x86, sólo se utilizan tres páginas para la tabla de páginas:

- la página que contiene el catálogo;
- la página que contiene la parte de la tabla de páginas correspondiente a los primeros 4 megabytes de memoria;
- la página que contiene la parte de la tabla de páginas correspondiente a los 4 megabytes de memoria siguientes (de la que sólo se utiliza la mitad de entradas).

5.1.3 Tablas de páginas gestionadas por Linux

Linux gestiona la memoria central y las tablas de páginas utilizadas para convertir las direcciones virtuales en direcciones físicas. Implementa una gestión de la memoria que es ampliamente independiente del procesador sobre el que se ejecuta.

En realidad, la gestión de la memoria implementada por Linux considera que dispone de una tabla de páginas a tres niveles:

- la tabla global (*page global directory*) cuyas entradas contienen las direcciones de páginas que contienen tablas intermedias;
- las tablas intermedias (*page middle directory*) cuyas entradas contienen las direcciones de páginas que contienen tablas de páginas;
- las tablas de páginas (*page table*) cuyas entradas contienen las direcciones de páginas de memoria que contienen el código o los datos utilizados por el núcleo o los procesos de usuario.

Evidentemente, este modelo no siempre corresponde al procesador sobre el cual Linux se ejecuta (los procesadores x86, por ejemplo, utilizan una tabla de páginas que sólo posee dos niveles). El núcleo efectúa una correspondencia entre el modelo implementado por el procesador y el modelo de Linux. En los procesadores x86, por ejemplo, el núcleo considera que la tabla intermedia sólo contiene una entrada.

5.1.4 Compartir páginas

Cuando varios procesos acceden a los mismos datos, Linux intenta compartir al máximo las páginas en memoria. Por ejemplo, si varios procesos ejecutan el mismo programa, el código del programa se carga una sola vez en memoria, y las entradas de las tablas de páginas de los diferentes procesos apuntan a las mismas páginas de memoria.

Además, Linux implementa una técnica de copia en escritura, que permite minimizar el número de páginas de memoria utilizadas. Cuando se crea un proceso, llamando a la primitiva *fork*, hereda el espacio de direccionamiento de su padre. Evidentemente, el segmento de código no se duplica, pero Linux tampoco duplica el segmento de datos. Al duplicar el proceso, todas las páginas del segmento de datos se marcan como accesibles en lectura exclusiva en las tablas de páginas de los dos procesos. Cuando uno de los dos procesos intenta modificar un dato, el procesador provoca una interrupción de memoria, que gestiona el núcleo. Al tratar esta interrupción, Linux duplica la página afectada y la inserta en la tabla de páginas del proceso que ha causado la interrupción.

De este modo, las páginas de memoria sólo se duplican físicamente cuando su contenido se modifica. Toda página de memoria que contiene únicamente datos no modificados puede permanecer compartida entre los dos procesos, minimizando así el espacio de memoria necesario.

5.1.5 Tipos

El archivo de cabecera `<asm/page.h>` define los tipos utilizados para representar las entradas de tablas de páginas:

- `pte_t`: tipo de una entrada de la tabla de páginas;
- `pmd_t`: tipo de una entrada de la tabla intermedia;
- `pgd_t`: tipo de una entrada de la tabla global;
- `pgprot_t`: tipo utilizado para expresar las protecciones de memoria.

Aunque estos tipos pueden expresarse en forma de enteros de cuatro bytes en la arquitectura x86, se definen en forma de estructuras a fin de permitir al compilador la verificación de los tipos de datos manipulados.

Varias macroinstrucciones permiten acceder a estos tipos:

macroinstrucción	significado
<code>pte_val</code>	Devuelve el contenido de una entrada de la tabla de páginas
<code>pmd_val</code>	Devuelve el contenido de una entrada de la tabla intermedia
<code>pgd_val</code>	Devuelve el contenido de una entrada de la tabla global
<code>pgprot_val</code>	Devuelve la protección de memoria
<code>__pte</code>	Convierte un entero en entrada de la tabla de páginas
<code>__pmd</code>	Convierte un entero en entrada de la tabla intermedia
<code>__pgd</code>	Convierte un entero en entrada de la tabla global
<code>__pgprot</code>	Convierte un entero en protección de memoria

5.2 Gestión de las páginas de memoria

5.2.1 Descriptores de página

Linux actualiza el estado de cada página que forma parte de la memoria central. Para ello utiliza una tabla de descriptores, apuntada por la variable `mem_map`, que se declara en el archivo fuente `mm/memory.c`. Cada uno de los descriptores corresponde a una página de memoria.

La estructura `page`, declarada en el archivo de cabecera `<linux/mm.h>`, define el formato de este descriptor. Contiene los campos siguientes:

<i>tipo</i>	<i>campo</i>	<i>descripción</i>
<code>struct page *</code>	<code>next</code>	Puntero a la página libre siguiente
<code>struct page *</code>	<code>prev</code>	Puntero a la página libre anterior
<code>struct inode *</code>	<code>inode</code>	Descriptor del i-nodo correspondiente al contenido de la página
<code>unsigned long</code>	<code>offset</code>	Desplazamiento de la página en el i-nodo
<code>struct page *</code>	<code>next_hash</code>	Dirección del descriptor de página siguiente en una lista de <i>hash</i>
<code>atomic_t</code>	<code>count</code>	Número de referencias a la página
<code>unsigned</code>	<code>flags</code>	Estado de la página (véase más adelante)
<code>unsigned:16</code>	<code>dirty</code>	Booleano que indica si el contenido de la página se ha modificado
<code>unsigned:8</code>	<code>age</code>	Contador utilizado para seleccionar las páginas a descartar de la memoria
<code>struct wait_queue *</code>	<code>wait</code>	Cola de espera usada para esperar la disponibilidad de la página
<code>struct page *</code>	<code>prev_hash</code>	Dirección del descriptor de página anterior en una lista de <i>hash</i>
<code>struct buffer_head *</code>	<code>buffers</code>	Dirección del primer descriptor de memoria intermedia cuyo contenido se sitúa en la página

Las constantes siguientes, declaradas en el archivo de cabecera `<linux/mm.h>`, definen el estado de una página:

<i>constante</i>	<i>significado</i>
PG_locked	La página está bloqueada en memoria
PG_error	Se ha producido un error en la carga de la página en memoria
PG_referenced	La página ha sido accedida
PG_update	El contenido de la página está actualizado
PG_free_after	La página debe liberarse tras el fin de la entrada/salida en curso
PG_decr_after	El número de referencias a la página debe decrementarse tras el fin de la entrada/salida en curso
PG_DMA	La página es utilizable para una transferencia DMA (en la arquitectura x86, está contenida en los primeros 16 MB de memoria central)
PG_reserved	La página está reservada para un uso futuro, no es posible acceder a ella

5.2.2 El caché de páginas

Linux mantiene en memoria un caché de páginas. Todas las páginas en memoria asociadas a un i-nodo se registran en una lista de hash. Pueden corresponder a código ejecutado por los procesos o al contenido de archivos proyectados en memoria.

Este caché se gestiona dinámicamente. Cuando el contenido de una página asociada a un i-nodo debe cargarse en memoria, se asigna una nueva página, se inserta en el caché,

y su contenido se lee desde el disco. En los accesos ulteriores al contenido de la página, su carga ya no es necesaria porque está ya presente en memoria.

Cuando Linux está falto de memoria, selecciona páginas de memoria para su descarte. Las páginas presentes en el caché que no han sido accedidas desde hace tiempo se descartan por las funciones `page_unuse` (véase la sección 6.8) y `try_to_swap_out` (véase la sección 6.9.3).

Desde la versión 2.0, Linux utiliza también este mecanismo de caché de páginas para las lecturas sobre archivos. Cuando se utiliza la llamada al sistema *read*, la lectura se efectúa cargando en memoria las páginas correspondientes del i-nodo. Esto permite utilizar el caché de páginas y aprovechar páginas ya presentes en memoria.

Sólo las lecturas de datos desde archivos regulares se efectúan por medio del caché de páginas. Las escrituras de datos, así como las manipulaciones de directorios, se efectúan mediante el *búfer caché*.

5.3 Asignación de memoria para el núcleo

5.3.1 Asignación de páginas de memoria

Linux proporciona las funciones `__get_free_pages`, `get_free_page`, `free_pages`, `__free_page` y `free_page` para asignar y liberar páginas contiguas en memoria central.

El núcleo mantiene una lista de páginas disponibles en memoria utilizando el principio del *Buddy system* [Knowlton 1965]. Su principio es bastante simple: el núcleo mantiene una lista de grupos de páginas. Estos grupos son de tamaño fijo (pueden contener 1, 2, 4, 8, 16 o 32 páginas) y se refieren a páginas contiguas en memoria.

El principio básico del *Buddy System* es el siguiente: a cada petición de asignación, se usa la lista no vacía que contiene los grupos de páginas de tamaño inmediatamente superior al tamaño especificado, y se selecciona un grupo de páginas de esta lista. Este grupo se descompone en dos partes: las páginas correspondientes al tamaño de memoria especificado, y el resto de páginas que siguen disponibles. Este resto puede insertarse en las otras listas.

Al liberar un grupo de páginas, el núcleo intenta fusionar este grupo con los grupos disponibles, a fin de obtener un grupo disponible de tamaño máximo.

Para simplificar la asignación y liberación de un grupo de páginas, el núcleo sólo permite asignar grupos de páginas cuyo tamaño sea predeterminado y corresponda a los tamaños gestionados en las listas.

Si suponemos que se deben asignar 8 páginas y sólo está disponible un grupo de 32 páginas, el núcleo utiliza dicho grupo, asigna las 8 páginas solicitadas, y reparte las 24 páginas restantes en un grupo de 16 páginas y otro de 8 páginas. Estos dos grupos se insertan en las listas de grupos correspondientes.

Al liberar estas 8 páginas, el núcleo comprueba si las 8 páginas adyacentes están disponibles, es decir, si un grupo de páginas disponibles contiene estas páginas. Si no es así, se crea un grupo disponible para las 8 páginas liberadas. Si existe un grupo adyacente, su tamaño se modifica para incluir las 8 páginas liberadas, y el núcleo comprueba si puede fusionarlo con otro grupo de 16 páginas, y así sucesivamente.

La estructura `free_area_struct`, declarada en el archivo fuente `mm/page_alloc.c`, define el formato de los descriptores de lista de grupos:

tipo	campo	descripción
<code>struct page *</code>	<code>next</code>	Puntero a la primera página contenida en el primer grupo de páginas
<code>struct page *</code>	<code>prev</code>	Puntero no utilizado
<code>unsigned int *</code>	<code>map</code>	Puntero a una tabla de bits: cada bit indica si el grupo correspondiente está asignado o disponible

La tabla `free_area`, declarada en el archivo fuente `mm/page_alloc.c`, contiene la dirección del primer descriptor de grupo de páginas disponible para cada tamaño de grupo.

5.3.2 Asignación de zonas de memoria

Linux ofrece varios tipos de funciones que permiten asignar zonas de memoria para la utilización propia del núcleo:

- `kmalloc` y `kfree` permiten asignar y liberar zonas de memoria formadas por páginas contiguas en memoria central;
- `vmalloc` y `vfree` permiten asignar y liberar zonas de memoria formadas por páginas que no son forzosamente contiguas en memoria central. El uso de estas funciones se explica en la sección 5.5.

Para la implementación de `kmalloc` y `kfree`, Linux utiliza listas de zonas disponibles. Para cada tamaño de zona, se gestiona una lista de páginas descompuestas en bloques. Los tamaños de zonas gestionadas en la arquitectura x86 son los siguientes: 32, 64, 128, 252, 508, 1020, 2040, 4080, 8176, 16368, 32752, 65520 y 131056. Cuando

`kmalloc` se llama con un tamaño especificado, se usa la lista correspondiente al tamaño inmediatamente superior.

A cada una de las listas se le asocia un descriptor. La estructura `size_descriptor`, declarada en el archivo fuente `mm/kmalloc.c`, define el formato de este descriptor:

tipo	campo	descripción
<code>struct page_descriptor *</code>	<code>firstfree</code>	Puntero al descriptor del primer grupo de páginas disponible en la lista
<code>struct page_descriptor *</code>	<code>dmafree</code>	Puntero al descriptor del primer grupo de páginas utilizables para un acceso DMA (es decir, páginas de memoria situadas en los primeros 16 MB de memoria central) disponible en la lista
<code>int</code>	<code>nblocks</code>	Número de bloques de memoria en una página
<code>int</code>	<code>nmablocks</code>	Número de bloques asignados de esta lista
<code>int</code>	<code>nfrees</code>	Número de bloques libres en esta lista
<code>int</code>	<code>nbytesmalloved</code>	Número de bytes asignados en la lista
<code>int</code>	<code>npages</code>	Número de grupos de páginas asignadas a la lista
<code>unsigned long</code>	<code>gfporder</code>	Número de páginas a asignar (expresado en logaritmo de 2)

A cada descriptor de lista se le vincula una lista de grupo de páginas de memoria. Cada grupo contiene una o varias páginas contiguas en memoria central. El número de páginas del grupo está en función del tamaño de los bloques almacenados en la lista: debe ser suficiente para contener al menos un bloque. Este número puede expresarse mediante el campo `gfporder`, por la fórmula: 2^{gfporder} .

El inicio de cada grupo de páginas contiene un descriptor. El formato de dicho descriptor se define por la estructura `page_descriptor`:

tipo	campo	descripción
<code>struct page_descriptor *</code>	<code>next</code>	Puntero al descriptor del grupo de páginas siguiente de la lista
<code>struct block_header *</code>	<code>firstfree</code>	Puntero al descriptor del primer bloque libre en el grupo de páginas
<code>int</code>	<code>order</code>	Número de páginas a asignar (expresado en logaritmo de 2)
<code>int</code>	<code>nfrees</code>	Número de bloques libres en el grupo de páginas

Cada grupo de páginas se descompone en una tabla de bloques de tamaño fijo. Al principio de cada bloque se encuentra un descriptor de bloque definido por la estructura `block_header`:

tipo	campo	descripción
<code>unsigned long</code>	<code>bh_flags</code>	Estado del bloque (asignado, utilizable para un acceso DMA o libre)
<code>unsigned long</code>	<code>bh_length</code>	Tamaño del bloque
<code>struct block_header *</code>	<code>bh_next</code>	Puntero al descriptor de bloque siguiente en el grupo de páginas

La figura 8.5 representa estas estructuras de datos. En la figura, dos grupos de páginas están contenidos en la primera lista. El primer grupo contiene dos bloques libres, mientras que el segundo sólo contiene uno.

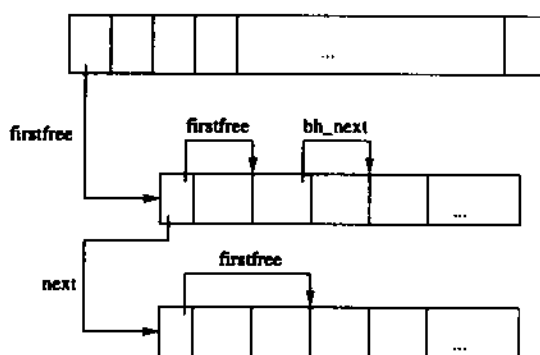


FIG. 8.5 – Listas de bloques de memoria disponibles

5.4 Espacios de direccionamiento de los procesos

5.4.1 Descriptores de regiones de memoria

El espacio de direccionamiento de los procesos puede estar formado por varias regiones de memoria, como se explica en la sección 3.1.

El núcleo mantiene en memoria una descripción de las regiones utilizadas por un proceso. La estructura `vm_area_struct`, declarada en el archivo de cabecera `<linux/mm.h>`, define el formato del descriptor de cada región:

tipo	campo	descripción
<code>struct mm_struct *</code>	<code>vm_mm</code>	Puntero al descriptor del espacio de direccionamiento del proceso afectado
<code>unsigned long</code>	<code>vm_start</code>	Dirección de inicio de la zona
<code>unsigned long</code>	<code>vm_end</code>	Dirección de fin de la zona
<code>pgprot_t</code>	<code>vm_page_prot</code>	Protección asociada a la región de memoria
<code>unsigned short</code>	<code>vm_flags</code>	Estado de la región de memoria (ver más adelante)
<code>short</code>	<code>vm_avl_height</code>	Peso asociado al descriptor en tanto que nodo del AVL (ver la sección 5.4.3)
<code>struct vm_area_struct *</code>	<code>vm_avl_left</code>	Puntero al subárbol izquierdo en el AVL
<code>struct vm_area_struct *</code>	<code>vm_avl_right</code>	Puntero al subárbol derecho en el AVL

<code>struct</code>	<code>vm_next</code>	Puntero al descriptor siguiente en la lista de zonas de memoria asociadas a un proceso
<code>vm_area_struct *</code>		
<code>struct</code>	<code>vm_next_share</code>	Puntero al descriptor siguiente en la lista de zonas de memoria compartidas
<code>vm_area_struct *</code>		
<code>struct</code>	<code>vm_prev_share</code>	Puntero al descriptor anterior en la lista de zonas de memoria compartidas
<code>vm_area_struct *</code>		
<code>struct</code>	<code>vm_ops</code>	Lista de operaciones asociadas a la zona de memoria
<code>vm_operations_struct *</code>		
<code>unsigned long</code>	<code>vm_offset</code>	Dirección de inicio de la zona de memoria respecto al inicio del objeto proyectado en memoria
<code>struct inode *</code>	<code>vm_inode</code>	Descriptor del i-nodo proyectado en memoria en esta zona

El valor del campo `vm_flags` se expresa en función de las constantes siguientes:

constante	significado
VM_READ	La región de memoria es accesible en lectura
VM_WRITE	La región de memoria es accesible en escritura
VM_EXEC	La región de memoria es accesible en ejecución
VM_SHARED	La región de memoria es compartida entre varios procesos
VM_MAYREAD	La protección de la región de memoria puede modificarse para hacerla accesible en lectura
VM_MAYWRITE	La protección de la región de memoria puede modificarse para hacerla accesible en escritura
VM_MAYEXEC	La protección de la región de memoria puede modificarse para hacerla accesible en ejecución
VM_MAYSHARE	La región de memoria puede compartirse entre varios procesos
VM_GROWSDOWN	La región de memoria crece por abajo
VM_GROWSUP	La región de memoria crece por arriba
VM_SEM	La región de memoria corresponde a un segmento de memoria compartida
VM_DENYWRITE	Debe devolverse el error <code>EBUSY</code> en el caso en que un proceso intente escribir en el archivo proyectado en esta región de memoria
VM_LOCKED	La región de memoria está bloqueada

5.4.2 Descriptores de espacio de direccionamiento

Linux mantiene un descriptor del espacio de direccionamiento. Este descriptor es accesible por el campo `mm` contenido en el descriptor del proceso. Cada proceso posee normalmente un descriptor propio, pero dos procesos clones pueden compartir el mismo descriptor si la opción `CLONE_VM` se especifica al llamar a la primitiva `clone`.

La estructura `mm_struct`, declarada en el archivo de cabecera `<linux/sched.h>`, define el formato de este descriptor:

tipo	campo	descripción
<code>int</code>	<code>count</code>	Número de referencias (es decir, número de procesos clones que comparten este descriptor)

<code>pgd_t *</code>	<code>pgd</code>	Dirección de la tabla global de páginas utilizada por el proceso
<code>unsigned long</code>	<code>start_code</code>	Dirección virtual de inicio del código
<code>unsigned long</code>	<code>end_code</code>	Dirección virtual de fin del código
<code>unsigned long</code>	<code>start_data</code>	Dirección virtual de inicio de datos
<code>unsigned long</code>	<code>end_data</code>	Dirección virtual de inicio de bloques de memoria asignados llamando a <code>brk</code>
<code>unsigned long</code>	<code>start_brk</code>	Dirección virtual de inicio de los bloques de memoria asignados llamando a <code>brk</code>
<code>unsigned long</code>	<code>brk</code>	Dirección virtual de fin de los bloques de memoria asignados llamando a <code>brk</code>
<code>unsigned long</code>	<code>start_stack</code>	Dirección virtual de inicio de la pila
<code>unsigned long</code>	<code>arg_start</code>	Dirección virtual de inicio del bloque de memoria que contiene los argumentos del programa ejecutado
<code>unsigned long</code>	<code>arg_end</code>	Dirección virtual de fin del bloque de memoria que contiene los argumentos del programa ejecutado
<code>unsigned long</code>	<code>env_start</code>	Dirección virtual de inicio del bloque de memoria que contiene las variables de entorno
<code>unsigned long</code>	<code>env_end</code>	Dirección virtual de fin del bloque de memoria que contiene las variables de entorno
<code>unsigned long</code>	<code>rss</code>	Número de páginas residentes en memoria para el proceso
<code>unsigned long</code>	<code>total_vm</code>	Número total de bytes contenidos en el espacio de direccionamiento
<code>unsigned long</code>	<code>locked_vm</code>	Número de bytes bloqueados en memoria
<code>unsigned long</code>	<code>def_flags</code>	Estado a utilizar de modo predeterminado en la creación de regiones de memoria
<code>struct vm_area_struct *</code>	<code>mmap</code>	Dirección del primer descriptor de región que forma parte del espacio de direccionamiento (ver más adelante)
<code>struct vm_area_struct *</code>	<code>mmap_avl</code>	Dirección del descriptor raíz del AVL que contiene los descriptors de regiones de memoria (ver más adelante)

5.4.3 Organización de los descriptors de regiones

Los descriptors de región de un proceso son referenciados por el descriptor de espacio de direccionamiento de un proceso de dos formas diferentes.

Se mantiene una lista de descriptores. El campo `mmap` contiene la dirección del primer descriptor de región asociado al proceso, y cada descriptor contiene la dirección del siguiente en su campo `vm_next`. La lista se ordena por dirección de fin de regiones.

Esta lista no se utiliza para buscar un descriptor de región particular. Para acelerar las búsquedas, los descriptores se colocan también en un árbol AVL (Adelson-Velskii and Landis), lo que permite reducir la complejidad de la búsqueda de $O(n)$ a $O(\log n)$.

Un AVL es un árbol binario que siempre está equilibrado. A cada nodo del árbol se le asocia un peso, que corresponde a la diferencia entre las profundidades de su subárbol izquierdo y de su subárbol derecho, y este peso siempre debe tener como valor -1 , 0 o 1 . Si este peso se modifica más allá de estos límites, al añadir o suprimir elementos en el árbol, se efectúan operaciones de rotación para reequilibrar el AVL, como se muestra en la figura 8.6. Puede encontrarse una descripción más completa de las características de los AVL y de los algoritmos de manipulación asociados en [Froidevaux *et al.* 1993].

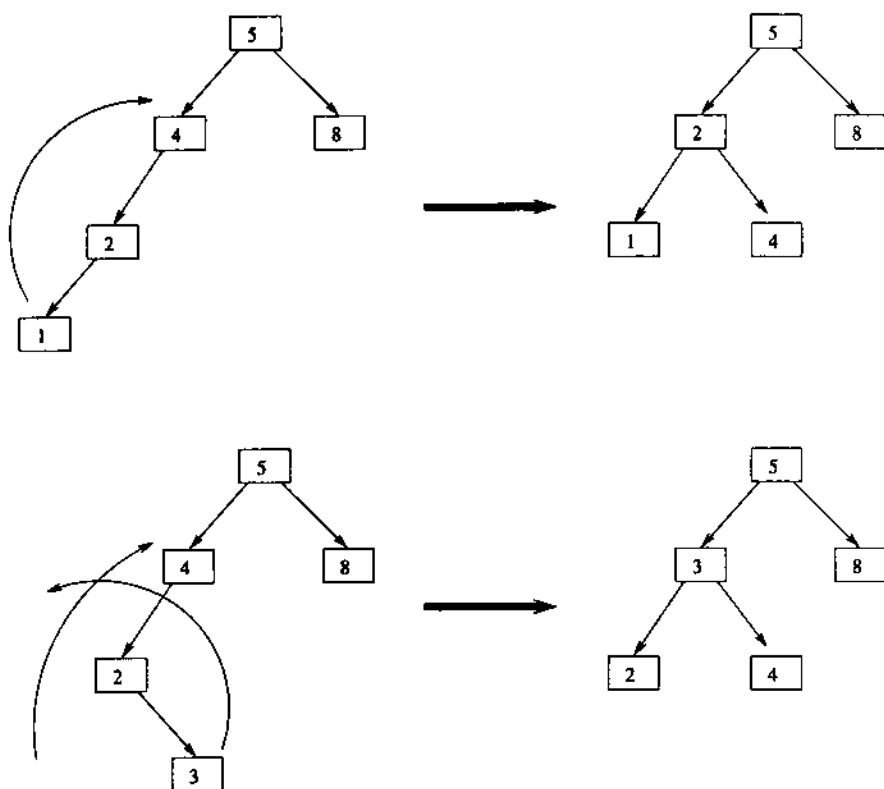


FIG 8.6 - Ejemplos de rotaciones en un AVL

El campo `mmap_avl` contiene la dirección del descriptor de región utilizado como nodo raíz del AVL.

La figura 8.7 representa la organización del AVL para un proceso cuyo espacio de direccionamiento contiene las regiones siguientes:

08048000-0804a000	r-xp	00000000	03:02	7914
0804a000-0804b000	rw-p	00001000	03:02	7914
0804b000-08053000	rwxp	00000000	00:00	0
40000000-40005000	rwxp	00000000	03:02	18336
40005000-40006000	rw-p	00004000	03:02	18336
40006000-40007000	rw-p	00000000	00:00	0
40007000-40009000	r--p	00000000	03:02	18255
40009000-40082000	r-xp	00000000	03:02	18060
40082000-40087000	rw-p	00078000	03:02	18060
40087000-400b9000	rw-p	00000000	00:00	0
bffffe000-c0000000	rwxp	ffffff000	00:00	0

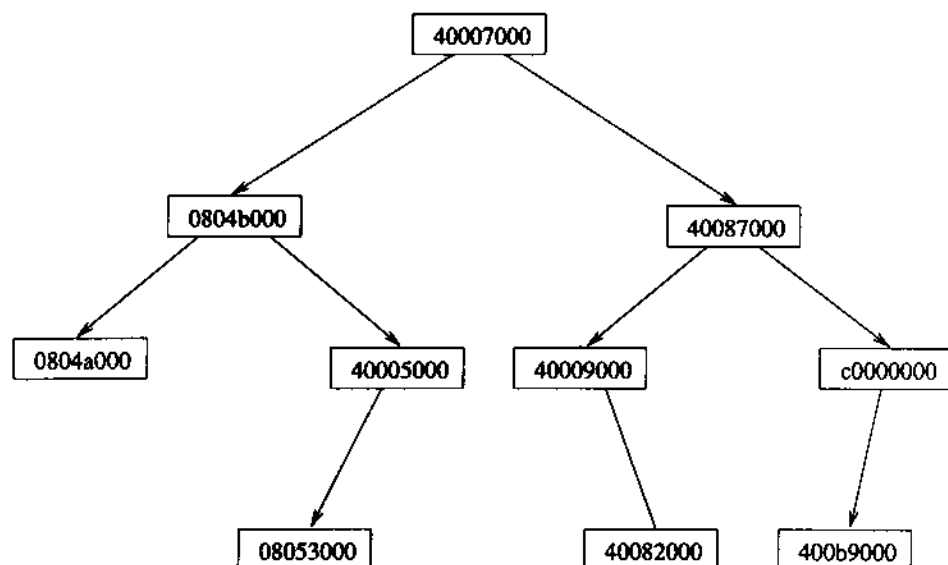


FIG 8.7 – Organización de las regiones de memoria de un proceso

5.5 Asignación de memoria no contigua al núcleo

Las funciones `kmalloc` y `kfree` permiten al núcleo asignar y liberar zonas de memoria contenidas en páginas contiguas. Aunque son muy útiles para la gestión de zonas de pequeño tamaño, son notoriamente ineficaces para zonas de grandes tamaños, ya que utilizan bloques de tamaño fijo y devuelven un bloque cuyo tamaño puede ser muy superior al tamaño solicitado. Por ejemplo, si debe asignarse un bloque de 17 kilobytes en la arquitectura x86, `kmalloc` debe utilizar un bloque de 32 kilobytes, y por tanto se asignan 15 kilobytes de manera superflua.

Las funciones `vmalloc` y `vfree` no poseen este inconveniente. En una asignación de memoria, se asignan varias páginas de memoria, que no forzosamente son contiguas. Estas páginas se insertan seguidamente en el segmento reservado al núcleo modificando las tablas de páginas. De este modo, el tamaño de la memoria «desperdiciada» es menos importante.

5.6 Gestión del *swap*

5.6.1 Formato de los dispositivos de *swap*

Bajo Linux, todo dispositivo en modo bloque o archivo regular puede usarse como dispositivo de *swap*. Sin embargo, un dispositivo de *swap* debe haber sido inicializado por el mandato `mkswap`.

Este mandato crea un catálogo al arrancar el dispositivo. Este catálogo tiene el tamaño de una página de memoria y está constituido por una tabla de bits. Cada uno de los bits indica si la página correspondiente en el dispositivo es utilizable. Si suponemos, por ejemplo, que el mandato `mkswap` se ejecuta sobre un archivo de cuatro megabytes, en una máquina basada en un procesador x86 que utiliza un tamaño de página de cuatro kilobytes, el contenido del catálogo es el siguiente:

- el primer bit está a 1, e indica que la primera página, que contiene el catálogo, no es utilizable;
- los 1.023 bits siguientes están a 0, e indican que las páginas correspondientes son utilizables;
- los bits siguientes están a 1, para indicar que las páginas siguientes no existen en el archivo.

La primera página contiene también una firma: los 10 últimos bytes contienen la cadena de caracteres «SWAP-SPACE». Esta firma permite que el núcleo verifique la validez de un dispositivo en su activación.

El formato de un dispositivo de *swap* define su tamaño límite. En la medida en que un bit debe existir en el catálogo (que tiene el tamaño de una página) para cada página, el número máximo de páginas puede expresarse por la fórmula: $(\text{tamaño_página} - 10) * 8$. En la arquitectura x86, el tamaño máximo de un dispositivo de *swap* es pues de unos 128 megabytes.

5.6.2 Descriptores de dispositivos de *swap*

El núcleo guarda en memoria una lista de dispositivos de *swap* activos. Se utiliza una tabla de descriptores, en la que cada uno describe un dispositivo de *swap*.

La estructura `swap_info_struct`, declarada en el archivo de cabecera `<linux/swap.h>`, define el formato de estos descriptores. Contiene los campos siguientes:

tipo	campo	descripción
unsigned int	flags	Estado del dispositivo
kdev_t	swap_device	Identificador del dispositivo en modo bloque
struct inode *	swap_file	Puntero al descriptor del i-nodo correspondiente, en el caso de un archivo regular
unsigned char *	swap_map	Puntero a una tabla de bytes que representan el estado de cada página
unsigned char *	swap_lockmap	Puntero a una tabla de bits que indican por cada página si está bloqueada o es utilizable
int	lowest_bit	Número de la menor página utilizable
int	highest_bit	Número de la mayor página utilizable
int	prio	Prioridad asociada al dispositivo
int	pages	Número de páginas disponibles (asignadas o no)
unsigned long	max	Número de páginas en el dispositivo
int	next	Puntero al descriptor siguiente en la lista

Los campos `swap_map` y `swap_lockmap` se utilizan para mantener actualizadas las páginas utilizadas. `swap_map` apunta a una tabla de bytes, cada uno de los cuales contiene el número de referencias a la página. Si este número es nulo, significa que la página está disponible, si no está asignada.

`swap_lockmap` apunta a una tabla de bits, en la que cada miembro indica si la página correspondiente está bloqueada (si está a 1) o está libre (si está a 0). Esta tabla se inicializa a partir de la que se encuentra en el catálogo del dispositivo de *swap*. En las entradas/salidas sobre el dispositivo, el bit correspondiente se posiciona a 1 para impedir cualquier otra lectura o escritura durante la entrada/salida.

5.6.3 Direcciones de entradas del *swap*

Cuando una página se escribe en un dispositivo de *swap*, se le atribuye una dirección. Esta dirección combina el número del dispositivo de *swap* y el índice de la página utilizada en el dispositivo.

Varias macroinstrucciones, declaradas en el archivo de cabecera `<asm/pg-table.h>`, manipulan estas direcciones:

macroinstrucción	significado
SWP_ENTRY	Combina un número de dispositivo y un índice de página para formar una dirección de entrada de <i>swap</i>
SWP_TYPE	Devuelve el número del dispositivo correspondiente a una dirección de entrada de <i>swap</i>
SWP_OFFSET	Devuelve el índice de la página correspondiente a una dirección de entrada de <i>swap</i>

Cuando una página debe descartarse de la memoria, se le asigna una página en un dispositivo de *swap*, y la dirección de dicha página se memoriza para que Linux pueda recargar la página ulteriormente. En lugar de utilizar una tabla que establezca una correspondencia entre las direcciones de páginas de memoria y las direcciones de entradas de *swap*, Linux utiliza un método original:

- Cuando una página de memoria se descarta, la dirección de la entrada de *swap* asignada se guarda en la tabla de páginas, en lugar de la dirección física de la página. Esta dirección está concebida para indicar al procesador que la página no está presente en memoria.
- Cuando se efectúa un acceso de memoria sobre una página que se ha guardado en un dispositivo de *swap*, el procesador detecta que la página no está presente en memoria y desencadena una interrupción. Al tratar esta interrupción, Linux extrae la dirección de entrada de *swap* de la entrada correspondiente de la tabla de páginas, y utiliza esta dirección para localizar la página en el *swap*.

5.6.4 Selección de páginas a descartar

A fin de descartar páginas de la memoria, se ejecuta el proceso *kswapd*. Este proceso se lanza al arrancar el sistema y se ejecuta en modo núcleo. Es más o menos equivalente al proceso *bdflush*, descrito en el capítulo 6, sección 6.2.7.

La función del proceso *kswapd* es descartar las páginas inútiles de la memoria. La mayor parte del tiempo, *kswapd* se duerme y se despierta cuando el núcleo se queda sin memoria. Entonces explora la lista de proceso e intenta descartar páginas no utilizadas.

A fin de determinar las páginas de memoria no utilizadas, el núcleo utiliza el campo `age` del descriptor de página de memoria. Este contador se incrementa cuando se utiliza la página, y se decrementa cuando deja de utilizarse. Sólo las páginas que poseen un campo `age` nulo pueden descartarse de la memoria.

5.7 Operaciones de memoria

Linux implementa no sólo una gestión de tablas de páginas independiente del procesador: además implementa una gestión genérica de la memoria. El módulo de gestión de la memoria gestiona las regiones de memoria, las tablas de páginas y el *swap*, pero no contiene los mecanismos de gestión del contenido de las regiones de memoria.

Para acceder al contenido de las regiones de memoria, el módulo de gestión de memoria utiliza operaciones de memoria dependientes de las regiones de memoria. Este funcionamiento es muy similar al del sistema virtual de archivos que llama a operaciones sobre superbloques, i-nodos y archivos, para efectuar los accesos a los archivos.

La estructura `vm_operations_struct`, declarada en el archivo de cabecera `<linux/mm.h>`, define las operaciones de memoria asociadas a las regiones de memoria. Estas operaciones son las siguientes:

- `void (*open)(struct vm_area_struct *area)`

La operación `open` se llama cuando se crea una nueva región de memoria, referenciada por `area`.

- `void (*close)(struct vm_area_struct *area)`

La operación `close` se llama cuando se suprime la región de memoria referenciada por `area`.

- `void (*unmap)(struct vm_area_struct *area, unsigned long start, size_t len)`

La operación `unmap` se llama cuando una parte de la región de memoria referenciada por `area` se suprime de la memoria. El parámetro `start` indica la dirección de inicio de la sección a suprimir, y `len` especifica su tamaño.

- `void (*protect)(struct vm_area_struct *area, unsigned long start, size_t len, unsigned int newprot)`

La operación `protect` se llama cuando las protecciones se modifican sobre una parte de la región de memoria referenciada por `area`. El parámetro `start` indica

la dirección de inicio de la sección, y `len` especifica su tamaño. Las nuevas protecciones a aplicar se especifican en el parámetro `newprot`.

- `int (*sync)(struct vm_area_struct *area, unsigned long start, size_t len, unsigned int flags)`

La operación `sync` se llama para reescribir el contenido de una parte de la región de memoria referenciada por `area`. El parámetro `start` indica la dirección de inicio de la sección, y `len` especifica su tamaño. El parámetro `flags` especifica las modalidades de reescritura.

- `void (*advise)(struct vm_area_struct *area, unsigned long start, size_t len, unsigned int advise)`

La operación `advise` no se utiliza por Linux 2.0.

- `unsigned long (*nopage)(struct vm_area_struct *area, unsigned long address, int write_access)`

La operación `nopage` se llama para cargar una página en la región de memoria referenciada por `area`. El parámetro `address` indica la dirección a la que la página debe cargarse, y `write_access` especifica si la página será accesible en escritura o no.

- `unsigned long (*wppage)(struct vm_area_struct *area, unsigned long address, unsigned long page)`

La operación `wppage` no es utilizada por Linux 2.0.

- `int (*swapout)(struct vm_area_struct *area, unsigned long page, pte_t *pte)`

La operación `swapout` se llama para guardar en memoria secundaria la página `page` de la región de memoria referenciada por `area`. El parámetro `pte` contiene la dirección de la entrada correspondiente en la tabla de páginas.

- `int (*swpin)(struct vm_area_struct *area, unsigned long page, unsigned long entry)`

La operación `swpin` se llama para recargar en memoria central la página `page` de la región de memoria referenciada por `area`.

6 Presentación detallada de la implementación

6.1 Asignación de memoria para el núcleo

Linux contiene varias funciones que permiten asignar dinámicamente regiones de memoria al núcleo, y liberarlas.

6.1.1 Asignación de páginas de memoria

El archivo fuente *mm/page_alloc.c* contiene las primitivas que permiten asignar y liberar páginas de memoria completas. Las funciones de este módulo gestionan listas de grupos de páginas disponibles, según el principio del *Buddy System*, presentado en la sección 5.3.1.

La función `init_mem_queue` inicializa una lista de grupo de páginas, indicando que la lista está vacía. La función `add_mem_queue` añade un grupo de páginas a una lista, y `remove_mem_queue` suprime un grupo de la lista donde está registrado.

La función `free_pages_ok` se llama para proceder a la liberación de un grupo de páginas. Procede a la actualización de las tablas de bits indicando los grupos asignados, y añade el grupo a su lista llamando a `add_mem_queue`.

La función `__free_page` decrementa el número de referencias de la página a liberar, y llama a `free_pages_ok` si llega a nulo. La función `free_pages` decrementa el número de referencias del grupo de páginas a liberar, y llama a `free_pages_ok` si llega a nulo.

La función `__get_free_pages` se llama para asignar un grupo de páginas. Utiliza la macroinstrucción `RMQUEUE` para buscar un grupo de bloques correspondiente. Esta macroinstrucción explora todas las listas correspondientes a tamaños de grupo superiores o iguales al tamaño solicitado. Cuando se encuentra un grupo de tamaño suficiente, la macroinstrucción `EXPAND` lo descompone en varios grupos: uno se asigna, y los demás permanecen disponibles y se colocan en las listas correspondientes. En el caso que `__get_free_pages` no encuentre el grupo de páginas de tamaño suficientemente importante, la función `try_to_free_page` se llama para intentar liberar páginas de memoria, y luego se reinicia el proceso.

La función `free_area_init` se llama en la inicialización del sistema. Asigna la tabla `mem_map` con un descriptor para cada página contenida en memoria central, y asigna las listas de grupos de páginas. Todas estas listas están vacías al crearse.

6.1.2 Asignación de regiones de memoria

El archivo fuente `mm/kmalloc.c` contiene la implementación de las funciones `kmalloc` y `kfree` que proceden a la asignación y a la liberación de zonas de memoria para el núcleo.

La función `get_kmalloc_pages` asigna nuevas páginas llamando a `__get_free_pages`. La función `free_kmalloc_pages` libera páginas llamando a `free_pages`. También puede colocar las páginas especificadas en un caché, sin liberarlas. Este caché será utilizado por `kmalloc` en caso de error en la asignación de páginas.

La función `kmalloc_init` se llama al inicializarse el sistema. Inicializa las listas utilizadas por `kmalloc`.

La función `kmalloc` asigna una nueva zona de memoria. Primero explora la lista de descriptors de listas para determinar la lista de páginas a utilizar. Si esta lista no está vacía, asigna el primer bloque y luego actualiza la lista. Si la lista está vacía, la función `get_kmalloc_pages` se llama para asignar nuevas páginas a la lista. Seguidamente se efectúa un bucle de creación de bloques libres en las páginas asignadas, y se asigna el primer bloque. En el caso que no puedan asignarse nuevas páginas, se usa el caché: si contiene páginas correspondientes al tamaño especificado, se utilizan éstas.

La función `kfree` libera una zona de memoria asignada por una llamada a `kmalloc`. El bloque especificado se marca como libre, y el número de bloques libres en las páginas correspondientes se incrementa. Si este número llega a ser igual al número de bloques por página, significa que el contenido de la página está completamente libre y se libera ésta llamando a `free_kmalloc_pages`.

6.2 Gestión de las tablas de páginas

El archivo fuente `mm/memory.c` contiene la gestión de las tablas de páginas. Utiliza el modelo independiente de la arquitectura descrito en la sección 5.1.3.

Las funciones dependientes de la arquitectura se definen en `<asm/pgtable.h>`:

- `pt_none`: devuelve el valor 1 si la entrada especificada de la tabla de páginas no está inicializada;
- `pte_present`: devuelve el valor 1 si la página especificada está presente en memoria;
- `pte_clear`: inicializa a 0 la entrada especificada de la tabla de páginas;
- `pmd_none`: devuelve el valor 1 si la entrada especificada de la tabla intermedia no se inicializa;
- `pmd_bad`: devuelve el valor 1 si la entrada especificada de la tabla intermedia es errónea;
- `pmd_present`: devuelve el valor 1 si la página que contiene la tabla intermedia está presente en memoria;
- `pmd_clear`: inicializa a 0 la entrada especificada de la tabla intermedia;
- `pgd_none`: devuelve el valor 1 si la entrada especificada de la tabla global no se inicializa;
- `pgd_bad`: devuelve el valor 1 si la entrada especificada de la tabla global es errónea;
- `pgd_present`: devuelve el valor 1 si la página que contiene la tabla global está presente en memoria;
- `pgd_clear`: inicializa a 0 la entrada especificada de la tabla global;
- `pte_read`: devuelve 1 si la página especificada es accesible en lectura;
- `pte_write`: devuelve 1 si la página especificada es accesible en escritura;
- `pte_exec`: devuelve 1 si la página especificada es accesible en ejecución;
- `pte_dirty`: devuelve 1 si el contenido de la página especificada se ha modificado;
- `pte_young`: devuelve 1 si el contenido de la página especificada ha sido accedido;
- `pte_wrprotect`: hace la página especificada inaccesible en escritura;
- `pte_rdprotect`: hace la página especificada inaccesible en lectura;
- `pte_exprotect`: hace la página especificada inaccesible en ejecución;
- `pte_mkclean`: pone a 0 el indicador de modificación del contenido de la página especificada;
- `pte_mkold`: pone a 0 el indicador de acceso al contenido de la página especificada;

- `pte_mkwrite`: hace la página especificada accesible en escritura;
- `pte_mkdir`: hace la página especificada accesible en lectura;
- `pte_mkexec`: hace la página especificada accesible en ejecución;
- `pte_mkdirty`: marca la página como modificada;
- `pte_mkyoung`: marca la página como accedida;
- `mk_pte`: devuelve el contenido de una entrada de la tabla de páginas combinando la dirección de la página de memoria asociada y su protección;
- `pte_modify`: modifica el contenido de una entrada de la tabla de páginas;
- `pte_page`: devuelve la dirección de la página de memoria contenida en una entrada de la tabla de páginas;
- `pmd_page`: devuelve la dirección de la página de memoria que contiene una tabla intermedia;
- `pgd_offset`: devuelve la dirección de una entrada de la tabla global;
- `pmd_offset`: devuelve la dirección de una entrada de la tabla intermedia;
- `pte_offset`: devuelve la dirección de una entrada de la tabla de páginas;
- `pte_free_kernel`: libera una tabla de páginas utilizada por el núcleo;
- `pte_alloc_kernel`: asigna una tabla de páginas utilizada por el núcleo;
- `pmd_free_kernel`: libera una tabla intermedia utilizada por el núcleo;
- `pmd_alloc_kernel`: asigna una tabla intermedia utilizada por el núcleo;
- `pte_free`: libera una tabla de páginas utilizada por un proceso;
- `pte_alloc`: asigna una tabla de páginas utilizada por un proceso;
- `pmd_free`: libera una tabla intermedia utilizada por un proceso;
- `pmd_alloc`: asigna una tabla intermedia utilizada por un proceso;
- `pgd_free`: libera la tabla global utilizada por un proceso;
- `pgd_alloc`: asigna la tabla global utilizada por un proceso.

Las funciones independientes de la arquitectura son las siguientes:

- `copy_page`: copia el contenido de una página de memoria en otra;
- `com`: muestra un mensaje de error, y envía la señal SIGKILL al proceso que haya sobrepasado la memoria disponible;
- `free_one_pmd`: libera la tabla de páginas apuntada por una entrada de la tabla intermedia;
- `free_one_pgd`: libera la tabla intermedia apuntada por una entrada de la tabla global, llamando a `free_one_pmd`;
- `clear_page_tables`: libera las tablas asociadas al espacio de direccionamiento de un proceso usuario, llamando a `free_one_pmd` para cada tabla intermedia;
- `free_page_tables`: libera las tablas asociadas al espacio de direccionamiento de un proceso usuario, llamando a `free_one_pmd` para cada tabla intermedia, y seguidamente a `pgd_free`;
- `new_page_tables`: asigna nuevas tablas para un proceso;
- `copy_one_pte`: copia el contenido de una entrada de la tabla de páginas en otra, borrando la autorización de escritura si la copia en escritura se especifica;
- `copy_pte_range`: copia el contenido de una serie de entradas de la tabla de páginas, llamando a `copy_one_pte` para cada entrada;
- `copy_pmd_range`: copia el contenido de una serie de entradas de la tabla intermedia, llamando a `copy_pte_range` para cada entrada de la tabla intermedia;
- `copy_page_range`: copia el contenido de una región de memoria, llamando a la función `copy_pmd_range` para cada tabla intermedia afectada;
- `free_pte`: libera una página apuntada por una entrada de la tabla de páginas;
- `forget_pte`: libera una página apuntada por una entrada de la tabla de páginas, si esta entrada no es nula;
- `zap_pte_range`: libera varias páginas llamando a `free_pte`;
- `zap_pmd_range`: libera una serie de entradas de la tabla intermedia, llamando a la función `zap_pte_range` para cada entrada de la tabla intermedia;
- `zap_page_range`: libera el contenido de una región de memoria, llamando a la función `zap_pmd_range` para cada tabla intermedia afectada;
- `zeromap_pte_range`: inicializa varias entradas de la tabla de páginas con el mismo descriptor;

- `zeromap_pmd_range`: asigna tablas de páginas y las inicializa, llamando a la función `zeromap_pte_range`;
- `zeromap_page_range`: asigna varias tablas intermedias, y llama a la función `zeromap_pmd_range` para cada tabla, para inicializar las entradas de las tablas de páginas;
- `remap_pte_range`: modifica las direcciones de páginas contenidas en varias entradas de la tabla de páginas;
- `remap_pmd_range`: asigna tablas de páginas y modifica las direcciones contenidas, llamando a `remap_pte_range`;
- `remap_page_range`: asigna varias tablas intermedias y modifica las entradas de tablas, llamando a la función `remap_pmd_range`.

6.3 Gestión de las regiones de memoria

Las regiones de memoria asociadas a los procesos se gestionan por las funciones contenidas en el archivo fuente `mm/mmap.c`.

Se definen dos funciones de apoyo en el archivo de cabecera `<linux/mm.h>`:

- `find_vma`: esta función explora el AVL que contiene los descriptores de regiones de memoria asociadas a un proceso para buscar una región de memoria con la dirección especificada, o la primera región situada tras la dirección indicada.
- `find_vma_intersection`: esta función se llama para buscar el descriptor de una región de memoria con una intersección con una región especificada. Llama a `find_vma` para obtener el descriptor de región con la dirección de fin especificada. Si se encuentra un descriptor, comprueba si las dos regiones poseen una intersección no vacía comparando sus direcciones de inicio y de fin.

La función `get_unmapped_area` busca una región de memoria no asignada en el espacio de direccionamiento del proceso actual, a partir de una dirección especificada. Llama a `find_vma` para obtener el descriptor de la región situada justo tras la dirección especificada. Este tratamiento se efectúa haciendo variar la dirección mientras no se encuentre una zona de memoria no utilizada de tamaño suficiente.

Se definen varias funciones de gestión del AVL:

- `avl_neighbours`: esta función explora el AVL y devuelve los vecinos de un nodo, es decir, el nodo mayor de su subárbol izquierdo y el más pequeño de su subárbol derecho.
- `avl_rebalance`: esta función efectúa las rotaciones necesarias para mantener el AVL equilibrado.
- `avl_insert`: esta función inserta un nuevo nodo en el árbol.
- `avl_insert_neighbours`: esta función inserta un nuevo nodo en el árbol y devuelve sus vecinos.
- `avl_remove`: esta función suprime un nodo del árbol.

La función `build_mmap_avl` construye el AVL que contiene los descriptores de las regiones de memoria de un proceso llamando a `avl_insert` para cada región.

La función `insert_vm_struct` añade una región de memoria en el espacio de direccionamiento de un proceso. Llama a `avl_insert_neighbours` para insertar el descriptor de la región en el AVL, e inserta este descriptor en la lista de regiones encadenándolo a los descriptores de las regiones vecinas.

La función `remove_shared_vm_struct` suprime un descriptor de región de memoria de un i-nodo.

La función `merge_segments` se llama para fusionar las regiones de memoria de un proceso entre dos direcciones especificadas. Obtiene la primera región de memoria por una llamada a `find_vma`, y explora todas las regiones situadas delante de la dirección de fin. Para cada región, comprueba si la región es adyacente y si las características de las dos regiones son idénticas. Si es así, las dos regiones se fusionan en una sola.

6.4 Tratamiento de las excepciones

El procesador provoca una excepción en ciertos accesos a la memoria:

- acceso incompatible con la protección asociada a una página de memoria;
- acceso a una página no presente en memoria.

Las funciones llamadas en el tratamiento de una excepción de memoria se definen en el archivo fuente `mm/memory.c`:

- `do_wp_page`: esta función se llama para gestionar la copia en escritura, cuando un proceso accede en escritura a una página compartida y protegida en lectura exclusiva. Se asigna una nueva página llamando a `__get_free_page`, y se comprueba si la página afectada por la excepción es compartida entre varios procesos. Si es así, su contenido se copia en la nueva página, que se inserta en la tabla de páginas del proceso actual por `set_pte`, y el número de referencias a la anterior página se decrementa por una llamada a `free_page`. En el caso en que la página afectada no sea compartida, su protección simplemente se modifica para hacer posible la escritura.
- `do_swap_page`: esta función se llama para volver a cargar en memoria el contenido de una página situada en el espacio de *swap*. Si una operación *swpin* está asociada a la región de memoria que contiene la página, se llama. En caso contrario, se llama a la función `swap_in`. En ambos casos, la página asignada se inserta en el espacio de direccionamiento del proceso actual.
- `do_no_page`: esta función se llama en el acceso a una página no presente en memoria. En primer lugar, comprueba si la página ha sido descartada de la memoria y está contenida en el espacio de *swap*. Si es así, `do_swap_page` se llama para volver a cargar el contenido de la página en memoria, si no `do_no_page` comprueba si una operación de memoria *nopage* está asociada a la región que contiene la página. Si es así, esta operación se llama para cargar el contenido de la página en memoria, y la página se inserta en la tabla de páginas del proceso actual. Si no hay ninguna operación *nopage* asociada a la región de memoria que contiene la página, debe asignarse una nueva página rellena con ceros: la página se asigna por una llamada a `__get_free_page`, su contenido se inicializa con ceros, y se inserta en la tabla de páginas del proceso actual.
- `handle_pte_fault`: esta función se llama para tratar una excepción de memoria. Se determina si la excepción afecta a una página no presente, se llama a `do_no_page` en este caso, o si la página es compartida y debe copiarse, se llama a `do_wp_fault`.
- `handle_mm_fault`: esta función se llama en el tratamiento de una excepción de memoria en las arquitecturas que poseen un gestor de memoria separado del procesador. Se llama a `handle_pte_fault`, y se actualiza el caché del gestor de memoria.

La función `do_page_fault`, declarada en el archivo fuente `arch/i386/mm/fault.c` para la arquitectura x86, se llama para tratar las excepciones de memoria. Primero obtiene el descriptor de la región de memoria afectada llamando a `find_vma`, y luego comprueba el tipo de error:

- Si el error lo ha causado un acceso a una página no presente en el segmento de pila, se llama a la función `expand_stack` para aumentar el tamaño de la pila.
- Si el error se debe a un acceso en escritura en una región de memoria protegida en lectura, el error se señala al proceso actual enviándole la señal `SIGSEGV`.
- Si el error es debido a un acceso en escritura a una página de memoria protegida en lectura, a fin de implementar la copia en escritura, se llama a la función `do_wp_page` para efectuar la copia.
- Si el error se debe a un acceso a una página no presente en memoria, se llama a la función `do_no_page` a fin de cargar la página en memoria.

6.5 Acceso al espacio de direccionamiento de los procesos

Como se expone en la sección 5.1.1, Linux utiliza los mecanismos de segmentación de memoria para proteger los espacios de direccionamiento de los procesos y del núcleo, en ciertas arquitecturas. El núcleo no puede, pues, acceder directamente a los datos referenciados por una dirección proporcionada por un proceso.

Varias funciones permiten al núcleo acceder al contenido del espacio de direccionamiento de los procesos. Estas funciones, definidas en el archivo de cabecera `<asm/segment.h>`, son las siguientes:

- `memcpy_fromfs`: lectura de datos desde el espacio de direccionamiento del proceso que llama;
- `memcpy_tofs`: escritura de datos en el espacio de direccionamiento del proceso que llama;
- `get_user`: lectura de un entero compuesto por 1, 2 o 4 bytes desde el espacio de direccionamiento del proceso que llama;
- `put_user`: escritura de un entero compuesto de 1, 2 o 4 bytes en el espacio de direccionamiento del proceso que llama;
- `get_fs`: devuelve el valor del registro de segmento `fs`, que apunta al segmento de datos del proceso que llama;
- `get_ds`: devuelve el valor del registro de segmento `ds`, que apunta al segmento de datos del proceso que llama;
- `set_fs`: modifica el valor del registro de segmento `fs`, que apunta al segmento de datos del proceso que llama.

Además de estas funciones de acceso, la función `verify_area`, definida en el archivo fuente `mm/memory.c`, es utilizada por el núcleo para comprobar la validez de un puntero proporcionado por el proceso que llama. Utiliza `find_vma` para obtener el descriptor de la región que contiene la dirección especificada, y verifica que la protección asociada a la región es compatible con el acceso solicitado (lectura o escritura). `verify_area` efectúa esta verificación sobre varias regiones, si la zona a verificar se extiende por varias regiones. En el caso en que las protecciones de memoria no sean compatibles con el acceso solicitado, se devuelve el error `EFAULT`.

Hay que destacar que `verify_area` tiene en cuenta el funcionamiento del procesador sobre el que Linux se ejecuta. En ciertos procesadores, como el i386, las protecciones de memoria se ignoran en modo núcleo. Sobre tal procesador, si el acceso en escritura se especifica, `verify_area` llama a `do_wp_page` para duplicar las páginas compartidas.

6.6 Modificación de regiones de memoria

6.6.1 Creación y supresión de regiones de memoria

El archivo fuente `mm/mmap.c` contiene la implementación de las llamadas al sistema que permiten crear y suprimir regiones de memoria (primitivas `mmap` y `munmap`). Contiene también la implementación de la llamada al sistema `brk`.

La función `do_mmap` procede a la creación de una región de memoria. Primero verifica la validez de sus parámetros. En el caso en que el contenido de un archivo deba proyectarse en memoria en la nueva región, verifica también que el modo de apertura del archivo sea compatible con las modalidades de la proyección. Seguidamente se asigna e inicializa un descriptor de región. Si la región de memoria debe corresponder a un archivo proyectado en memoria, se llama a la operación `mmap` asociada al i-nodo del archivo. Finalmente, el descriptor de la región se inserta en el espacio de direccionamiento del proceso por una llamada a `insert_vm_struct`, y se llama a `merge_segments` para fusionar regiones de memoria si ello es posible.

La función `unmap_fixup` procede a la liberación de una parte de una región de memoria, cuyo descriptor ha sido suprimido ya del espacio de direccionamiento del proceso actual. Pueden producirse cuatro casos:

- la región completa debe liberarse: se llama a la operación de memoria `close` asociada;

- una sección situada al principio de la región debe liberarse: la dirección de inicio de la región se actualiza en el descriptor de región;
- debe liberarse una sección situada al final de la región: la dirección de fin de la región se actualiza en el descriptor de región;
- debe liberarse una sección situada en medio de la región: la región se descompone entonces en dos. Se asigna un nuevo descriptor de región y se inicializa. El descriptor de la región original se actualiza. Finalmente, el nuevo descriptor se inserta en el espacio de direccionamiento llamando a `insert_vm_struct`.

En los tres últimos casos, `unmap_fixup` crea un nuevo descriptor para la región modificada, y la inserta en el espacio de direccionamiento del proceso actual.

La función `do_munmap` libera las regiones de memoria situadas en un intervalo especificado. Primero explora la lista de regiones de memoria contenidas en el intervalo, memoriza sus descriptors en una lista, y los suprime del espacio de direccionamiento del proceso llamando a `avl_remove`. Tras este bucle, explora la lista de descriptors guardados anteriormente. Por cada región se llama a la operación de memoria `unmap`, las páginas correspondientes se suprimen de la tabla de páginas llamando a `zap_page_range`, se llama a la función `unmap_fixup`, y el descriptor de la región se libera.

La función `sys_munmap` implementa la llamada al sistema `munmap` llamando simplemente a `do_munmap`. La implementación de la llamada al sistema `mmap` se encuentra en el archivo fuente `arch/i386/kernel/sys_i386.c` para la arquitectura x86. La función `old_mmap` controla la validez de sus parámetros y llama a la función `do_mmap`.

La función `sys_brk` implementa la primitiva `brk`. Ésta verifica que la dirección pasada está situada en el segmento de datos, y que no sobrepasa los límites impuestos al proceso. En el caso en que la dirección de fin del segmento de datos sea disminuida, se llama a la función `do_munmap` para liberar la zona de memoria especificada. En el caso en que el tamaño del segmento de datos deba incrementarse, se llama a la función `find_vma_intersection` para verificar que la zona a asignar no entra en conflicto con una región existente, y seguidamente se extiende la región de memoria del segmento llamando a la función `do_mmap`.

6.6.2 Bloqueo de regiones de memoria

El archivo fuente `mm/mlock.c` contiene las funciones que implementan el bloqueo de regiones de memoria.

La función `mlock_fixup_all` bloquea o desbloquea una región entera de memoria. Únicamente modifica el campo `vm_flags` del descriptor de la región.

La función `mlock_fixup_start` bloquea o desbloquea una zona situada al principio de una región. La región debe descomponerse en dos partes: la primera representa la parte cuyo estado cambia, y la segunda representa la parte la parte cuyo estado no se modifica. Se asigna un descriptor para la nueva región, y se inicializa. El contenido del descriptor de la región original también se modifica, a fin de indicar que la región empieza tras la región creada. Finalmente, se inserta el descriptor de la nueva región en el espacio de direccionamiento del proceso llamando a `insert_vm_struct`.

La función `mlock_fixup_end` bloquea o desbloquea una zona situada al final de una región. La región debe descomponerse en dos partes: la primera representa la parte cuyo estado no cambia, y la segunda representa la parte cuyo estado se modifica. Se asigna un descriptor para la nueva región y se inicializa. El contenido del descriptor de la región original también se modifica, para indicar que la región termina antes de la región creada. Finalmente, se inserta el descriptor de la nueva región en el espacio de direccionamiento del proceso llamando a `insert_vm_struct`.

La función `mlock_fixup_middle` bloquea o desbloquea una zona situada en medio de una región. La región debe descomponerse en tres partes: la primera representa la zona cuyo estado no cambia, la segunda representa la zona cuyo estado se modifica, y la tercera representa el fin de la región original, cuyo estado no cambia. Se asignan e inicializan dos descriptors de región. El descriptor de la región original también se modifica, y los nuevos descriptors se insertan en el espacio de direccionamiento del proceso llamando a `insert_vm_struct`.

La función `mlock_fixup` modifica el bloqueo de una parte o de la totalidad de una región de memoria. Si el estado de la región no cambia, termina inmediatamente su ejecución. Si no, llama a una de las funciones precedentes según la ubicación de la zona a bloquear o desbloquear en la región.

La función `do_mlock` se llama para modificar el bloqueo de una parte del espacio de direccionamiento de un proceso. Primero verifica que el proceso que llama posee los privilegios necesarios, y luego comprueba la validez de sus parámetros. Seguidamente, efectúa un bucle de exploración de las regiones de memoria comprendidas en el intervalo especificado. Para cada región se llama a la función `mlock_fixup` a fin de modificar el bloqueo de una parte o de la totalidad de la región. Tras este bucle, se llama a la función `merge_segments` a fin de fusionar las regiones adyacentes si el cambio del estado de las diferentes regiones las ha hecho compatibles.

Las funciones `sys_mlock` y `sys_munlock` implementan las primitivas `mlock` y `munlock` llamando a `do_mlock`.

La función `do_mlockall` modifica el bloqueo del espacio de direccionamiento completo de un proceso. Primero verifica que el proceso que llama posee los privilegios necesarios, y luego efectúa un bucle de exploración de todas las regiones de memoria situadas en el espacio de direccionamiento del proceso. Por cada región, se llama a la función `mlock_fixup` a fin de modificar el bloqueo de la región. Tras este bucle, se llama a la función `merge_segments` a fin de fusionar las regiones adyacentes si el cambio del estado de las diferentes regiones las ha hecho compatibles.

Las funciones `sys_mlockall` y `sys_munlockall` implementan las primitivas `mlockall` y `munlockall` llamando a `do_mlockall`.

6.6.3 Modificaciones de protecciones de memoria

Las llamadas al sistema que modifican la protección asociada a regiones de memoria se implementan en el archivo fuente `mm/mprotect.c`.

Varias funciones modifican las protecciones asociadas a páginas de memoria:

- `change_pte_range`: esta función modifica las protecciones de memoria contenidas en una serie de entradas de la tabla de páginas.
- `change_pmd_range`: esta función modifica las protecciones de memoria contenidas en las entradas de la tabla de páginas correspondientes a un intervalo. Se llama a la función `change_pte_range` para efectuar las modificaciones.
- `change_protection`: esta función modifica las protecciones de memoria de las páginas correspondientes a una región de memoria, llamando a `change_pmd_range`.

Las protecciones asociadas a las regiones de memoria se modifican por cuatro funciones, cuyo funcionamiento es similar a las funciones de modificación del bloqueo:

- `mprotect_fixup_all`: modificación de la protección asociada a la totalidad de una región de memoria;
- `mprotect_fixup_start`: modificación de la protección asociada a una zona situada al principio de una región de memoria;
- `mprotect_fixup_end`: modificación de la protección asociada a una zona situada al fin de una región de memoria;
- `mprotect_fixup_middle`: modificación de la protección asociada a una zona situada en medio de una región de memoria.

La función `mlock_fixup` llama a una de las cuatro funciones anteriores según la ubicación de la zona a modificar en el interior de una región de memoria. Seguidamente llama a `change_protection` para modificar las protecciones asociadas a las páginas de memoria correspondientes.

La función `sys_mprotect` implementa la primitiva `mprotect`. Verifica la validez de sus parámetros, y efectúa un bucle sobre las regiones de memoria comprendidas en el intervalo especificado. Para cada región de memoria, `mlock_fixup` se llama para modificar las protecciones. Tras este bucle, se llama a la función `merge_segments` a fin de fusionar las regiones adyacentes si el cambio del estado de las diferentes regiones las ha hecho compatibles.

6.6.4 Reasignar regiones de memoria

El archivo fuente `mm/mremap.c` contiene las funciones que implementan la reasignación de zonas de memoria. Estas reasignaciones son provocadas por la primitiva `mremap`.

Se definen varias funciones de gestión de la tabla de páginas:

- `get_one_pte`: devuelve la dirección de la entrada de la tabla de páginas correspondiente a una dirección especificada;
- `alloc_one_pte`: asigna una entrada, correspondiente a una dirección especificada, en la tabla de páginas;
- `copy_one_pte`: copia el contenido de una entrada de la tabla de páginas en otra entrada;
- `move_one_page`: llama a `get_one_pte` y a `copy_one_pte` para copiar el contenido de una entrada de la tabla de páginas;
- `move_page_tables`: llama a `move_one_page` para cada página comprendida en el intervalo especificado.

La función `move_vma` reasigna una región de memoria. Asigna un nuevo descriptor de memoria, obtiene una dirección inutilizada llamando a `get_unmapped_area`, desplaza el contenido de la tabla de páginas llamando a `move_page_tables`. El nuevo descriptor se inicializa seguidamente, y se inserta en el espacio de direccionamiento del proceso por una llamada a `insert_vm_struct`, y luego se llama a la función `merge_segments` para fusionar las regiones de memoria adyacentes. Finalmente, se libera la antigua región de memoria llamando a `do_munmap`.

La función `sys_mremap` implementa la llamada al sistema `mremap`. Si el nuevo tamaño especificado es inferior al anterior, `sys_mremap` se limita a llamar a `do_munmap` para liberar la memoria superflua. Si no es así, llama a `find_vma` para obtener el descriptor de la región de memoria afectada. Modifica la dirección de fin de la región si es posible, es decir, si ninguna región está situada en el espacio necesario. Si el cambio de tamaño no es posible y se especifica la opción `MREMAP_MAYMOVE`, se llama a la función `move_vma` para desplazar la región de memoria.

6.7 Creación y supresión de espacio de direccionamiento

Al crearse un proceso por la llamada al sistema `fork`, el espacio de direccionamiento del proceso padre se duplica. Al terminar un proceso, su espacio de direccionamiento se libera.

Las funciones `dup_mmap` y `copy_mm`, contenidas en el archivo fuente `kernel/fork.c` (véase el capítulo 4, sección 6.2.1), duplican el espacio de direccionamiento del proceso padre. La función `dup_mmap` explora la lista de descriptors de regiones de memoria asociadas al proceso. Se asigna un nuevo descriptor por cada región, su contenido se inicializa, y las tablas de páginas se copian llamando a `copy_page_range`. Al final de este bucle, se construye el AVL que contiene los descriptors de regiones llamando a `build_mmap_avl`. La función `copy_mm` duplica el descriptor del espacio de direccionamiento del proceso padre, si el proceso hijo no es un clon. Seguidamente, crea la tabla de páginas del proceso hijo llamando a `new_page_tables`, y luego llama a `dup_mmap` para copiar las regiones de memoria.

La función `exit_mmap`, definida en el archivo fuente `mm/mmap.c`, se llama para liberar las regiones de memoria contenidas en el espacio de direccionamiento de un proceso, cuando éste termina. Explora la lista de descriptors de región, llama a las operaciones de memoria `unmap` y `close` asociadas a cada región, suprime las tablas de páginas correspondientes llamando a `zap_page_range`, y libera el descriptor de la región.

La función `__exit_mm`, contenida en el archivo fuente `kernel/exit.c` (véase el capítulo 4, sección 6.2.2), se llama para liberar el espacio de direccionamiento de un proceso. Primero decrementa el número de referencias del descriptor de espacio de direccionamiento. Si este número llega a nulo, llama a `exit_mmap` para suprimir las regiones de memoria asociadas al proceso, libera las tablas de páginas llamando a `free_page_tables`, y libera el descriptor de espacio de direccionamiento.

6.8 Proyección de archivos en memoria

El archivo fuente *mm/filemap.c* contiene las rutinas que gestionan el caché de páginas y los archivos proyectados en memoria.

Los descriptores de páginas asociadas a los archivos proyectados en memoria se registran en varias listas de hash. La tabla *page_hash_table* contiene la dirección del primer elemento de cada lista.

Varias funciones, definidas en el archivo de cabecera *<linux/pagemap.h>*, permiten gestionar estas listas:

- *_page_hashfn*: esta función implementa la tabla de hash, basada en un descriptor de i-nodo y un desplazamiento en el archivo.
- *find_page*: esta función efectúa la búsqueda de una página en las listas de hash y devuelve su descriptor. Posiciona el indicador *PG_referenced* en el descriptor.
- *remove_page_from_hash_queue*: esta función suprime un descriptor de página de su lista de hash.
- *add_page_to_hash_queue*: esta función añade un descriptor de página en las listas de hash. Posiciona el indicador *PG_referenced* en el descriptor.
- *remove_page_from_inode_queue*: esta función suprime un descriptor de página de la lista de páginas vinculadas a un i-nodo.
- *add_page_to_inode_queue*: esta función añade un descriptor de página en la lista de páginas vinculadas a un i-nodo.
- *wait_on_page*: esta función suspende el proceso actual en espera de una página si el descriptor de ésta está bloqueado.

La función *release_page* decrementa el número de referencias a una página. Este número debe ser superior a 1 en la llamada. Se trata de una versión rápida de *__free_pages*.

La función *invalidate_inode_pages* invalida todas las páginas asociadas a un i-nodo. Efectúa un bucle sobre la lista de páginas asociadas al i-nodo, suprime cada página de las listas llamando a *remove_page_from_hash_queue*, y llama a *__free_page* para liberar cada página.

La función *truncate_inode_pages* suprime todas las páginas asociadas a un i-nodo cuya dirección en el i-nodo se sitúa más allá de un tamaño especificado. Efectúa

un bucle sobre la lista de páginas asociadas al i-nodo, y llama a `remove_page_from_hash_queue` y a `__free_page` para cada página afectada. Se llama en la ejecución de la llamada al sistema *truncate*.

La función `shrink_mmap` se llama para liberar páginas de memoria. Explora de manera circular las páginas de memoria, considerando las páginas asociadas a un i-nodo o que contienen memorias intermedias. Comprueba el número de referencias de cada una de las páginas correspondientes. Si este número es igual a 1, comprueba si la página se ha referenciado recientemente. Si es así, `shrink_mmap` intenta liberar la página:

- si la página se asocia a un i-nodo, se suprime del caché llamando a la función `remove_page_from_hash_queue` y a `remove_page_from_inode_queue`, y la página se libera;
- si la página contiene memorias intermedias, se llama a la función `try_to_free_buffer` para intentar liberar las memorias intermedias contenidas en la página y la propia página.

La función `page_unuse` se llama para intentar liberar una página de memoria. Si el número de referencias a la página es de 2 y si la página está asociada a un i-nodo, se suprime de las listas llamando sucesivamente a las funciones `remove_page_from_hash_queue` y `remove_page_from_inode_queue`.

La función `update_vm_cache` actualiza el contenido de las páginas de memoria asociadas a un i-nodo cuando el contenido del archivo se modifica llamando a la primitiva *write*. Ésta explora la lista de páginas asociadas al i-nodo. Comprueba si el contenido de cada página forma parte de la sección que ha sido modificada por la escritura. Si es así, el contenido de la página se actualiza.

La función `add_to_page_cache` añade un descriptor en el caché de páginas llamando a `add_page_to_hash_queue` y a `add_page_to_inode_queue`.

La función `try_to_read_ahead` efectúa una lectura anticipada de página. Asigna una página por una llamada a `__get_free_page` si es necesario. Seguidamente llama a `find_page` para buscar la página en el caché. Si la página no se encuentra en el caché, se lanza una lectura de página llamando a la operación sobre i-nodo `readpage`.

La función `generic_file_read` implementa la operación de lectura de datos desde un archivo, utilizando el caché de páginas para leer los datos en memoria. Efectúa un bucle llamando a `find_page` para buscar la página correspondiente a los datos a leer. Si la página no se encuentra en el caché, se asigna una página llamando a `__get_free_page`, esta página se añade al caché llamando a `add_to_pa-`

`ge_cache`, y su contenido se lee desde el disco llamando a la operación `readpage` asociada al i-nodo. Una vez leída la página desde el disco, o si la página se ha encontrado en el caché, su contenido se copia en la memoria intermedia proporcionada por el proceso que llama.

La función `filemap_nopage` implementa la operación de memoria `nopage` para los archivos proyectados en memoria. Primero efectúa una búsqueda de la página especificada en el caché llamando a `find_page`. Si la página no se encuentra en el caché, se asigna una página llamando a `__get_free_page`, esta página se añade al caché llamando a `add_to_page_cache`, y su contenido se lee desde el disco llamando a la operación `readpage` asociada al i-nodo. Una vez leída la página desde el disco, o si la página se ha encontrado en el caché, `filemap_nopage` comprueba si la página puede ser compartida entre varios procesos. Si no es así (por ejemplo por una proyección privada), la página se duplica: se asigna una nueva página y su contenido se inicializa a partir del contenido de la primera página.

La función `do_write_page` reescribe en disco el contenido de una página asociada a un archivo proyectado en memoria. Para ello llama a la operación `write` asociada al archivo especificado.

La función `filemap_write_page` escribe el contenido de una página en un archivo. Crea un descriptor de archivo abierto temporal, y llama a `do_write_page` para efectuar la escritura.

La función `filemap_swapout` implementa la operación `swapout` asociada a los archivos proyectados en memoria. Llama a `filemap_write_page` para escribir el contenido de la página y pone a 0 la entrada de la tabla de páginas correspondiente llamando a `pte_clear`.

La función `filemap_swapin` sólo puede llamarse durante la escritura de una página, ya que, si la página ha sido descartada de la memoria, su entrada en la tabla de páginas es nula y se llamará a la operación `readpage`. Si, por el contrario, la página se está guardando mediante `filemap_swapout`, su entrada no es nula e indica que la página no está presente. En este caso, `filemap_swapin` reinicia el contenido de la entrada correspondiente a la página.

La reescritura de páginas modificadas se efectúa mediante varias funciones:

- `filemap_sync_pte`: esta función reescribe una página en disco llamando a la función `filemap_write_page`.
- `filemap_sync_pte_range`: esta función reescribe una serie de páginas en disco llamando a `filemap_sync_pte` por cada página.

- `filemap_sync_pmd_range`: esta función reescribe una serie de páginas en disco llamando a `filemap_sync_pte_range`.
- `filemap_sync`: esta función implementa la operación de memoria `sync` asociada a los archivos proyectados en memoria. Utiliza `filemap_sync_pmd_range` para reescribir en disco todas las páginas contenidas en el intervalo especificado.

La función `filemap_unmap` implementa la operación de memoria `unmap` asociada a los archivos proyectados en memoria. Se limita a llamar a `filemap_sync` para reescribir en disco las páginas modificadas.

La función `generic_file_map` implementa la operación sobre archivo `mmap` asociada a los archivos. Verifica que puede efectuarse la proyección en memoria, e inicializa el campo `vm_inode` del descriptor de región de memoria. El número de referencias del i-nodo se incrementa, y el puntero `vm_ops` se inicializa con la dirección de la variable `file_private_mmap`, en el caso de una proyección privada, o de `file_shared_mmap`, en el caso de una proyección compartida.

Finalmente, el archivo fuente `mm/filemap.c` contiene la implementación de la llamada al sistema `msync`. La función `msync_interval` llama a la operación de memoria `sync` asociada a una región, y la función `file_sync` para forzar la reescritura en disco si se especifica la opción `MS_SYNC`. La función `sys_msync` controla la validez de sus parámetros, y llama a `msync_interval` para cada región de memoria situada en el intervalo especificado.

6.9 Gestión del *swap*

6.9.1 Gestión de los dispositivos de *swap*

El archivo fuente `mm/swapfile.c` contiene las funciones que gestionan los archivos o dispositivos utilizados para guardar las páginas descargadas de la memoria.

La tabla `swap_info` contiene las características de los dispositivos de *swap* activos. La variable `swap_list` contiene el índice del dispositivo al que está asociada la prioridad más importante.

La función `scan_swap_map` busca una página disponible en el dispositivo especificado. Explora la tabla de bytes que indican el estado de las páginas y devuelve el número de una página disponible.

La función `get_swap_page` se llama para asignar una página en un dispositivo de *swap*. Explora la lista de dispositivos disponibles y llama a `scan_swap_map` para cada dispositivo. Cuando se encuentra una página, se devuelve su dirección de *swap*.

La función `swap_free` se llama para liberar una página en un dispositivo de *swap*. Decrementa el byte que contiene el número de referencias a la página.

Las funciones `unuse_pte`, `unuse_pmd` y `unuse_pg` se llaman al desactivarse un dispositivo de *swap*. Exploran las tablas de páginas y recargan en memoria todas las páginas presentes en el dispositivo especificado. La función `unuse_vma` utiliza `unuse_pg` para recargar todas las páginas correspondientes a una región de memoria.

La función `unuse_process` explora la lista de regiones de memoria contenidas en el espacio de direccionamiento de un proceso. Llama a `unuse_vma` por cada región.

La función `try_to_unuse` se llama para recargar en memoria todas las páginas presentes en el dispositivo especificado. Explora la tabla de procesos y llama a la función `unuse_process` para recargar el espacio de direccionamiento de cada proceso.

La función `sys_swapoff` implementa la primitiva *swapoff* que desactiva un dispositivo de *swap*. Primero verifica que el proceso que llama posee los privilegios necesarios, y busca el descriptor del dispositivo en la tabla `swap_info`, y lo suprime de la lista. Seguidamente llama a `try_to_unuse` para recargar en memoria todas las páginas guardadas en ese dispositivo. Si esto falla, el descriptor del dispositivo se inserta de nuevo en la lista y se devuelve un error. En caso de éxito, se llama a la operación de archivo *release* asociada al dispositivo, y el descriptor se libera.

La función `sys_swapon` implementa la primitiva *swapon* que activa un dispositivo de *swap*. Primero verifica que el proceso que llama posee los privilegios necesarios, luego busca un descriptor libre en la tabla `swap_info`, e inicializa dicho descriptor. Seguidamente, se asigna una página de memoria, el catálogo del dispositivo de *swap* se lee en esta página, y se verifica la firma del dispositivo de *swap*. Tras esta verificación, se efectúa un bucle a fin de contar las páginas disponibles en el dispositivo, y la tabla de bytes que indica el estado de cada página se asigna e inicializa. Para terminar, el descriptor del dispositivo se inserta en la lista.

6.9.2 Entrada/salida de páginas de *swap*

El archivo fuente `mm/page_io.c` contiene las funciones de entrada/salida de páginas sobre dispositivos de *swap*.

La función `rw_swap_page` lee o escribe una página en un dispositivo de *swap*. Verifica que el dispositivo especificado está activo, y que el número de página es válido.

do, y a continuación bloquea la página en cuestión en el descriptor del dispositivo de *swap*. Seguidamente, comprueba el tipo del dispositivo de *swap*:

- Si se trata de un dispositivo en modo bloque, se llama a la función `ll_rw_page` para leer o escribir el contenido de la página.
- Si se trata de un archivo de *swap*, se llama a la operación `bmap` asociada al i-nodo del archivo para determinar la dirección de los bloques que componen la página, y se llama a `ll_rw_swap_file` para leer o escribir los bloques. Esta función, definida en el archivo fuente `drivers/block/ll_rw_blk.c`, genera peticiones de entradas/salidas para los bloques especificados.

La función `swap_after_unlock_page` se llama cuando la lectura o la escritura de una página en un dispositivo de *swap* ha terminado, y desbloquea la página en cuestión en el descriptor del dispositivo de *swap*.

La función `ll_rw_page` se utiliza para leer o escribir una página en un dispositivo en modo bloque. Bloquea la página en memoria activando el indicador `PG_locked`, y llama a la función `brw_page` para efectuar la entrada/salida.

Las funciones `read_swap_page` y `write_swap_page`, declaradas en el archivo de cabecera `<linux/swap.h>`, llaman a `rw_swap_page` para efectuar una lectura o una escritura de página.

La función `swap_in`, definida en el archivo fuente `mm/page_alloc.c`, se llama para cargar una página en memoria para un proceso. Asigna una página de memoria, y llama a `read_swap_page` para cargar su contenido. Seguidamente, la dirección de la página se registra en la entrada de la tabla de páginas afectada.

6.9.3 Eliminación de páginas de memoria

El archivo fuente `mm/vmscan.c` contiene las funciones que deciden la eliminación de páginas de memoria y su escritura en un dispositivo de *swap*.

La función `try_to_swap_out` se llama para intentar eliminar una página específica. Si la página está reservada o bloqueada, o si ha sido accedida recientemente, no se elimina de la memoria. En caso contrario, se comprueba su estado. Si la página ha sido modificada, debe guardarse en disco: si hay asociada una operación de memoria `swpout` a la región que contiene la página, se llama; si no, se asigna una entrada de *swap* llamando a `get_swap_page`, y se llama a la función `write_swap_page` para escribir la página. En el caso en que la página no haya sido modificada, se llama a la

función `page_unuse` para suprimirla del caché de páginas, y la página se libera por `free_page`.

Las funciones `swap_out_pmd` y `swap_out_pgd` exploran la tabla de páginas de un proceso intentando eliminar cada página. Estas funciones paran su ejecución en cuanto una página ha sido eliminada. La función `swap_out_vma` llama a `swap_out_pgd` para intentar eliminar una página en una región de memoria, parando su ejecución cuando una página ha sido eliminada.

La función `swap_out_process` se llama para eliminar una página del espacio de direccionamiento de un proceso. Explora la lista de regiones de memoria contenidas en el espacio de direccionamiento del proceso, llama a `swap_out_vma` para cada región y para su ejecución en cuanto se elimina una página.

La función `swap_out` se llama para eliminar una página que forme parte del espacio de direccionamiento de uno de los procesos existentes. Explora la tabla de procesos y llama a la función `swap_out_process` para cada proceso que posee páginas residentes en memoria. Cuando una página ha podido ser eliminada o cuando ninguna página puede ser eliminada de la memoria, para su ejecución.

La función `try_to_free_page` se llama para intentar liberar una página de memoria cualquiera. Se llama sucesivamente a `shrink_mmap`, `shm_swap` y `swap_out` para intentar liberar páginas de memoria.

La función `kswapd` implementa el proceso *kswap* que elimina páginas de la memoria en segundo plano. Este proceso efectúa un bucle infinito durante el cual se suspende por una llamada a `interruptible_sleep_on`, y seguidamente intenta liberar páginas de memoria llamando a `try_to_free_page` cuando se despierta.

La función `swap_tick` se llama en cada ciclo de reloj, y despierta el proceso *kswapd* para liberar memoria, si el número de páginas disponibles es inferior al umbral tolerado.

Capítulo 9

Terminales POSIX

1 Conceptos básicos

1.1 Presentación general

El término de *terminales* designa entidades que van de la consola de la máquina como la que usa cualquier usuario hasta la ventana `xterm`, pasando por el enlace serie con el que una persona puede conectarse una máquina por módem. Estos terminales pueden compararse a una pasarela entre el usuario y la aplicación. La complejidad de su uso proviene de la diversidad de dispositivos con los que son capaces de comunicar. Sin embargo constituyen también una prueba de potencia.

En todo sistema Unix, los terminales se representan en forma de archivos especiales llamados de *modo carácter* (véase el capítulo 7, sección 1, para más detalles sobre este tipo de dispositivos) en los que se puede leer, cuando por ejemplo un usuario escribe por teclado, y escribir, cuando una aplicación envía datos a un módem en un puerto serie.

Generalizando, se puede considerar que existen en Linux los cinco tipos de terminales siguientes (también se describirán los archivos periféricos asociados):

- las consolas virtuales que se utilizan principalmente cuando el usuario se conecta físicamente a la máquina: `tty1` → `tty63`;
- los pseudoterminales maestros: `ptyp0` → `ptysf`;

- los pseudoterminales esclavos que se utilizan en una conexión remota o en una ventana X-Window: `ttyp0` → `ttysf`;
- los puertos serie utilizados por los módems o el mouse: `ttys0` → `ttys63` y `cua0` → `cua63`. La existencia para cada puerto de dos terminales se explicará más adelante;
- los terminales particulares: la consola, las tarjetas serie específicas (existen ciertas tarjetas que permiten tener más de dos puertos serie generalmente presentes en un PC; en el caso en que la máquina posea una de estas tarjetas, se le asignan terminales particulares), ciertos mouses particulares como los modelos PS/2...

Linux posee ciertas especificidades a nivel de la gestión de terminales. Se puede considerar que la gestión de terminales virtuales y de pseudoterminales es bastante parecida a la de otros sistemas Unix, pero la gestión de los puertos serie es particular (dos dispositivos por puerto). La primera categoría `ttysx` se utiliza para las llamadas entrantes. Por ejemplo, es posible conectarse a ciertas máquinas por módem y, en este caso, se utilizará un archivo `ttysx` para «responder» a la conexión. Los archivos `cuax` se utilizan en la máquina sobre la que el usuario se encuentra para controlar el módem.

Cuando un proceso abre un archivo especial `ttysx`, se suspende hasta que el dispositivo correspondiente esté a punto. Un programa de establecimiento de conexión (*getty*), en un puerto serie, se suspende pues hasta que se produzca una conexión y el módem conectado active la señal DTR (*Data Terminal Ready*). Por el contrario, no se provoca una suspensión si un proceso abre un archivo especial `cuax`. Entonces estos puertos se usarán para efectuar las llamadas «salientes».

El principal problema que ha desesperado a más de un desarrollador es la configuración de estos terminales porque debido a su variedad el código es poco portable entre las diferentes versiones Unix existentes. La norma POSIX.1 ha uniformado las operaciones de manipulación basándose principalmente en las funcionalidades existentes en System V (véase [AT&T 1989], y especialmente [Lewine 1991]), añadiendo primitivas de alto nivel que evitan así el uso masivo de la llamada *ioctl*.

Si, bajo el término de *Terminales Posix* se agrupan los mecanismos de la gestión «física» de los puertos serie, esta norma ha presentado de manera clara las nociones de sesiones y de grupos de procesos. Estas nociones permiten entre otras cosas distinguir a nivel del sistema operativo los procesos que se ejecutan en segundo plano y los que se ejecutan en primer plano. Esta diferenciación permite así identificar que un proceso pertenezca a la sesión de un usuario, o bien si se considera a sí mismo como el *líder* de la sesión.

1.2 Configuración de un terminal

1.2.1 Modo canónico - modo no canónico

Antes de presentar el detalle de la configuración de un terminal es necesario presentar los dos modos de uso de un terminal: el modo *canónico* y el modo *no canónico*.

El modo canónico es el más simple: la entrada de un terminal se gestiona en forma de líneas. Una línea se delimita por un carácter de nueva línea (LF), un carácter de fin de archivo (EOF) o por un carácter de fin de línea (EOL). Los diferentes caracteres se presentan en la sección 1.3.

Esto significa que un programa que intenta leer una línea en un terminal debe esperar que una línea completa haya sido introducida antes de poder tratarla. El número de caracteres enviados importa poco: lo que importa es que se envíe una línea completa. Sin embargo, es posible leer sólo un número finito de caracteres (sin ninguna pérdida de información). La lectura también puede pararse si se recibe una señal. En este modo, se efectúa normalmente el sistema de supresión de caracteres o de líneas (los caracteres ERASE, WERASE, KILL se detallarán ulteriormente).

En el modo no canónico, los caracteres en entrada no se tratan en forma de línea. Los valores MIN y TIME se utilizan para determinar la manera como se reciben los caracteres. MIN y TIME sólo son utilizables cuando la línea no está configurada en modo canónico. Permiten determinar cómo un proceso debe tratar los caracteres recibidos. Su comportamiento es bastante complejo. MIN corresponde al número mínimo de caracteres que deben recibirse antes de que la lectura sea satisfecha. TIME corresponde a un *timer* en décimas de segundo que se utiliza para hacer accesibles los datos tras un cierto lapso de tiempo. Estos dos caracteres están relacionados. Existen cuatro comportamientos particulares:

1. MIN > 0 y TIME > 0: TIME se emplea como *timer* entre cada carácter. Se activa tras la recepción del primer carácter y se reinicia tras cada nueva recepción. Si MIN caracteres se reciben antes de la expiración de la duración TIME, entonces se satisface la lectura. En caso contrario, los caracteres recibidos se devuelven al usuario. Si el número de caracteres leídos es inferior al número de caracteres disponibles, entonces el *timer* no se reactiva y la lectura siguiente se satisface inmediatamente.
2. MIN > 0, TIME = 0: en este caso, sólo se usa MIN. Una lectura sólo se satisface cuando se han recibido MIN caracteres.

3. `MIN = 0, TIME > 0`: como `MIN = 0`, `TIME` se utiliza como un *timer* de lectura que se activa cuando se efectúa una lectura. Una lectura se satisface cuando se recibe un solo carácter, a menos que transcurra la duración. En este caso, no se devuelve ningún carácter. Por tanto, si no se ha recibido ningún carácter en `TIME * 0.10` segundos, la lectura fracasa.
4. `MIN = 0, TIME = 0`: el retorno es inmediato. Se devuelve el número mínimo de caracteres solicitados (o bien los disponibles).

1.2.2 La estructura `termios`

La configuración de un terminal POSIX realmente no es fácil, debido al gran número de variantes que pueden existir. Esto abarca desde la velocidad de la comunicación hasta el tipo de codificación utilizado, etc.

Los archivos de cabecera que permiten utilizar las funcionalidades de los terminales POSIX son bastante numerosos. Sin embargo, la mayoría de las primitivas, estructuras y constantes utilizadas se encuentran en el archivo de cabecera `<termios.h>`. Este último incluye los archivos `<linux/termios.h>`, `<asm/termios.h>` y `<asm/termbits.h>`, que efectúan las definiciones propiamente dichas.

La configuración de un terminal se efectúa a través de la estructura `termios`. Ésta permite tanto examinar la configuración del terminal como cambiar sus parámetros. Veámosla en detalle:

tipo	campo	descripción
<code>tcflag_t</code>	<code>c_iflag</code>	Modos de entrada
<code>tcflag_t</code>	<code>c_oflag</code>	Modos de salida
<code>tcflag_t</code>	<code>c_cflag</code>	Modos de control
<code>tcflag_t</code>	<code>c_lflag</code>	Modos locales
<code>cc_t</code>	<code>c_line</code>	Disciplina de línea (presente sólo por razones de compatibilidad con la antigua estructura <code>termio</code>)
<code>cc_t[NCCS]</code>	<code>c_cc</code>	Caracteres de control

Los tipos de los campos de esta estructura son en realidad `unsigned int` para `tcflag_t` y `unsigned char` para `cc_t`. Los «verdaderos» tipos de estos campos sólo se dan a título indicativo y no deben utilizarse en absoluto en un código fuente, porque pueden depender de la implementación.

1.2.3 Funcionamiento de una comunicación

Antes de seguir detallando los diferentes valores que pueden asignarse a los campos de esta estructura, es necesario comprender cómo se establece y funciona la comunicación con un terminal.

El intercambio de datos entre un proceso y un terminal POSIX se efectúa, como se muestra en la figura 9.1, mediante dos memorias intermedias. Estas memorias permiten (a menos que el terminal haya configurado de otro modo) acelerar las transferencias entre el proceso y el terminal. Cuando se transfiere un carácter de la memoria al terminal, sufre una transformación en función de las características del terminal, y al revés cuando el carácter proviene del terminal, ya que, para poder dialogar correctamente con los parámetros concretos (velocidad, codificación en 7 u 8 bits, bit de control, paridad...), resulta necesario adaptarse al protocolo de comunicación del terminal. De este modo, la escritura y la lectura son transparentes porque la operación de conversión se encapsula y no es visible por el desarrollador, excepto en la inicialización del terminal. Esto facilita en gran medida la comunicación.

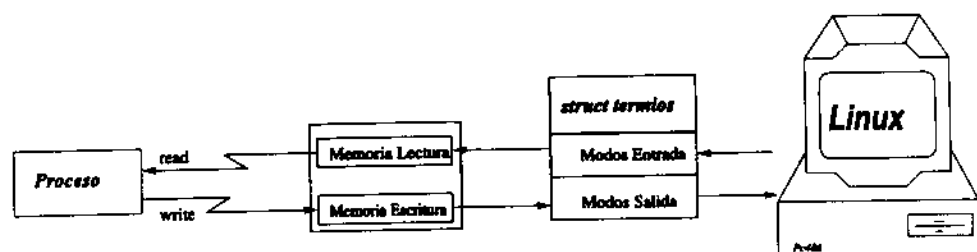


FIG. 9.1 – Comunicación entre proceso y terminal

1.2.4 Configuración de los modos de entrada

Los modos de entrada, que corresponden al campo `c_iflag` de la estructura `terminios`, permiten definir los tratamientos (conversiones, formato...) que es necesario efectuar para hacer transitar un carácter desde el terminal hacia la memoria intermedia.

Para configurar este campo, el desarrollador dispone de un cierto número de opciones declaradas en el archivo de cabecera `<linux/tty.h>`. En la tabla siguiente se presentan todas las opciones que es posible utilizar para configurar la línea serie. Sin embargo, algunas se especifican en la norma POSIX.1, mientras que otras son extensiones (SVR4, BSD...). En el caso en que no se trate de la norma POSIX.1, la celda correspondiente de la tabla se dejará en blanco.

macro-instrucción	significado	norma
BRKINT	Si IGNBRK no se ha fijado, la recepción de un carácter BREAK provoca la emisión de la señal SIGINT y las memorias de entrada y salida se vacían. En caso contrario, la lectura de un carácter BREAK equivale a un carácter \0 .	POSIX.1
ICRNL	Si IGNCR no está activado, un carácter CR recibido se convierte en un carácter NL .	POSIX.1
IGNBRK	Los caracteres BREAK no se tienen en cuenta, simplemente se ignoran	POSIX.1
IGNCR	Ignora el carácter CR .	POSIX.1
IGNPAR	Ignora los errores de paridad y de ventanas.	POSIX.1
IMAXBEL	Provoca un <i>bip</i> cuando la memoria de entrada está llena	
INLCR	Convierte un carácter NL en CR .	POSIX.1
INPCK	Activa la verificación de la paridad. En el caso en que esta opción no esté activa, la verificación de error no se efectúa.	POSIX.1
ISTRIP	Suprime el octavo bit.	POSIX.1
IUCLC	Convierte las letras mayúsculas en minúsculas.	
IXANY	Activa todos los caracteres.	
IXOFF	Activa el flujo de control XON/XOFF en la entrada: los caracteres START y STOP los envía automáticamente el sistema para evitar pérdidas de datos.	POSIX.1
IXON	Activa el flujo de control XON/XOFF en la salida: si se recibe el carácter STOP , el flujo de datos se para, hasta que se envíe el carácter START .	POSIX.1
PARMRK	Marca los errores de paridad. Si IGNPAR no está activado, se envía un byte con un error de paridad o de ventana a la aplicación en forma de tres caracteres sucesivos, \377 \0 y el carácter erróneo. En el caso en que IGNPAR y PARMRK no estén fijados, la lectura de un carácter con error de paridad o de ventana equivale a la lectura de un carácter \0 .	POSIX.1

1.2.5 Configuración de los modos de salida

Los modos de salida que corresponden al campo `c_oflag` de la estructura `termios` permiten definir los tratamientos (conversiones, formato...) que es necesario efectuar para hacer transitar un carácter desde la memoria intermedia al terminal.

Para configurar este campo, el desarrollador dispone de varias opciones declaradas en el archivo de cabecera `<linux/tty.h>`:

macro-instrucción	significado	norma
BSOPLY	Espera tras un carácter backspace	
BSO, BS1	Máscaras para BSOPLY	
CRDLY	Máscara de temporización tras un retorno de carro	
CR0, CR1, CR2, CR3	Máscaras para CRDLY	

FFDLY	Espera tras el carácter FF
FF0, FF1	Máscaras para FFDLY
NLDLY	Máscara de temporización tras un carácter NL
NL0, NL1	Máscaras para NLDLY
OCRNL	Transforma los caracteres CR en NL
OFDEL	El carácter de relleno es el carácter ASCII DEL , si no es el carácter NUL
OFILL	Envía caracteres de relleno para temporizar
OLCUC	Transforma las minúsculas en mayúsculas
ONLCR	Transforma los caracteres NL en CR NL
ONLRET	El carácter NL se interpreta como un carácter CR
ONOCR	No muestra un carácter CR en la columna 0
OPOST	Activa el tratamiento de caracteres
TABDLY	Espera tras una tabulación
TAB0, TAB1	Máscaras para TABDLY . La máscara XTABS transforma una
TAB2, TAB3	tabulación en ocho espacios
XTABS	
VTILY	Espera tras una tabulación vertical
VT0, VT1	Máscaras para VTILY

POSIX.1

macro-instrucción	significado	norma
CBAUD B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, B38400	Velocidad de comunicación por la línea Máscara de velocidad	
CBAUDEX B57600, B115200, B230400, B450800	Extensión de declaración de la velocidad de comunicación del módem Máscara de velocidad	
CIBAUD	Máscara para la velocidad de entrada de los datos (no se usa)	POSIX.1
CLOCAL	Ignora el control de línea del módem	POSIX.1
CREAD	Activa el receptor	
CRTSCTS	Control de flujo	

CSIZE	Tamaño de la codificación de los caracteres	POSIX.1
CS5, CS6, CS7, CS8	Máscaras para CSTYPE	POSIX.1
CSTOPB	Dos bits de parada en lugar de uno solo	POSIX.1
HUPCL	Desconexión automática al terminar el último proceso bajo control	POSIX.1
PARITY	Paridad activada	POSIX.1
PARODD	Paridad impar. Si esta máscara no se indica, la paridad es par	POSIX.1

1.2.7 Configuración de los modos locales

Estos modos son cruciales porque permiten especificar cómo deben interpretarse los caracteres.

macro-instrucción	significado	norma
ECHO	Si esta máscara está activada, el texto escrito es visible	POSIX.1
ECHOCTL	Si TEXTEN está activada, todo carácter de control diferente de TAB, NL, START y STOP, CR y BS posee su valor ASCII más 100 en octal	POSIX.1
ECHOE	Si ICANON está activado, el carácter ERASE suprime el carácter anterior, y el carácter WERASE suprime la palabra anterior	POSIX.1
ECHOK	Si ICANON está activado, el carácter KILL suprime la línea actual	POSIX.1
ECHONL	Si ICANON está activado, se emite el carácter KILL para suprimir cada uno de los caracteres de la línea, como se ha especificado para ECHOE y ECHOPRT	POSIX.1
ECHONL	Si ICANON está activado, el carácter NL se emite incluso si ECHO no está activado	POSIX.1
ECHOPRT	Si ICANON e ECHO están activados, los caracteres se muestran a medida que se suprimen (lo cual sirve para ver los caracteres suprimidos)	POSIX.1
FLUSHO	Se vacía la memoria de salida. Esta opción se activa cuando se recibe el carácter DISCARD	POSIX.1
ICANON	Activa el modo canónico, es decir, un uso del terminal en modo interactivo. Esto activa ciertos caracteres especiales como EOF , BOL , BOL2 , ERASE , KILL , REPRINT , STATUS y WERASE . Finalmente, el paso a este modo particular hace que los caracteres provenientes del terminal se inserten en la memoria en forma de línea, terminada por el carácter NL o LF	POSIX.1
TEXTEN	Activa el sistema de tratamiento definido por la implementación del dispositivo	POSIX.1
ISIG	Cuando se reciben los caracteres INTR , QUIT , SUSP (o DSUSP), se emiten respectivamente las señales SIGINT , SIGQUIT y SIGTSP	POSIX.1
NOFLSH	Desactiva el sistema de vaciado de las memorias de entrada y salida cuando se reciben las señales SIGINT o SIGQUIT . Esto desactiva también el vaciado de la memoria de entrada cuando se recibe la señal SIGSUSP .	POSIX.1
PREPND	Todos los caracteres de la memoria de entrada se envían cuando se envía el carácter siguiente	POSIX.1
TOSTOP	Envía la señal SIGTTOU al grupo de procesos de un proceso en segundo plano que intenta escribir en su terminal de control	POSIX.1
ICASE	Si ICANON está activado, el terminal está en mayúsculas únicamente. Todo carácter recibido en la entrada del terminal se convierte a mayúsculas, salvo los caracteres precedidos por \ . En la salida, los caracteres en mayúsculas van precedidos de un \ y las letras minúsculas se convierten en mayúsculas.	POSIX.1

1.3 Los caracteres especiales y la tabla `c_cc`

En la descripción de los valores que pueden tomar los diferentes campos, se han utilizado ciertos caracteres particulares. La norma POSIX.1 define 11 caracteres que se gestionan de manera particular. Las otras normas añaden ciertos caracteres, que también son gestionados por Linux. Sin embargo, no es aconsejable su uso porque no es conforme a POSIX.

POSIX.1 permite también cambiar el valor de estos caracteres (salvo NL y CR). Para efectuar esta operación, basta con modificar la entrada deseada en la tabla `c_cc` de la estructura `termios`. En el ejemplo que sigue, se ha desactivado el carácter de control QUIT y se ha redefinido el carácter INTR. No es aconsejable utilizar este ejemplo en un terminal, ya que el comportamiento de la línea serie se verá alterado debido a estas modificaciones. Lo mejor es efectuar la prueba en una ventana *xterm* dedicada a esta prueba.

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

void main ()
{
    struct termios  term;

    /* recuperación de la configuración del terminal */
    if (tcgetattr (STDIN_FILENO, &term) == -1) {
        perror ("tcgetattr");
        exit (-1);
    }
    /* modificación de la composición de c_cc */
    term.c_cc[VINTR] = 65; /* cambio */
    term.c_cc[VQUIT] = _POSIX_VDISABLE; /* desactivación */

    /* actualización de la estructura */
    if (tcsetattr (STDIN_FILENO, TCSAFLUSH, &term) == -1) {
        perror ("tcsetattr");
        exit (-1);
    }
}
```

El ejemplo anterior ilustra la modificación de un carácter, pero sobre todo la anulación del comportamiento de otro. Esto se efectúa dándole el valor `_POSIX_VDISABLE`. Esta constante viene definida por la norma POSIX.1 y permite anular fácilmente el efecto de un carácter.

Para terminar esta parte y antes de tratar las funciones de acceso y manipulación de los terminales, veamos una tabla que enumera cada uno de estos caracteres y su efecto.

Carácter	significado	c_cc	activado por	Valor	Norma
CR	Retorno de carro		ICANON	\r	POSIX.1
DISCARD	No utilizar la salida	VDISCARD	IXOFF	^O	
DSUSP		VDSUSP	ISIG	^Y	
EOF	Fin de archivo: todos los caracteres escritos anteriormente son accesibles en lectura	VEOF	ICANON	^D	POSIX.1
EOL	Fin de línea	VEOL	ICANON		POSIX.1
EOL2	Otro carácter de fin de línea	VEOL2	ICANON		
ERASE	Carácter de borrar atrás	VERASE	ICANON	^H	POSIX.1
INTR	Señal de interrupción SIGINT	VINTR	ISIG	^? ^C	POSIX.1
KILL	Supresión de línea	VKILL	ICANON	^U	POSIX.1
LNEXT	Todo carácter siguiente se ignora	VLNEXT	IXOFF	^V	
MIN	Ver sección 1.2.1	VMIN		4	
NL	Retorno de carro		ICANON	\n	POSIX.1
QUIT	Señal SIGQUIT	VQUIT	ISIG	^\	POSIX.1
REPRINT	Vuelve a mostrar los caracteres	VREPRINT	ICANON	^R	
START	Vuelve a autorizar el envío de caracteres con destino al terminal	VSTART	IXON/IXOFF	^Q	POSIX.1
STATUS	Peticion de estatus	CSTATUS	ICANON	^T	
STOP	Suspende el envío de caracteres destinados al terminal	VSTOP	IXON/IXOFF	^S	POSIX.1
SUSP	Señal SIGSTP	VSUSP	ISIG	^Z	POSIX.1
TIME	Ver sección 1.2.1	VTIME		N.D.	
WERASE	Borra la última palabra escrita en la línea	VWERASE	ICANON	^W	

Estos caracteres se consideran como caracteres especiales porque su presencia en una comunicación influye en la transformación del código así recibido. Un cierto número de ellos no están definidos en la norma POSIX.1, como se indica en la tabla. Sin embargo, para facilitar la portabilidad, Linux los gestiona sin problemas.

1.4 El mandato stty

El mandato *stty* permite obtener todos los parámetros del terminal a partir del cual se conecta el usuario. En realidad, este mandato permite leer los parámetros de un terminal, pero también modificarlos.

Veamos un ejemplo de informaciones dadas:

```
gandalf# stty -a
speed 9600 baud; rws 24; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany -imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel n10 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echopr
echoctl echoke
```

1.5 El mandato setserial

El mandato *setserial* permite recuperar información sobre un puerto serie (que es un tipo particular de terminal).

Por ejemplo, veamos la información respecto al puerto serie utilizada por el mouse:

```
gandalf# setserial -a /dev/mouse
/dev/mouse, Line 0, UART: 16550A, Port: 0x03f8, IRQ: 4
    Baud_base: 115200, close_delay: 50, divisor: 0
    Flags: spd_normal skip_test
```

1.6 Grupos de terminales – sesión

Las nociones de grupos de procesos y de sesiones han sido introducidas en el capítulo 4, sección 1.7.

Un terminal de control distingue uno de los grupos de procesos en la sesión a la cual está asociado para formar el grupo de procesos en primer plano. Todos los demás grupos de procesos de la sesión se designan entonces como grupos de procesos en segundo plano. El grupo de procesos en primer plano juega un papel particular en la gestión de los caracteres que deben entrarse. De modo predeterminado, cuando se le asocia un terminal de control, el proceso puede leer y escribir datos en el terminal. Sólo hay un proceso de este tipo al mismo tiempo para un terminal dado.

Los grupos de procesos en segundo plano deben seguir la disciplina de la línea cuando intentan acceder a su terminal de control. Ello provoca la parada de un proceso que efectúa una lectura en un terminal porque éste está en segundo plano. En este caso, recibe la señal *SIGTTIN*. Cuando está activado el bit *TOSTOP* del terminal de control, los grupos de procesos que están en segundo plano y que intentan escribir en el terminal de control reciben la señal *SIGTTOU*, y por ello también se suspenden. Además, el terminal no le envía las señales *SIGINTR*, *SIGQUIT* y *SIGSUSP*.

1.7 Pseudoterminales

Un pseudoterminal es una capa lógica que permite establecer comunicación entre dos terminales. Un pseudoterminal se compone de dos terminales: un terminal amo y un terminal esclavo. Desde el punto de vista de dispositivo, el pseudoterminal esclavo se asocia a un *ty*, mientras que un pseudoterminal amo se asocia a un *pty*. La diferencia principal entre un pseudoterminal y un verdadero terminal es que la totalidad de lo que se escribe en el pseudoterminal amo es accesible en lectura por el terminal esclavo como se ilustra en la figura 9.2. Asimismo, todo lo que se escribe en el terminal esclavo es accesible en lectura en el terminal amo.

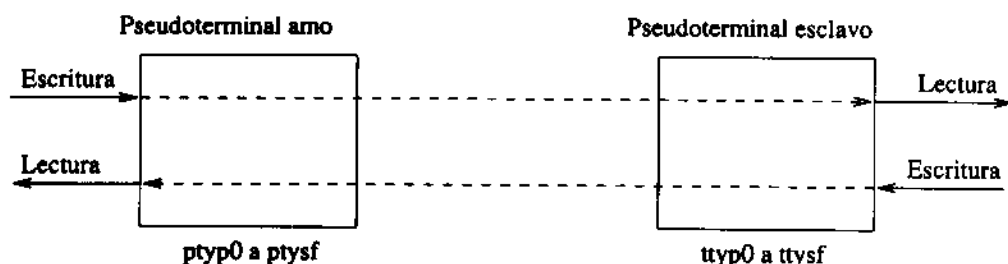


FIG. 9.2 – Lectura/escritura en un pseudoterminal

Los pseudoterminales son accesibles mediante los archivos `/dev/ptyp0` a `/dev/ptyrf` y los pseudoterminales esclavos son accesibles mediante los archivos `/dev/ttyp0` a `/dev/ttyrf`.

Los pseudoterminales tienen la ventaja de usarse exactamente como si se tratase de archivos normales, es decir, con las llamadas al sistema estándar como *read* y *write*. Sin embargo, un pseudoterminal amo sólo puede abrirse una vez porque dos procesos no pueden abrir al mismo tiempo el mismo pseudoterminal amo. En caso contrario, se devuelve un error `EBUSY`.

Los pseudoterminales son utilizados principalmente por los servidores de red y los sistemas de ventanas como X-Window. Por ejemplo, el principio del mandato *rlogin* consiste en abrir una conexión de red hacia un demonio *rlogind* que se ejecuta en otra máquina. Este demonio lanza un proceso que ejecuta el mandato *login*, y comunica con él mediante pseudoterminales:

- *rlogind* posee el pseudoterminal amo y escribe en él todos los paquetes recibidos desde la red; también lee los datos escritos por el proceso hijo y los transmite por la red;
- *rlogin* posee el pseudoterminal esclavo y le envía su visualización; también lee en él, por su entrada estándar, los datos transmitidos por *rlogind*.

La figura 9.3 ilustra este mecanismo.

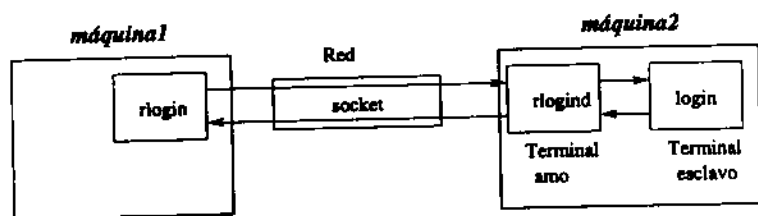


FIG 9.3 – Utilización de un pseudoterminal con rlogin

2 Las funciones POSIX

Antes que la norma POSIX.1 normalizara los terminales, la configuración se efectuaba mediante la llamada al sistema *ioctl*. Sin embargo, la legibilidad se reducía en gran medida, y el código era de realización particularmente complicada. Con la definición de las funciones POSIX.1, la configuración y el acceso a terminales es más simple... En la descripción de las funciones de manipulación de terminales, la utilización mediante la llamada al sistema *ioctl* también se dará, aunque éste sea el método *anti-guo* y sea preferible dejar de usarlo.

2.1 Acceder y modificar los atributos de un terminal

Los dos métodos que permiten recuperar la configuración de un terminal y, al revés, cambiar su configuración han sido utilizados ya en el ejemplo anterior (véase la sección 1.3).

```
#include <termios.h>
#include <unistd.h>

int tcgetattr (int fd, struct termios *termios_p);
int tcsetattr (int fd, int optional_actions, struct termios
               *termios_p);
```

La primera función permite recuperar la configuración de un terminal. Basta para ello con indicar el descriptor de archivo y un puntero a una estructura *termios* asignada. Esta función equivale a utilizar la petición *TCGETS*.

```
int tcgetattr (int fd, struct termios *termios_p)
{
    return ioctl (fd, TCGETS, termios_p);
}
```

La función `tcsetattr` permite efectuar la configuración del terminal deseado. El primer y tercer parámetros corresponden a lo mismo que en el caso de la función anterior. El segundo parámetro permite especificar la forma como se efectuará la configuración:

- **TCSANOW** (petición **TCSETS**): el cambio se efectúa inmediatamente.
- **TCSADRAIN** (petición **TCSETSW**): el cambio se efectúa después de haberse efectuado todas las escrituras en el descriptor de archivo. Es preferible utilizar esta opción cuando el desarrollador cambia los atributos de salida del terminal.
- **TCSAFLUSH** (petición **TCSETSF**): el cambio se efectúa después de haberse efectuado todas las escrituras en el descriptor de archivo. La memoria de lectura se vacía si quedan caracteres en ella.

La llamada a esta función equivale al código siguiente:

```
int tcsetattr (int fd, int optional_actions, struct termios
               *termios_p)
{
    switch (optional_actions) {
        case TCSANOW:
            return ioctl (fd, TCSETS, termios_p);
        case TCSADRAIN:
            return ioctl (fd, TCSETW, termios_p);
        case TCSAFLUSH:
            return ioctl (fd, TCSETSF, termios_p);
        default:
            errno = EINVAL;
            return -1;
    }
}
```

Es importante observar que, tras haber modificado el comportamiento de un terminal, todos los procesos con acceso a ese terminal adoptarán el nuevo comportamiento.

Los valores de retorno de estos dos métodos son 0 en caso de éxito. En caso de producirse un fallo, el valor de retorno es -1, y se activa la variable `errno`.

Los errores siguientes pueden generarse cuando se utilizan las funciones POSIX de manipulación de terminales:

<i>error</i>	<i>significado</i>
EINVAL	Parámetro incorrecto
ENOTTY	El dispositivo no es un terminal
ETO	Error de entrada/salida
ESPIPE	Operación de posicionamiento directo imposible

ENOMEM	Falta memoria
ENODEV	El dispositivo correspondiente al terminal no existe
EPERM	El usuario no tiene los derechos necesarios para realizar la operación
EBUSY	Terminal ocupado

2.2 Un ejemplo de configuración de línea

Antes de proseguir, veamos un breve ejemplo para ilustrar la configuración de una línea:

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

#define PORT_IO "/dev/cua1"      /* Port COM2 */

void main ()
{
    struct termios  term;
    int             fd;

    if ((fd = open (PORT_IO, O_RDWR)) == -1) {
        perror ("open");
        exit (-1);
    }
    if (tcgetattr (fd, &term) == -1) {
        perror ("tcgetattr");
        exit (-1);
    }
    /* MIN = 1 y puesta a 0 de TIME */
    term.c_cc[MIN] = 1;
    term.c_cc[VTIME] = _POSIX_VDISABLE;

    /* configuración del modo de entrada */
    term.c_iflag &= ~(IXON | IXOFF | ICRNL);

    /* configuración modo local: paso al modo canónico */
    term.c_lflag &= ~(ICANON | ISIG | ECHO);

    /* configuración modo control */
    term.c_cflag &= ~(CBAUD | CSIZE);
    term.c_cflag |= B9600; /* velocidad: 9600 bps */
    term.c_cflag |= CS8;   /* paso a 8 bits */
    term.c_cflag &= ~PARENB; /* sin paridad */
}
```

```
/* actualización de la configuración del terminal */
if (tcsetattr (fd, TCSAFLUSH, &term) == -1) {
    perror ("tcsetattr");
    exit (-1);
}
/* lecturas/escrituras en el terminal ... */

close (fd);
}
```

Este programa abre primero el segundo puerto serie (COM2 en otro mundo) en lectura y escritura. Seguidamente, utilizando la función *tcgetattr*, se recupera la configuración del terminal asociado al dispositivo.

En este ejemplo, el modo elegido es el modo canónico, utilizando la tercera configuración MIN/TIME (véase el apartado sobre el modo canónico y el modo no canónico: sección 1.2.1). Esto significa que se efectúa una lectura carácter por carácter (car MIN == 0). La configuración de la línea de entrada es bastante simple: se trata de indicar que se activa la entrada (IXON) así como el control de la entrada (IXOFF). Finalmente, todo carácter CR recibido se convierte en un carácter NL.

La configuración del modo local consiste en activar el modo canónico (ICANON), las señales (ISIG) y el sistema de eco (ECHO).

Finalmente, el modo de control de la línea se realiza configurando la línea para que trabaje a una velocidad de 9.600 baudios, con caracteres de 8 bits y sin paridad.

Una vez convenientemente rellena la estructura, basta con llamar a la función *tcsetattr* para actualizar el terminal.

2.3 La velocidad de transmisión

La norma POSIX.1 proporciona 4 funciones de la biblioteca C para leer y modificar la velocidad de entrada y de salida. En este caso, basta con utilizar una de las cuatro funciones siguientes:

```
#include <termios.h>
#include <unistd.h>
speed_t cfgetospeed (struct termios *termios_p);
int cfsetospeed (struct termios *termios_p, speed_t speed);
speed_t cfgetispeed (struct termios *termios_p);
int cfsetispeed (struct termios *termios_p, speed_t speed);
```


Las variables de tipo `speed_t` deben corresponder de hecho a constantes definidas en `<asm/termbits.h>`: B0, B50, B75, B110, B134, B150, B200, B300, B600, B1200, B1800, B2400, B4800, B9600, B19200, B38400, B57600, B115200, B230400 y B460800. Cada una de estas constantes define una velocidad de transmisión. Su uso es particularmente simple:

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

#define NB_VELOCIDAD 20
#define PORT_IO "/dev/cua1"      /* Port COM2 */

struct velocidad_s {
    speed_t velocidad_b;
    int      velocidad_i;
};

static struct velocidad_s tabla_velocidad[NB_VELOCIDAD] =
{
    {B0, 0},
    {B50, 50},
    {B75, 75},
    {B110, 110},
    {B134, 134},
    {B150, 150},
    {B200, 200},
    {B300, 300},
    {B600, 600},
    {B1200, 1200},
    {B1800, 1800},
    {B2400, 2400},
    {B4800, 4800},
    {B9600, 9600},
    {B19200, 19200},
    {B38400, 38400},
    {B57600, 57600},
    {B115200, 115200},
    {B230400, 230400},
    {B460800, 460800}};

static void muestra_velocidad (speed_t v);
```

```
void main ()
{
    int          fd;
    struct termios term;
    speed_t      velocidad;

    if ((fd = open (PORT_IO, O_RDWR)) == -1) {
        perror ("open");
        exit (-1);
    }
    if (tcgetattr (fd, &term) == -1) {
        perror ("tcgetattr");
        exit (-1);
    }
    if ((velocidad = cfgetispeed (&term)) == -1) {
        perror ("cfgetispeed");
        exit (-1);
    }
    muestra_velocidad (velocidad);

    /* Paso de la línea a 19200 bits por segundo */
    if ((cfsetospeed (&term, B19200)) == -1) {
        perror ("cfsetospeed");
        exit (-1);
    }
    /* actualización de la configuración del terminal */
    if (tcsetattr (fd, TCSAFLUSH, &term) == -1) {
        perror ("tcsetattr");
        exit (-1);
    }
    close (fd);
}

static void muestra_velocidad (speed_t v)
{
    int i;

    for (i = 0; i < NB_VELOCIDAD; i++)
        if (tabla_velocidad[v].velocidad_b == v) {
            printf ("velocidad: %d \n",
                    tabla_velocidad[v].velocidad_i);
            return;
        }
    printf (";Velocidad desconocida!\n");
}
```

Las dos funciones que efectúan únicamente una lectura se implementan con gran simplicidad: basta con enviar el valor de la velocidad del campo `c_cflag` de la estructura `termios`:

```
speed_t cfgetospeed (struct termios *tp)
{
    return (tp->c_cflag & CBAUD);
}
```

La modificación de la velocidad de la línea es similar: basta con modificar el mismo campo:

```
int cfsetospeed (struct termios *tp, speed_t speed)
{
    if (speed & ~CBAUD)
        return 0;

    tp->c_cflag &= ~CBAUD;
    tp->c_cflag |= speed;

    return 0;
}
```

Por esta razón es por lo que tras haber efectuado la modificación de la velocidad de la línea es necesario actualizar los parámetros de la estructura `termios`.

2.4 Control de línea

Existen también cuatro funciones que permiten actuar sobre el control del terminal, utilizando directamente el descriptor de archivo.

El prototipo de estas funciones es el siguiente:

```
#include <termios.h>
#include <unistd.h>

int tcdrain (int fd);
int tcflow (int fd, int action);
int tcflush (int fd, int queue_selector);
int tcsendbreak (int fd, int duration);
```

La función `tcdrain` espera que todos los caracteres en espera hayan sido transmitidos al archivo asociado al descriptor de archivo. Esta función corresponde al uso de la petición `ioctl` `TCSBRK` y equivale al código siguiente:

```
int tcdrain (int fd)
{
    return ioctl (fd, TCSBRK, 1);
}
```

tcflow suspende la transmisión o la recepción de datos provenientes de o destinados al descriptor de archivos. Son posibles cuatro acciones:

- TCOOFF: suspende la salida;
- TCOON: reactiva una salida anteriormente suspendida;
- TCIOFF: emite un carácter STOP que provoca la parada de las transmisiones del terminal hacia el sistema;
- TCION: emite un carácter START que lanza la transmisión de datos entre el terminal y el sistema.

Esta función equivale al uso del *ioctl* con la petición TCXONC:

```
int tcflow (int fd, int action)
{
    return ioctl (fd, TCXONC, action);
}
```

tcflush permite vaciar, pero sin transmitir, los caracteres en espera de transmisión en las memorias intermedias, ya sea para una escritura en el descriptor de archivo asociado o para caracteres que aún no han sido leídos. El comportamiento exacto depende del valor del parámetro *queue_selector*:

- TCIFLUSH: suprime los datos recibidos pero no leídos;
- TCOFLUSH: suprime los datos enviados pero no transmitidos;
- TCIOFLUSH: suprime ambos.

Esta función usa la petición TCFLSH:

```
int tcflush (int fd, int queue_selector)
{
    return ioctl (fd, TCFLSH, queue_selector);
}
```

Para terminar, la función *tcsendbreak* transmite un flujo continuo de bits a 0, por una duración determinada por el segundo parámetro. Si la duración es igual a 0, la transmisión dura entre 0,25 y 0,5 segundos. POSIX.1 define que si la duración no es nula,

el tiempo de transmisión es dependiente de la implementación. En el caso de Linux, la duración de transmisión dura entonces: $\text{duración} * N$ segundos, donde N está comprendido entre 0,25 y 0,5. Hay que destacar que en el caso en que el terminal asociado al descriptor de archivo no utilice una transmisión serie asíncrona, la función no efectúa ninguna operación. Esta función utiliza la petición TCSBR en el caso en que la duración sea inferior o igual a 0 y TCSBRKP en el caso contrario:

```
int tcsendbreak (int fd, int duration)
{
    if (duration <= 0)
        return ioctl (fd, TCSBRK, 0);

    /* número de 100 ms: ¡específico de Linux! */
    return ioctl (fd, TCSBRKP, (duration + 99) / 100 );
}
```

2.5 Identificación del terminal

La norma POSIX.1 define principalmente tres funciones que permiten identificar el terminal actual: *ctermid*, *isatty* y *ttyname*.

La función *ctermid* sólo se mantiene por razones históricas. Bajo Linux, siempre devuelve la cadena */dev/tty*, y posee el prototipo siguiente:

```
#include <stdio.h>

char *ctermid (char *s);
```

La función *isatty* indica si un descriptor de archivo se refiere a un terminal:

```
#include <unistd.h>

int isatty (int fd);
```

Esta función resulta particularmente útil y es aconsejable llamarla antes de utilizar la función *tcgetattr*. Su implementación es particularmente simple:

```
int isatty (int fd)
{
    int save;
    int is_tty;
    struct termios term;

    save = errno; /* conservación de errno en caso de error */
    is_tty = tcgetattr (fd, &term) == 0;
```

```

    errno = save;
    return is_tty;
}

```

Finalmente, la función *ttyname* devuelve el nombre de un terminal:

```

#include <unistd.h>

char *ttyname (int fd);

```

Esta función devuelve un puntero al camino de acceso al terminal del descriptor de archivo. En caso de error, devuelve NULL. Esta función se basa en la llamada al sistema *stat*: una primera llamada al descriptor de archivo permite recuperar el número de dispositivo así como el número del í nodo del archivo. Seguidamente, basta con hacer la misma operación sobre todos los archivos del directorio */dev/* y compara los valores con los del archivo buscado:

```

char *ttyname (int fd)
{
    static const char dev[] = "/dev";
    static char      *name = NULL;
    static size_t    namelen = 0;
    struct stat      st;
    dev_t            mydev;
    ino_t            myino;
    DIR              *dirstream;
    struct dirent     *d;
    int              save = errno;
    int              d_namlen;

    if (isatty (fd) && fstat (fd,&st) < 0)
        return NULL;
    mydev = st.st_dev; /*recuperación de número de dispositivo*/
    myino = st.st_ino; /*y de i-nodo de descriptor de archivo */
                        /* que debe identificarse */

    /* Tratamiento de los archivos que se encuentran en
       el directorio /dev/tty */
    dirstream = opendir (dev);
    if (dirstream == NULL)
        return NULL;
    while ((d = readdir (dirstream)) != NULL)
        /* ¿mismo número de i-nodo? */
        if (->d_ino == myino) {
            d_namlen = strlen (d->d_name) + 1;
            if (sizeof (dev) + d_namlen > namelen) {

```

```
    if (name)
        free (name);
    namelen = (sizeof (dev) + d_namlen) << 1;
    name = malloc (namelen);

    if (!name)
        return NULL;

    memcpy (name, dev, sizeof (dev) - 1);
    name[sizeof (dev) - 1] = '/';
}
memcpy (&name[sizeof (dev)], d->d_name, d_namlen);

/* ¿mismo número de dispositivo? */
if (stat (name, &st) == 0 && st.st_dev == mydev) {
    closedir (dirstream);
    __ttyname = name;
    errno = save;
    return name;
}
}
closedir (dirstream);
errno = save;
return NULL;
}
```

2.6 Grupos de procesos

La norma POSIX.1 define dos funciones de manipulación de grupos de procesos: *tcgetpgrp* y *tcsetpgrp*.

El prototipo de estas dos funciones es el siguiente:

```
#include <termios.h>
#include <unistd.h>

pid_t tcgetpgrp (int fd);
int tcsetpgrp (int fd, pid_t pgrp);
```

La función *tcgetpgrp* permite devolver el número del grupo de procesos perteneciente al proceso en primer plano. Esto equivale a utilizar la petición TIOCGPRP:

```
pid_t tcgetpgrp (int fd)
{
    int pgrp;
```

```
    if (ioctl(fd, TIOCGPGRP, &pgrp) < 0)
        return (pid_t) -1;

    return (pid_t) pgrp;
}
```

La función *tcsetpgrp* permite especificar que el número de grupo de procesos tomará el valor *pgrp*. Sin embargo, es necesario que *pgrp* sea el número de un grupo de procesos de la misma sesión. Esto equivale al uso de la petición *TIOCSGRP*:

```
int tcsetpgrp (int fd, pid_t pgrp)
{
    return ioctl (fd, TIOCSGRP, &pgrp_id);
}
```

2.7 Pseudoterminales

Un proceso que quiera adquirir un pseudoterminal debe utilizar la llamada *open*, para intentar abrir un pseudoterminal amo. Una vez abierto, este proceso o uno de sus hijos puede abrir el pseudoterminal esclavo asociado.

En el caso en que el proceso no posea terminal de control (cosa que se da si el proceso ha utilizado la primitiva *setsid* anteriormente), el pseudoterminal esclavo se convierte automáticamente en su terminal de control.

El programa de ejemplo *talking2.c*, a continuación, utiliza los pseudoterminales para comunicar dos programas, llamados *uno* y *dos*.

```
/* uno.c */

#include <stdio.h>

void main (void)
{
    int          a = 2;
    int          b = 3;
    int          c;

    printf ("%d %d\n", a, b);
    scanf ("%d", &c);
    fprintf (stderr, "uno: Resultado= %d\n", c);
    exit (0);
}
```



```
/* dos.c */

#include <stdio.h>

void main (void)
{
    int          a;
    int          b;
    int          c;

    scanf ("%d %d", &a, &b);
    fprintf (stderr, "dos: a = %d, b = %d\n", a, b);
    c = a + b;
    printf ("%d\n", c);
    exit (0);
}

/* talking2.c */

#include <fcntl.h>
#include <stdio.h>
#include <termios.h>
#include <unistd.h>
#include <sys/ioctl.h>

static char      letter[] = "pqr";
static char      number[] = "012345689";
static char      amo[] = "/dev/ptyXX";
static char      esclavo[] = "/dev/ttyXX";

void main (void)
{
    int          d1 = -1;
    int          d2 = -1;
    int          i;
    int          j;
    struct termios  t;

    /* Búsqueda de un pseudoterminal amo */
    for (i = 0; i < sizeof (letter) && d1 == -1; i++) {
        amo[8] = letter[i];
        for (j = 0; j < sizeof (number) && d1 == -1; j++) {
            amo[9] = number[j];
            d1 = open (amo, O_RDWR);
        }
    }
    if (d1 == -1) {
```

```

    printf ("Ningún pseudo-terminal !\n");
    exit (1);
}
printf ("Pseudoterminal amo %s abierto (%d)\n", amo, d1);
esclavo[8] = amo[8];
esclavo[9] = amo[9];
/* Modificación de los parámetros del terminal */
tcgetattr (d1, &t);
cfmakeraw (&t);
t.c_lflag &= ~ECHO;
(void) tcsetattr (d1, TCSAFLUSH, &t);
/* Creación de un proceso hijo */
switch (fork ()) {
    case -1:          /* Error de creación */
        perror ("fork");
        exit (2);
    case 0:           /* Código del proceso hijo */
        /* Apertura del pseudoterminal esclavo */
        d2 = open (esclavo, O_RDWR);
        if (d2 == -1) {
            close (d1);
            perror ("open");
            exit (3);
        }
        printf ("Pseudoterminal esclavo %s abierto (%d)\n",
                esclavo, d2);
        /* Redirección de los flujos estándar */
        dup2 (d2, 0);
        dup2 (d2, 1);
        /* Cierre del pseudoterminal amo */
        close (d1);
        /* Ejecución del programa "dos" */
        execl ("dos", "dos", NULL);
        perror ("execl");
        exit (4);
        break;
    default:          /* Código del proceso padre */
        /* Redirección de los flujos estándar */
        dup2 (d1, 0);
        dup2 (d1, 1);
        /* Ejecución del programa "uno" */
        execl ("uno", "uno", NULL);
        perror ("execl");
        exit (5);
}
}

```

El programa *uno* escribe dos enteros en su salida estándar y lee una respuesta en su entrada estándar. El programa *dos* lee estos dos números desde su entrada estándar, los suma y envía el resultado a su salida estándar.

El programa *talking2* conecta estos programas mediante un pseudoterminal. Abre un pseudoterminal amo (que se usará para las entradas/salidas del programa *uno*) y crea un proceso hijo. Este último abre el pseudoterminal esclavo correspondiente y redirige su entrada y su salida estándar hacia este pseudoterminal. Seguidamente ejecuta el programa *dos*.

El proceso padre, por su parte, redirige su entrada y su salida estándar hacia el pseudoterminal amo y ejecuta el programa *uno*.

La ejecución de este programa provoca la visualización siguiente:

```
bbj> ./talking2
Pseudo-terminal amo /dev/ptyp8 abierto (3)
Pseudo-terminal esclavo /dev/ttyp8 abierto (4)
dos: a = 2, b = 3
uno: Resultado = 5
```

3. Organización en el núcleo - estructuras de datos

3.1 Organización

Los terminales pueden considerarse como una interfaz lógica entre los datos y el material que debe transmitirse a través de un dispositivo cualquiera como una línea serie, un mouse, una impresora en paralelo o incluso la consola de la máquina de un usuario, como se ilustra en la figura 9.4.

Básicamente son importantes cuatro archivos que encapsulan las operaciones de alto nivel:

- *tty_io.c*: gestiona todas las entradas/salidas de alto nivel sobre los terminales;
- *tty_ioctl.c*: gestiona la llamada a *ioctl* sobre un terminal y se encarga de repercutir la llamada, si es necesario, al gestor del dispositivo;
- *n_tty.c*: se encarga de la disciplina de la línea;
- *pty.c*: se encarga de la gestión de los pseudoterminales, que se basa realmente en los archivos anteriores.

Estos módulos se detallarán en la última parte de este capítulo, dedicada a la implementación de terminales.

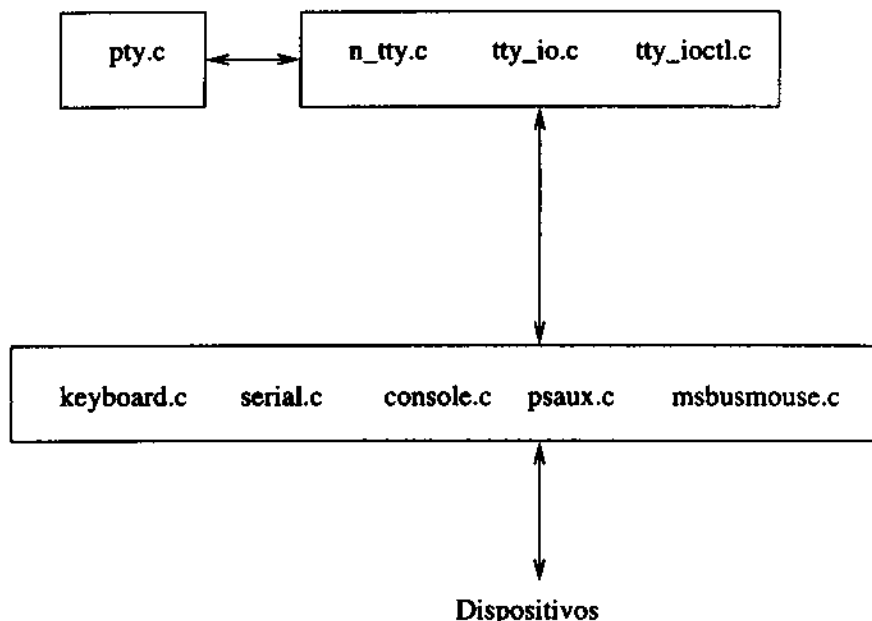


FIG. 9.4 – Organización de los terminales en el núcleo.

3.2 Estructuras de datos internas

3.3 La estructura `tty_struct`

La estructura principal a partir de la cual se efectúan todas las operaciones realizadas en el núcleo es la estructura `tty_struct`, definida en el archivo de cabecera `<linux/tty.h>`.

Esta estructura es particularmente larga, pero su función es absolutamente crucial. Veámosla pues en detalle:

tipo	campo	descripción
int	magic	Número mágico que identifica la estructura: TTY_MAGIC
struct tty_driver	driver	Interfaz de acceso al dispositivo asociado al terminal
struct tty_ldisc	ldisc	Interfaz para la disciplina de la línea
struct termios *	termios	Configuración del terminal
struct termios *	termios_locked	Bloqueo de la configuración del terminal. Si este campo vale NULL, la configuración no está bloqueada

int	pgrp	Identificador del grupo de procesos
int	session	Número de la sesión
kdev_t	device	Número del dispositivo; permite obtener los números mayor y menor del dispositivo
unsigned long	flags	Estado del terminal
int	count	Número de aperturas que se han efectuado en el terminal
struct winsize	winsize	Tamaño de la ventana
unsigned char	stopped:1	Terminal bloqueado
unsigned char	hw_stopped:1	Utilizado en los gestores de bajo nivel para indicar que el dispositivo está en un estado (temporal) no disponible
unsigned char	packet:1	Modo paquete
unsigned char	ctrl_status	Tipo de control para el modo paquete
struct tty_struct*	link	Lista encadenada de terminales utilizada para los pseudoterminal
struct fasync_struct*	fasync	Estructura utilizada por ciertos dispositivos como el mouse. Utiliza ciertas estructuras de datos definidas en el sistema de archivos
struct	flip	Memoria intermedia de datos que deben enviarse a la disciplina de la línea
tty_flip_buffer		No se usa
int	max_flip_cnt	Lista de procesos en espera de escritura
struct wait_queue *	write_wait	Lista de procesos en espera de lectura
struct wait_queue *	read_wait	No se usa
void *	disc_data	Puntero genérico utilizado para manipular una estructura de datos propia del dispositivo
void *	driver_data	

Los campos siguientes se usan para la disciplina de la línea.
Por razones históricas, estos datos se incluyen en esta estructura

unsigned int	column	Número de columnas de la ventana
unsigned char:1	lnext	Gestión del carácter LNEXT
unsigned char:1	erasing	Gestión de la supresión de un carácter
unsigned char:1	raw	Gestión del modo <i>raw</i> (no canónico)
unsigned char:1	real_raw	Otro modo <i>raw</i>
unsigned char:1	icanon	Modo canónico
unsigned char:1	closing	Rechazo de carácter esperando generalmente que los caracteres hayan sido transmitidos correctamente al dispositivo
unsigned short	minimum_to_wake	Tiempo mínimo restante antes de desencadenar el <i>timer</i> en el caso en que se use el modo canónico
unsigned	overrun_time	Duración del desbordamiento
int	num_overrun	Número de situaciones de desbordamiento encontradas
unsigned long	process_char_map	El tamaño de esta tabla es $256/(8 * \text{sizeof}(\text{unsigned long}))$. Se trata de una tabla de bytes que permite saber si los caracteres están definidos
[tamaño]		Memoria de entrada circular
char *	read_buf	Índice del primer carácter no leído
int	read_head	Índice del último carácter no leído
int	read_tail	

int	read_cnt	Número de caracteres en la memoria intermedia
unsigned long {tamaño}	read_flags	El tamaño de esta tabla es $N_TTY_BUF_SIZE/8 * sizeof(unsigned long)$. Se trata de una tabla que indica si una línea situada en la tabla <code>read_buf</code> es legible o no (modo canónico donde los datos son tratados por líneas completas)
int	canon_data	Número de líneas a punto para ser leídas
unsigned long	canon_head	Índice del primer carácter de la primera línea en <code>read_buf</code>
unsigned int	canon_column	Utilizado en el formateado de los datos para la salida: representa el número de la columna del próximo carácter a insertar

La constante `N_TTY_BUF_SIZE` se define en este mismo archivo de cabecera. A título informativo, posee el valor 512. Los diferentes valores que puede tomar el campo `flags` de esta estructura, que indica más un mandato que una opción, son:

constante	significado
TTY_THROTTLED	La línea acepta de nuevo caracteres
TTY_IO_ERROR	Error de entrada/salida
TTY_OTHER_CLOSED	Los demás terminales han sido cerrados
TTY_EXCLUSIVE	El terminal debe utilizarse de manera atómica
TTY_DEBUG	Opción utilizada para la depuración
TTY_DO_WRITE_WAKEUP	Escritura asíncrona a efectuar
TTY_PUSH	Vaciado de las memorias intermedias (caso de un EOF)
TTY_CLOSING	Peticion de cierre del terminal

3.4 La estructura `tty_driver`

La estructura `tty_driver` se define en el archivo de cabecera `<linux/tty_driver.h>`, y define el dispositivo que gestionará la capa de bajo nivel del terminal. Está constituida por los campos siguientes:

tipo	campo	descripción
int	magic	Número mágico que identifica la estructura: TTY_DRIVER_MAGIC
const char *	name	Nombre del dispositivo
int	name_base	Desplazamiento para acceder al nombre del terminal
short	major	Número mayor del dispositivo
short	minor_start	Inicio del número menor del dispositivo
short	num	Número de dispositivos
short	type	Tipo del gestor de dispositivo del terminal
short	subtype	Subtipo del gestor del terminal
struct termios	init_termios	Estructura termios de inicialización del terminal
int	flags	Estados y opciones del terminal
int *	refcount	Estructura termios de inicialización del terminal
struct tty_driver *	other	Utilizado únicamente por los pseudoterminales

Punteros a las estructuras de terminales		
struct tty_struct **	table	Lista de terminales vinculados a este dispositivo
struct termios **	termios	Lista de termios de los terminales
struct termios **	termios_locked	Listas de termios bloqueados de los terminales
<i>Punteros de funciones – ver su definición más adelante</i>		
Punteros de la lista doblemente encadenada		
struct tty_driver *	next	Dispositivo siguiente
struct tty_driver *	prev	Dispositivo anterior

Veamos la definición de los punteros de función de la estructura `tty_driver`. Estos punteros son llamados por el núcleo, y más exactamente por el subsistema de gestión de entradas/salidas:

- `int (*open)(struct tty_struct *tty, struct file *filp):`

Se abre un nuevo terminal. Si esta función falla, devuelve el error `ENODEV`.

- `void (*close)(struct tty_struct *tty, struct file *filp):`

se cierra un dispositivo.

- `int (*write)(struct tty_struct *tty, int from_user, const unsigned char *buf, int count):`

Debe escribir un cierto número de caracteres en el dispositivo. Los caracteres pueden provenir del espacio de usuario (`from_user` positivo) o bien del espacio del núcleo. Devuelve el número de caracteres que se han aceptado para la escritura.

- `void (*put_char)(struct tty_struct *tty, unsigned char ch):`

Debe escribir un carácter en el dispositivo. Si el núcleo utiliza esta rutina, debe llamar a la función `flush_chars` si está definida. Si no hay espacio en la memoria intermedia, el carácter se ignora.

- `void (*flush_chars)(struct tty_struct *tty):`

Se llama después de haber escrito un cierto número de caracteres en el dispositivo con `put_char`.

- `int (*write_room)(struct tty_struct *tty):`

Devuelve el número de caracteres que el dispositivo acepta recibir (en una memoria intermedia) para su escritura. Este número puede cambiar a medida que se llenan las memorias intermedias, o si el control de flujo está activo.

- `int (*chars_in_buffer)(struct tty_struct *tty):`

Devuelve un valor diferente de 0 si se encuentran caracteres en la memoria intermedia.

- `int (*ioctl)(struct tty_struct *tty, struct file *file, unsigned int cmd, unsigned long arg):`

Permite que el gestor de dispositivos implemente la llamada *ioctl* para el dispositivo. Si el parámetro *cmd* no se gestiona o no es reconocido por el gestor, la función devuelve el error `ENOIOCTLCMD`.

- `void (*set_termios)(struct tty_struct *tty, struct termios *old):`

Se llama para advertir al gestor cuando la configuración (termios) de un dispositivo ha sido modificada. Resulta útil destacar que los gestores bien diseñados deberían gestionar el caso en que *old* contenga el valor `NULL`, e intentar hacer algo racional en ese caso.

- `void (*set_ldisc)(struct tty_struct *tty):`

Permite que el gestor de dispositivos sea advertido cuando la configuración (termios) de un dispositivo ha cambiado.

- `void (*throttle)(struct tty_struct *tty):`

Las memorias de salida para la disciplina de la línea están casi llenas. El gestor debería señalar a los procesos que escriben que no debería enviarse ningún carácter más al terminal.

- `void (*unthrottle)(struct tty_struct *tty):`

Indica al gestor de dispositivo que debería señalar que pueden enviarse datos al terminal sin temor a desbordar la memoria intermedia de disciplina de la línea.

- `void (*stop)(struct tty_struct *tty):`

Indica al gestor que debería dejar de enviar caracteres al dispositivo.

- `void (*start)(struct tty_struct *tty):`

Indica al gestor que puede, si lo desea, emitir o reenviar caracteres en el dispositivo.

- `void (*hangup)(struct tty_struct *tty):`

El gestor debería cerrar el dispositivo.

- `void (*flush_buffer)(struct tty_struct *tty):`

Debe vaciar las memorias intermedias.

Este archivo de cabecera proporciona también un cierto número de constantes que permiten utilizar los campos de la estructura `tty_driver`.

El campo `flag` de la estructura `tty_driver` puede tomar los valores siguientes:

constante	significado
TTY_DRIVER_INSTALLED	Caso en que el gestor de dispositivo ya está instalado
TTY_DRIVER_RESET_TERMIOS	Solicita al gestor del dispositivo que reinicie la estructura <code>termios</code> del terminal una vez el último proceso ha cerrado el dispositivo
TTY_DRIVER_REAL_RAW	Utilizado por los pseudoterminales Si está activada, esta opción indica que el dispositivo garantiza que no modificará nunca los caracteres de control si la opción <code>iflag</code> contiene <code>((IGNBRK (BRKINT && PARMRK)) && (IGNPAR INPCK))</code> . De este modo, el dispositivo no enviará indicaciones de paridad o caracteres BREAK a la línea a menos que haya una razón para ello. Esto permite al gestor de dispositivos optimizar las transmisiones. Hay que destacar que también debe asegurar que, si la opción está activada, no señalará los desbordamientos

Los dispositivos pueden ser de varios tipos diferentes (campos `type` y `subtype`):

constante	significado
TTY_DRIVER_TYPE_CONSOLE	Consola
TTY_DRIVER_TYPE_SERIAL	Puerto serie
TTY_DRIVER_TYPE_PTY	Pseudoterminal
TTY_DRIVER_TYPE_SCC	Dispositivo SCC (tarjetas Z8530 - HDLC)
PTY_TYPE_MASTER	Pseudoterminal amo (subtipo para el tipo <code>TTY_DRIVER_TYPE_PTY</code>)
PTY_TYPE_SLAVE	Pseudoterminal esclavo (subtipo para el tipo <code>TTY_DRIVER_TYPE_PTY</code>)

3.5 La estructura `tty_ldisc`

La estructura `tty_ldisc` se define en el archivo de cabecera `<linux/tty_ldisc.h>`. Esta estructura proporciona una interfaz de acceso a la disciplina de la línea, y posee la definición siguiente:

tipo	campo	descripción
int	<code>magic</code>	Número mágico que identifica la estructura: TTY_LDISC_MAGIC
int	<code>num</code>	Identificador de la línea
int	<code>flags</code>	Tipo de línea (el único tipo definido es LDISC_FLAG_DEFINED)

Esta estructura define también punteros de funciones para manipular las entradas/salidas sobre la disciplina de la línea:

```
• int (*open)(struct tty_struct *):
```

Apertura de la línea.

- `void (*close)(struct tty_struct *):`

Cierre de la línea.

- `void (*flush_buffer)(struct tty_struct *tty):`

Vaciado de las memorias intermedias.

- `int (*chars_in_buffer)(struct tty_struct *tty):`

Indica si hay caracteres presentes en la memoria.

- `int (*read)(struct tty_struct *tty, struct file *file, unsigned char *buf, unsigned int nr):`

Lectura.

- `int (*write)(struct tty_struct *tty, struct file *file, const unsigned char *buf, unsigned int nr):`

escritura.

- `int (*ioctl)(struct tty_struct *tty, struct file *file, unsigned int cmd, unsigned long arg):`

Envío de una petición.

- `void (*set_termios)(struct tty_struct *tty, struct termios *old):`

Configuración de la línea.

- `int (*select)(struct tty_struct *tty, struct inode *inode, struct file *file, int sel_type, struct select_table_struct *wait):`

Implementación de la llamada *select*.

- `void (*receive_buf)(struct tty_struct *, const unsigned char *cp, char *fp, int count):`

Devuelve un puntero a la memoria intermedia de datos recibida.

- `int (*receive_room)(struct tty_struct *):`

Número de caracteres en la memoria intermedia de entrada.

- `void (*write_wakeup)(struct tty_struct *):`

Número de caracteres en la memoria intermedia de salida.

El campo `num` de esta estructura puede tener los valores siguientes, que están definidos en el archivo de cabecera `<asm/termios.h>`:

constante	significado
N_TTY	Consola
N_SLIP	Línea Slip
N_MOUSE	Mouse
N_PPP	Línea PPP
N_STRIP	Starmode Radio IP

3.6 La estructura *winsize*

Esta estructura se define en el archivo de cabecera `<asm/termios.h>`. Contiene simples campos que permiten configurar el tamaño de la ventana que será la del terminal. Veamos la definición:

tipo	campo	descripción
unsigned short	<code>ws_row</code>	Número de líneas
unsigned short	<code>ws_col</code>	Número de columnas
unsigned short	<code>ws_xpixel</code>	No se usa
unsigned short	<code>ws_ypixel</code>	No se usa

Esta estructura se utiliza principalmente en terminales de visualización, como la consola.

3.7 La estructura *tty_flip_buffer*

Esta estructura es una memoria intermedia circular que se utiliza para la recepción de caracteres. Esta memoria permite la transferencia entre los datos provenientes de la línea, hacia la disciplina de línea del terminal asociado.

tipo	campo	descripción
struct <code>tq_struct</code>	<code>tqueue</code>	Lista de tareas
unsigned char [2*TTY_FLIPBUF_SIZE]	<code>char_buf</code>	Tabla con los caracteres recibidos de la línea serie
char [2*TTY_FLIPBUF_SIZE]	<code>flag_buf</code>	Tabla de opciones de comunicación
char *	<code>char_buf_ptr</code>	Puntero al primer carácter de la memoria intermedia
unsigned char *	<code>flag_buf_ptr</code>	Puntero a la primera opción
int	<code>count</code>	Número de caracteres en la memoria
int	<code>buf_num</code>	Número de la memoria (0) o (1). Utilizado para la transferencia de datos de la memoria circular hacia la disciplina de línea

La constante `TTY_FLIPBUF_SIZE` se fija a 512.

4 Implementación

Como hemos visto en la sección anterior, la implementación de los terminales se agrupa principalmente en cuatro módulos. Cada uno de ellos se especializa en ciertas funciones. Veamos su descripción detallada.

4.1 La inicialización

Las inicializaciones se efectúan en el módulo *tty_io.c* encargado de gestionar las entradas/salidas sobre los terminales. Se exceptúan las funciones de inicialización que se presentarán más adelante. Las principales funciones de inicialización son las siguientes:

- **tty_init**: inicializa los terminales (*tty*) de la máquina. Esto consiste en llenar los campos de la variable estática a nivel del módulo *dev_tty_driver* que agrupa todos los terminales de la máquina. El número mayor asignado es *TTY_MAJOR*. Esta función realiza seguidamente el registro de la consola, que es un terminal particular, llamando a la función *tty_register_driver*. Seguidamente, en función de las opciones de compilación que se hayan fijado, se llaman las funciones de inicialización de los diferentes dispositivos. Estas funciones son opcionales, por lo que no se detallarán.

Al final de esta función, se efectúa el método de inicialización de los pseudoterminales (función *pty_init*) y dispositivos *vcs* (función *vcs_init*).

- **get_tty_driver**: crea una variable de tipo estructura *tty_driver*, y llena los campos correspondientes a los números menores y mayores del dispositivo.
- **init_dev**: inicializa un nuevo dispositivo. En un primer momento, esta función llama a *get_tty_driver* para crear y generar los datos relativos al dispositivo.

Tras esta operación, un bucle asigna cada uno de los terminales asociados a un dispositivo, y los inserta en la tabla *termios* de la estructura *tty_struct*, sin olvidar incrementar el campo *refcount* de la estructura *tty_driver* y el campo *count* de la estructura *tty_struct*.

En el caso en que se trate de un pseudoterminal, se asignan dos terminales cada vez (terminal amo y terminal esclavo).

Esta función utiliza las operaciones de asignación de memoria *get_free_page*, *kmalloc* y de liberación *free_page* y *kfree_s*.

- `initialize_tty_struct`: inicializa una estructura `tty_struct` con los valores predeterminados.
- `tty_register_driver`: es llamada por un dispositivo para registrarse. Esta función inserta el dispositivo en la lista encadenada de dispositivos pasada como parámetro. Hay que destacar que esta función utiliza la función `register_chrdev` que permite registrar el dispositivo como un dispositivo en modo carácter.
- `console_init`: inicializa el dispositivo de la consola. Esta función se declara aparte porque se lanza lo antes posible al arrancar la máquina, ya que es difícil poder detectar un error de arranque antes de que la consola esté operativa.

4.2 Las entradas/salidas en los terminales

El módulo `tty_io.c` se encarga ante todo de definir las entradas/salidas sobre los terminales. Las funciones encargadas de estas entradas/salidas se registran en la variable estática `tty_fops` de tipo estructura `file_operations`. Por ello es por lo que define las funciones siguientes:

- `tty_read`: llama a la función especializada del dispositivo utilizando el puntero de función puesto a nuestra disposición mediante el campo `read` de la estructura `tty_ldisc`.
- `tty_write`: esta función delega el tratamiento a la función `do_tty_write`.
- `do_tty_write`: utilizando el puntero de función `write` de la estructura `tty_ldisc`, esta función intenta escribir todo el contenido de la memoria intermedia de datos en espera.
- `tty_lseek`: devuelve siempre el error `ESPIPE`.
- `tty_select`: llama simplemente a la función `select` de la estructura `tty_ldisc`.
- `tty_open`: esta función realiza la apertura de un terminal, utilizando principalmente las funciones definidas en la sección anterior `init_dev`.
- `tty_release`: ejecuta la función `release_dev`.
- `release_dev`: efectúa una escritura asíncrona (vacía la memoria intermedia) y ejecuta seguidamente una llamada al método `close` asociado al dispositivo. Tras haber efectuado esta llamada, los campos de la estructura `tty_struct` se liberan.
- `tty_ioctl`: en función de los diferentes mandatos recibidos, este método se limita a posicionar bytes en las tablas `flags` de `tty_struct`, o bien a devolver las

estructuras cuando sea necesario. Hay que destacar que esta función es utilizada por el terminal, no por la disciplina de línea.

- `tty_fasync`: ejecuta la función `fasync_helper`.
- `fasync_helper`: esta función (y por tanto la llamada asociada `tty_fasync`) se usa bastante poco, excepto por algunos dispositivos particulares, como el mouse, para implementar la cola de escritura asíncrona.

4.3 Otras funciones generales

El módulo `tty_io.c` contiene otras funciones que se utilizan ampliamente en los demás módulos. Algunas son funciones de verificación:

- `_tty_name`: devuelve el nombre del terminal.
- `tty_name`: llama a `_tty_name`.
- `tty_paranoia_check`: verifica que el terminal que se va a utilizar es realmente un terminal, gracias a su número mágico. Esta función se utiliza en toda llamada que utiliza un terminal. Esta técnica de desarrollo evita muchos errores.
- `check_tty_count`: cuenta el número de descriptores de archivos abiertos en el terminal.
- `tty_register_ldisc`: llena la estructura `tty_ldisc` pasada como parámetro.
- `tty_check_change`: esta función verifica si el estado del terminal se ha modificado. Si el proceso que llama a esta función está en primer plano, se envía la señal `SIGTTOU`.
- `disassociate_ctty`: esta función es llamada por el líder de una sesión cuando desea separarse de su terminal de control. Esta función envía una señal `SIGHUP` al grupo de procesos en segundo plano y suprime el proceso de control de la sesión; finalmente, suprime el terminal de control de todos los procesos del grupo de procesos.
- `vt_waitactive`: esta función pone al proceso en espera hasta que una consola virtual se haya activado, o bien si la tarea se interrumpe.
- `reset_vc`: reinicia una consola virtual.
- `complete_change_console`: conmutación a otra consola virtual.
- `change_console`: paso a otra consola virtual. Llama a las funciones `reset_vc` y `complete_change_console`.

- `wait_for_keypress`: duerme un proceso hasta que se pulse una tecla, que se detecta mediante la variable global `keypress_wait`.
- `stop_tty`: deja a un terminal en estado parado, poniendo el campo `stopped` del terminal a 1, y ejecutando la función `stop` del dispositivo, si está definida. Esta función duerme todos los procesos que se sirven de este proceso.
- `start_tty`: función inversa, que pone el campo `stopped` a 0 y despierta todos los procesos bloqueados sobre este terminal.
- `do_SAK`: implementación del sistema *Secure Attention Key*. El principio es evitar los caballos de Troya liquidando todos los procesos asociados a un terminal cuando el usuario se sirve de esta funcionalidad. Ello es necesario para las aplicaciones muy sensibles.
- `tty_default_put_char`: escribe un carácter llamando a la función `write` de la estructura `tty_driver`.

4.4 Gestión de la desconexión

Linux declara una variable `hung_up_tty_fops` de tipo estructura `file_operations` cuyo objetivo es poder gestionar las desconexiones bruscas (corte de línea, interrupción de la comunicación...).

- `hung_up_tty_read`: devuelve 0;
- `hung_up_tty_write`: devuelve el error EIO;
- `hung_up_tty_select`: devuelve 1;
- `hung_up_tty_ioctl`: devuelve ENOTTY si la petición es TIOCSPGRP si no devuelve EIO.

Algunas funciones suplementarias se encargan de gestionar esta posibilidad:

- `do_tty_hangup`: tras haber intentado escribir lo que se encuentre en la memoria intermedia, cierra la disciplina de línea, envía las señales SIGHUP y SIGCONT a todos los líderes de los grupos de sesiones.
- `tty_hangup`: lanza la función `do_tty_hangup`;
- `tty_vhangup`: lo mismo;
- `tty_hung_up_p`: comprueba si la dirección del campo `f_op` del descriptor de archivo del dispositivo es igual a la dirección de la variable `hung_up_tty_fops`.

4.5 La gestión de la disciplina de la línea

La disciplina de línea se gestiona en una pequeña parte en el archivo *tty_io.c* y en lo esencial en *n_tty.c* y *tty_ioctl.c*.

Las funciones situadas en *tty_io.c* son las siguientes:

- **tty_set_ldisc**: inicializa una disciplina de línea para un terminal. Esta función configura el campo *ldisc* de la estructura *tty_struct*.
- **flush_to_ldisc**: esta función se llama en una interrupción de software para transferir los datos de la memoria de entrada hacia la memoria de la disciplina de la línea.

El archivo *n_tty.c* se especializa en la disciplina de la línea, y en la organización de los caracteres. Los accesos a la disciplina de la línea se gestionan a través de la variable *tty_ldisc_N_TTY* de tipo *tty_ldisc*. Veamos las funciones que permiten efectuar las operaciones sobre la línea:

- **put_tty_queue**: inserta un carácter en la memoria intermedia *read_buf* de la línea.
- **n_tty_flush_buffer**: vacía la memoria de entrada. Esta operación consiste en poner a cero todos los campos que gestionan las memorias de la estructura *tty_struct*, y en vaciar las tablas que gestionan las memorias intermedias. Los datos se pierden.
- **n_tty_chars_in_buffer**: devuelve el número de caracteres encontrados en la memoria intermedia que deben enviarse al usuario. Corresponde al campo *read_cnt* de *tty_struct*.
- **opost**: efectúa la acción OPOST, es decir, «dar formato» a los caracteres. Aquí es donde se interpretan los caracteres *\n*, *\r*... La interpretación consiste en realidad en modificar los valores del campo *column* de *tty_struct*.
- **put_char**: llamada a la función *put_char* del dispositivo.
- **echo_char**: envía un carácter ECHO a la línea.
- **finish_erasing**: función anexa utilizada para enviar el carácter / seguido de un espacio a la línea. Al finalizar la llamada, el terminal vuelve al modo no destructivo: *erasing* se pone a 0.
- **eraser**: gestión de los caracteres de supresión ERASE, WERASE y KILL, utilizando la función *finish_erasing*.

- `isig`: permite enviar señales a los procesos vinculados a un terminal.
- `n_tty_receive_break`: envía la señal `SIGINT` al terminal. Utiliza para ello la función `isig`.
- `n_tty_receive_overnun`: recepción de un caso de *overnun*. El campo `num_overnun` se incrementa.
- `n_tty_receive_parity_error`: recepción de un error de paridad. Si la línea ha sido configurada para marcar las paridades de errores, entonces este error de paridad se trata insertando en la cola destinada al terminal la secuencia de caracteres `\377\0`.
- `n_tty_receive_char`: recepción de un carácter. Si la línea está en modo *raw* (campo `raw` de `tty_struct` activo), el carácter se inserta inmediatamente en la cola.

Si el terminal está parado (campo `stopped`), el terminal se reactiva mediante la llamada a la función `start_tty`.

A partir de aquí, el carácter se trata en función de su naturaleza y de la configuración de la línea.

- `n_tty_receive_buf`: recepción de una memoria intermedia de caracteres. En este caso, cada uno de los caracteres recibidos se trata de hecho llamando a la función `n_tty_receive_char`.
- `n_tty_receive_room`: devuelve el número de caracteres recibidos.
- `is_ignored`: indica si la señal pasada como parámetro se trata o no.
- `n_tty_set_termios`: actualiza la estructura `termios` de un terminal.
- `n_tty_close`: cierra la disciplina de la línea.
- `n_tty_open`: abre la disciplina, inicializando ciertos campos.
- `input_available_p`: indica si quedan caracteres por leer.
- `copy_form_read_buf`: copia los caracteres de la memoria del terminal hacia el espacio de memoria del usuario.
- `read_chan`: efectúa la lectura de alto nivel de los caracteres en la entrada. Esta función gestiona también las diferentes lecturas en modo canónico o en modo no canónico.
- `write_chan`: función de escritura de alto nivel.

- `normal_select`: implementación de alto nivel de la llamada *select* asociada al terminal.

Para terminar, el archivo *tty_ioctl.c* define las funciones útiles para gestionar las llamadas a *ioctl* que afectan también a la disciplina de la línea:

- `tty_wait_until_sent`: pone en espera al proceso hasta que los datos hayan sido enviados.
- `unset_locked_termios`: desbloquea un terminal.
- `change_termios`: modifica el contenido de una estructura `termios`. Esta operación es atómica. Se trata de actualizar principalmente los campos y llamar a la función `set_termios` del campo `ldisc`.
- `set_termios`: efectúa la modificación de una estructura `termios`. Esta función utiliza de hecho `change_termios`.
- `get_termio`: copia una estructura `termio`.
- `inq_canon`: calcula el número de caracteres que pueden recuperarse.
- `n_tty_ioctl`: gestiona las diferentes peticiones *ioctl*.

4.6 Los pseudoterminales

La gestión de los pseudoterminales se efectúa en el módulo *pty.c*. La única función exportada es `pty_open`. Los pseudoterminales se gestionan como un dispositivo particular.

La gestión de los pseudoterminales se efectúa con una estructura algo particular: la estructura `pty_struct`, cuya definición es la siguiente:

tipo	campo	descripción
int	magic	Número mágico de la estructura: PTY_MAGIC
struct wait_queue *	open_wait	Lista de procesos en espera de una apertura

Esta estructura se utiliza mediante una variable estática en el módulo `pty_state` de tamaño `NR_PTYS`. Esta estructura se utiliza para definir el campo `driver_data` de la estructura `tty_struct`.

Veamos la lista de funciones que permiten efectuar la gestión de estos terminales algo particulares:

- `pty_init`: inicialización de un pseudoterminal. El tipo del terminal se asigna a `TTY_DRIVER_TYPE_PTY`.

Esta función define de hecho dos terminales: el amo y el esclavo, que difieren por su subtipo (campo `subtype`).

- `pty_open`: abre un nuevo pseudoterminal. Esta operación consiste en actualizar los campos del terminal pasado como parámetro.
- `pty_close`: cierre de un pseudoterminal. Tras haber terminado las entradas/salidas en curso, el terminal se desactiva, poniendo su estado a `TTY_OTHER_CLOSED`.
- `pty_set_termios`: modifica el campo `c_cflag` de `termios` para hacer pasar el terminal a 8 bits, y para activar el receptor.
- `pty_unthrottle`: utilizada por la disciplina de la línea para indicar que puede recibir más caracteres. La opción `TTY_THROTTLED` está siempre posicionada para los pseudoterminales, para forzar la disciplina de la línea a llamar siempre a esta función cuando haya menos de `TTY_THRESHOLD_UNTHROTTLE` caracteres en las memorias intermedias. Esto es necesario porque cada vez que se llama a esta función, los procesos que envían caracteres al terminal y que están bloqueados por una escritura se despiertan.
- `pty_write`: escritura en el pseudoterminal. Los caracteres que deben escribirse se envían a la disciplina de la línea mediante la llamada a la función `receive_buf` del campo `ldisc` del terminal.
- `pty_write_room`: llama a la función `receive_room` del campo `ldisc` del terminal.
- `pty_chars_in_buffer`: llama a la función `chars_in_buffer` del campo `ldisc` del terminal.
- `pty_flush_buffer`: vacía la memoria lanzando la función `flush_buffer` del campo `ldisc` del terminal.

Capítulo 10

Comunicación por tuberías

Primitivas detalladas

`mkfifo`, `pipe`, `pclose`, `popen`

1 Conceptos básicos

1.1 Presentación

Las tuberías constituyen un mecanismo de comunicación entre procesos. La transmisión de datos entre procesos se efectúa a través de un canal de comunicación: los datos escritos en un extremo del canal se leen en el otro extremo.

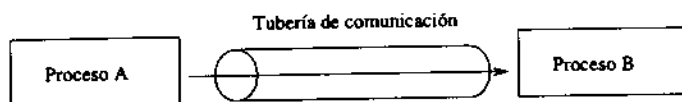


FIG. 10.1 – Comunicación entre dos procesos mediante tuberías

Desde un punto de vista algorítmico, se puede comparar la tubería con una cola de caracteres de tipo *FIFO* (*First In First Out*): primero en entrar, primero en salir. Con este sistema, cuando se lee un dato en la tubería, se retira automáticamente.

Las tuberías no ofrecen una comunicación estructurada. La lectura de los datos es independiente de la escritura, por lo que no es posible a nivel de las llamadas al sistema conocer el tamaño, el emisor o el destinatario de los datos contenidos en la tubería. Por el contrario, una ventaja de este método de comunicación es que permite leer de una sola vez datos que se han podido escribir en varias ocasiones.

Una utilización habitual del mecanismo de tuberías se realiza a través del intérprete de mandatos en el encadenamiento de mandatos:

```
gandal# ps -aux | grep -v root | tail
dumas 440 0.2 3.4 2084 508 1 S 09:45 0:16 fvwm
dumas 450 0.9 21.1 4900 3112 p5 S 10:06 1:06 emacs
dumas 489 0.0 4.6 3292 684 1 S 10:10 0:01 xterm -sb -fn fixed
dumas 490 0.0 2.9 1156 428 p6 S 10:10 0:01 bash
dumas 492 0.0 5.9 3292 880 1 S 10:10 0:01 xterm -sb -fn fixed
dumas 493 0.0 3.5 1140 516 p7 S 10:10 0:00 bash
dumas 543 2.7 6.3 1800 932 p6 S 10:27 2:32 xosview -1
dumas 721 0.3 3.9 1092 580 p0 S 11:52 0:01 elm
dumas 800 0.0 2.4 824 364 p5 R 11:59 0:00 ps -aux
dumas 802 0.0 1.6 768 248 p5 S 11:59 0:00 tail
```

En el ejemplo anterior, se crean dos tuberías. El primer proceso *ps* envía el resultado del mandato en la primera tubería, en lugar de enviar a la salida estándar. Este resultado se recupera por el mandato *grep* que lo trata y devuelve los datos tratados en una segunda tubería al proceso *tail*, el cual muestra su resultado en la entrada estándar. La figura 10.2 esquematiza la circulación de datos entre los diferentes procesos.

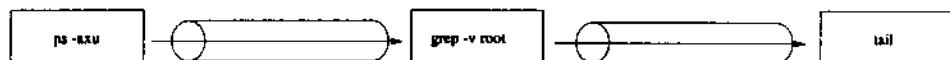


FIG. 10.2 – El intérprete de mandatos y las tuberías.

1.2 Tuberías anónimas y tuberías con nombre

La gestión de las tuberías está integrada en el sistema de archivos. El acceso a las tuberías se realiza por descriptores de entradas/salidas: un descriptor para la lectura en la tubería y un descriptor para la escritura en la tubería.

Históricamente, el primer tipo de tuberías es la tubería anónima. Se crea por un proceso y la transmisión de los descriptores sólo se hace por herencia hacia sus descendientes. Este mecanismo es restrictivo porque sólo permite la comunicación entre procesos cuyo antecesor común es el creador de la tubería.

Las tuberías con nombre (*FIFO* o *named pipe* en la literatura anglosajona) permiten salvar esta restricción porque se manipulan exactamente como archivos en lo que respecta a las operaciones de apertura, cierre, lectura y escritura. Esto es posible porque existen físicamente en el sistema de archivos:

```
gandalf# ls -al tube_exemple
prw-r--r--  1 dumas  users   0 May  1  16:17 tube_exemple
gandalf# file tube_exemple
tube_exemple: fifo (named pipe)
```

Es posible identificar un archivo de tipo tubería con nombre por el atributo *p* mostrado por el mandato *ls*. Es posible crear una tubería con nombre bajo el intérprete de mandatos mediante el mandato *mkfifo*.

2 Llamadas al sistema

Existe una llamada al sistema estrictamente dedicada a la creación de tuberías anónimas: *pipe*. Las tuberías con nombre se crean utilizando la función *mkfifo*, o bien la llamada al sistema *mknod*.

Las otras operaciones de acceso a las tuberías, como la escritura, la lectura, etc., se efectúan mediante las llamadas al sistema estándar de entradas/salidas (como las primitivas *read* y *write*).

2.1 Tuberías anónimas

2.1.1 Creación de una tubería

La llamada *pipe* permite crear una tubería anónima. Para efectuar esta operación, hay que pasar como parámetro una tabla de dos enteros.

```
#include <unistd.h>
int pipe (int filedes[2]);
```

Si la llamada se efectúa con éxito, la tabla de enteros contendrá el descriptor de lectura (*filedes[0]*) y de escritura (*filedes[1]*) de la tubería creada. Esta llamada al sistema puede generar los errores siguientes:

error	significado
EFAULT	La tabla pasada como parámetro no es válida
EMFILE	Se ha alcanzado el número máximo de archivos abiertos para el proceso actual
ENFILE	Se ha alcanzado el número máximo de archivos abiertos en el sistema

A partir de estas informaciones, la lectura/escritura en la tubería se efectúa de una manera transparente por las llamadas al sistema tradicionales *read* y *write*.

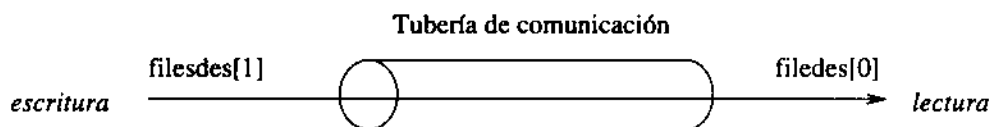


FIG. 10.3 – Tubería anónima

Resulta útil señalar que es posible utilizar funciones de alto nivel para las operaciones de lectura y escritura como *printf*, *sprintf*, *scanf*, *scanf...* Sin embargo, estas operaciones utilizan memorias intermedias, por lo que todo carácter escrito no está inmediatamente disponible en el otro extremo de la tubería, a menos que cada operación de escritura vaya seguida de una llamada a *fflush*.

La utilización de una tubería por un proceso único tiene un interés limitado. La creación de una tubería anónima va seguida de la creación de un proceso que hereda descriptores sobre la tubería y puede por tanto comunicarse con el creador de la tubería. Es importante realizar estas operaciones en este orden porque si no la tubería sólo podría ser accedida por uno solo de los dos procesos.

Cada proceso posee entonces un descriptor en lectura y un descriptor en escritura en la misma tubería. A fin de comunicarse correctamente, los procesos deben elegir un sentido de transmisión. Cada uno utiliza entonces uno solo de los dos descriptores. Uno de los procesos escribe en la tubería, y el otro lee. Los descriptores inutilizados pueden cerrarse por la llamada al sistema *close*. Una vez cerrado, el descriptor no permite ya acceder a la tubería.

De modo predeterminado, la lectura en una tubería es bloqueadora. El proceso lector queda bloqueado hasta que lee la cantidad de datos precisa en la llamada *read*. Dos procesos comunicantes pueden adoptar pues un protocolo para evitar bloquearse mutuamente.

2.1.2 Un ejemplo de uso

Vamos a implementar el ejemplo del principio del capítulo que consiste en realizar de una manera lógica lo que ocurre en una llamada a los tres mandatos *ps*, *grep* y *tail*.

El programa que realiza esta operación utiliza dos veces la llamada *fork* para crear un hijo y un nieto, como muestra la figura 10.4. De este modo, cada proceso se encarga de la ejecución de un programa.

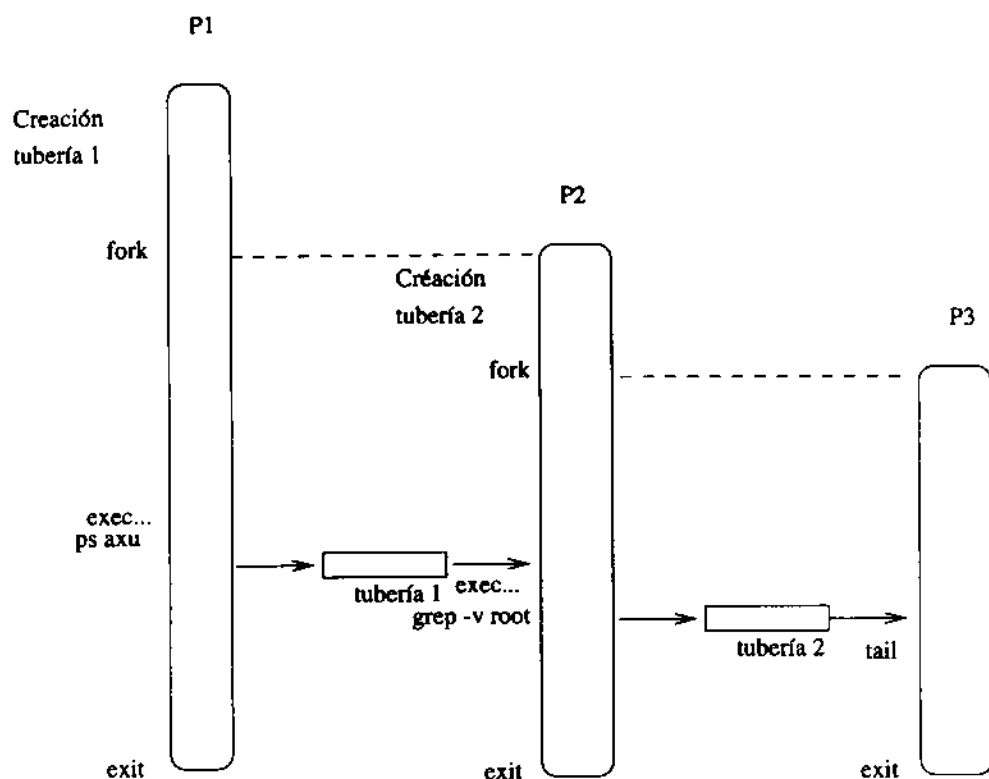


FIG. 10.4 – Ejemplo de tuberías anónimas

Además, la llamada al sistema *dup2* (véase el capítulo 6, sección 4.2), que permite duplicar una entrada de la tabla de descriptores de archivos, se utiliza para redirigir las entradas/salidas estándar hacia las tuberías.

```

#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>

```



```
#define CLOSE_ALL(_fd_) close(_fd_[0]); close(_fd_[1])

static void hijo (int fd1[2]);

void main ()
{
    int fd[2];

    /* Creación de la primera tubería: tube 1 */
    if (pipe (fd) == -1) {
        perror ("pipe");
        exit (-1);
    }
    /* Creación de un proceso hijo P1 que ejecutará *
     * la función hijo (fd)                               */
    switch (fork ()) {
        case -1:
            perror ("fork");
            exit (-1);
            break;
        case 0:
            hijo (fd);
            break;
        default:
            /* Redirección del descriptor de salida *
             * estándar hacia la tubería 1          */
            dup2 (fd[1], STDOUT_FILENO); /* stdout */
            /* Cierre de los descriptors de la tubería 1 */
            CLOSE_ALL (fd);

            /* ejecución de: ps axu */
            execlp ("ps", "ps", "axu", NULL);
            perror ("execlp");
            exit (-1);
            break;
    }
}

static void hijo (int fd1[2])
{
    int          fd2[2];
    /* Creación de una segunda tubería */
    if (pipe (fd2) == -1) {
        perror ("pipe");
        exit (-1);
    }
}
```

```

/* Creación de un proceso P2, nieto del *
 * proceso principal */
switch (fork ()) {
    case -1:
        perror ("fork");
        exit (-1);
        break;
    case 0:
        /* Cierre de los descriptores de la tubería 1 */
        CLOSE_ALL (fd1);
        /* Redirección del descriptor de entrada *
         * estándar a partir de la tubería 2 */
        dup2 (fd2[0], STDIN_FILENO); /* stdin */
        /* Cierre de los descriptores de la tubería 2 */
        CLOSE_ALL (fd2);
        /* ejecución de: tail */
        execlp ("tail", "tail", NULL);
        perror ("execlp");
        exit (-1);
        break;
    default:
        /* Redirección del descriptor de entrada *
         * estándar a partir de la tubería 1 */
        dup2 (fd1[0], STDIN_FILENO); /* stdin */
        /* Redirección del descriptor de salida *
         * estándar hacia la tubería 2 */
        dup2 (fd2[1], STDOUT_FILENO); /* stdout */
        /* Cierre de los descriptores *
         * de las tuberías 1 y 2 */
        CLOSE_ALL (fd1);
        CLOSE_ALL (fd2);
        /* ejecución de: grep -v root */
        execlp ("grep", "grep", "-v", "root", NULL);
        perror ("execlp");
        exit (-1);
        break;
}
}

```

El proceso que ejecuta este programa genera otros dos procesos mediante la llamada al sistema *fork*. En el primer proceso, llamado padre, la entrada estándar se redirige hacia la entrada estándar del hijo. El hijo redirige su salida estándar a la entrada estándar de su propio hijo. Finalmente, cada uno de ellos, gracias a la llamada *execlp*, lanza el programa deseado.

Este ejemplo evidencia la simplicidad de uso y de manipulación de las tuberías anónimas. La operación de encadenamiento que propone el intérprete de mandatos mediante el carácter | se implementa de esta manera.

2.1.3 Creación de una tubería y ejecución de un programa

Es frecuente utilizar tuberías para comunicarse con un programa externo, como se hacía en el ejemplo anterior. Se proporcionan dos funciones de biblioteca para simplificar esta tarea:

```
#include <stdio.h>

FILE *popen (const char *command, const char *type);
int pclose (FILE *stream);
```

La función *popen* crea una tubería, y luego un proceso hijo. Este proceso ejecuta el mandato especificado por el parámetro *mandato*. Se devuelve un descriptor de entradas/salidas correspondiente a la tubería, según el valor del parámetro *type*:

- si *type* es la cadena «r», el descriptor es accesible en lectura, y permite acceder a la salida estándar del mandato;
- si *type* es la cadena «w», el descriptor es accesible en escritura, y permite acceder a la entrada estándar del mandato.

La función *pclose* puede llamarse para cerrar la tubería, y utiliza la función *fclose* para el cierre. Luego provoca la finalización del proceso hijo asociado, que había sido creado por la llamada a *popen*.

2.2 Las tuberías con nombre

La creación de una tubería con nombre se efectúa mediante la función de la biblioteca C *mkfifo*. Tras una llamada a *stat*, el campo *st_mode* de la estructura *stat* correspondiente al tipo de archivo tiene por valor *S_IFIFO*:

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

static void uso (char *prg);
```

```
void main (int argc, char **argv)
{
    struct stat      st;
    if (argc != 2)
        uso (argv[0]);
    if (stat (argv[1], &st) == -1) {
        perror ("stat");
        exit (-1);
    }
    if (S_ISFIFO (st.st_mode))
        printf (" %s es de tipo tubería con nombre \n", argv[1]);
    else
        printf (" %s no es de tipo tubería con nombre: %o \n",
                argv[1], st.st_mode & S_IFMT);
}

static void uso (char *prg)
{
    fprintf (stderr, "%s NomArch: ¿el archivo es de tipo FIFO?\n",
            prg);
    exit (-1);
}
```

El prototipo de la función *mkfifo* es el siguiente:

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

int mkfifo (const char *path, mode_t mode);
```

Sin embargo, no se trata de una llamada al sistema. En realidad, esta función se implementa de una manera muy simple, con la llamada *mknod* detallada en el capítulo 6, sección 2.1.

La implementación de esta función se realiza con la llamada siguiente:

```
mknod(path, mode | S_IFIFO, 0);
```

Hay que destacar que el usuario puede crear una tubería con nombre utilizando *mkfifo* o bien el mandato *mknod*.

Como todo archivo, las tuberías con nombre permiten las entradas/salidas con bloqueo (de modo predeterminado) o sin bloqueo (especificando la opción *O_NDELAY* o *O_NONBLOCK* al abrir con *open*). Todas las operaciones de manipulación de un archivo son válidas para una tubería con nombre, salvo las llamadas a *lseek*.

3 Presentación general de la implementación

La implementación de las tuberías se realiza de manera un tanto peculiar debido a la gran similitud de comportamiento entre las tuberías anónimas y las tuberías con nombre. Sin embargo, para comprender el funcionamiento de la implementación de las tuberías es necesario haber consultado el capítulo sobre el sistema de archivos, ya que una tubería es un archivo particular del sistema de archivos.

Desde el punto de vista del i-nodo de una tubería, el campo `i_pipe` se pone a 1, lo que identifica el tipo del i-nodo. El campo `u` del i-nodo está constituido, en el caso de una tubería (anónima o no), por la estructura `pipe_inode_info` descrita en el archivo `<linux/pipe_fs_i.h>`:

tipo	campo	descripción
<code>struct wait_queue *</code>	<code>wait</code>	Variable usada para sincronizar los accesos concurrentes a la tubería
<code>char *</code>	<code>base</code>	Puntero a la página de memoria destinada a los datos de la tubería
<code>unsigned int</code>	<code>start</code>	Posición en la página de memoria
<code>unsigned int</code>	<code>len</code>	Número de bytes en espera en la tubería
<code>unsigned int</code>	<code>lock</code>	Bloqueo sobre la tubería
<code>unsigned int</code>	<code>rd_openers</code>	Número de procesos que han abierto la tubería con nombre en lectura
<code>unsigned int</code>	<code>wr_openers</code>	Número de procesos que han abierto la tubería con nombre en escritura
<code>unsigned int</code>	<code>readers</code>	Número de procesos que tienen acceso en lectura a la tubería
<code>unsigned int</code>	<code>writers</code>	Número de procesos que tienen acceso en escritura a la tubería

El tamaño de una tubería le parece infinito al usuario. Sin embargo, por razones de rendimiento, se limita a 4 KB (valor de la constante `PIPE_BUF` definida en el archivo `<linux/limits.h>`). Este límite corresponde al valor límite para la realización de una escritura atómica en la tubería. El usuario puede escribir más datos en la tubería, pero dejará de tener garantizada la atomización de la operación.

3.1 Las tuberías con nombre

La creación de una tubería con nombre corresponde a la creación de un archivo cualquiera, y se detalla en el archivo `fs/fifo.c`.

Las operaciones de lectura y de escritura se encuentran en el archivo destinado a las tuberías anónimas `fs/pipe.c`, porque las operaciones a efectuar son exactamente las mismas.

3.2 Las tuberías anónimas

La creación de una tubería anónima se efectúa en un primer momento creando un nuevo i-nodo en el sistema de archivos. Este i-nodo se inicializa con los diferentes campos útiles como los identificadores del creador, etc.

El punto importante aquí es que este i-nodo posee un número de enlaces nulo. De este modo, este archivo no es visible en el sistema de archivos, y la única manera de acceder a él, desde el punto de vista del programador, es poseer un descriptor sobre este archivo.

4 Presentación detallada de la implementación

4.1 Creación del i-nodo de una tubería

La creación del i-nodo de una tubería se efectúa por la función `get_pipe_inode` del archivo `fs/inode.c`. En un primer momento, llamando a la función `get_empty_inode`, se asigna un i-nodo. La operación consiste en asignar una página de memoria para contener los datos. Esta asignación se efectúa por la función `__get_free_page`.

El campo `i_op`, que contiene la lista de operaciones asociadas al i-nodo, se fija a `pipe_inode_operations`. Esta variable contiene de modo predeterminado un puntero a la variable `rdwr_pipe_fops` para la creación de una tubería anónima. Pero en función del tipo de tubería creada y de las opciones de creación, puede contener punteros a otras operaciones.

La creación del i-nodo termina por las inicializaciones de las características de la tubería, es decir, el posicionamiento de los derechos de acceso, de las fechas, etc.

4.2 Creación de una tubería con nombre

La creación de una tubería con nombre se efectúa por la función `fifo-open`, definida en el archivo fuente `fs/fifo.c`. Esta función se llama mediante la función de inicialización de una tubería con nombre: `init_fifo`.

La creación de una tubería con nombre depende de las opciones de creación. Las operaciones autorizadas sobre el i-nodo se fijan en función de estas opciones.

- Lectura exclusiva: `connecting_fifo_fops` en el caso de una lectura sin bloqueo, y `read_fifo_fops` en caso contrario;
- escritura exclusiva: `write_fifo_fops`;
- lectura y escritura: `rdwr_fifo_fops`.

4.3 Creación de una tubería anónima

Una tubería anónima se crea más simplemente con la función `do_pipe` del archivo *fs/pipe.c*. Esta llamada debe construir una tabla de dos descriptors de archivos que corresponden a la entrada y a la salida de la tubería anónima.

La creación de la tubería anónima consiste en un primer momento en crear un i-nodo para la tubería mediante la función `get_pipe_inode` descrita anteriormente. Seguidamente, se efectúa una asignación de dos descriptors de archivos mediante la función `get_empty_filp`.

Finalmente, el descriptor de archivo de índice 0 recibe `read_pipe_fops`, porque está dedicado a la lectura, y el descriptor de índice 1 recibe `write_pipe_fops`.

4.4 Operaciones de entradas/salidas

Las lecturas y las escrituras se efectúan en una memoria intermedia de tamaño `PIPE_BUF`. En función de las lecturas y escrituras de datos en la tubería, la ventana que contiene los datos de la tubería se desplaza de manera circular en el interior de la memoria intermedia.

Todas las operaciones de entradas/salidas, ya se trate de tuberías con nombre o anónimas, se encuentran en el archivo *fs/pipe.c*. Veamos los detalles y su funcionamiento:

- `pipe_read`: recorre la página de memoria de la tubería copiando los datos contenidos en la memoria intermedia destinada al usuario. Durante esta operación, la tubería se bloquea.

En el caso en que la lectura sea sin bloqueo, se devuelve el error `EAGAIN` si la tubería está bloqueada o vacía. En caso contrario, el proceso que llama se queda en espera mientras la tubería esté vacía.

Una vez haya datos disponibles, la tubería se bloquea, los datos se copian en la memoria intermedia proporcionada por el proceso que llama, y la tubería se desbloquea.

- `pipe_write`: escritura en la tubería de datos pasados como parámetro en la memoria intermedia, dentro del límite del tamaño restante disponible. Mientras dura esta operación, los accesos a la tubería están bloqueados.

Esta función efectúa un bucle hasta escribir todos los datos en la tubería.

En cada pasada por el bucle, comprueba si quedan procesos lectores de la tubería. Si es así, se envía la señal `SIGPIPE` al proceso que llama. En caso contrario, la tubería se bloquea, los datos se copian desde la memoria intermedia proporcionada por quien llama, y la tubería se desbloquea.

- `pipe_ioctl`: la única operación de tipo *ioctl* permitida es la designada por `FIONREAD`, que permite recuperar el número de bytes de datos almacenados actualmente en la tubería. Se devuelve el contenido del campo `len` de la estructura `pipe_inode_info`.
- `pipe_select`: para la operación de multiplexado asociada a la llamada *select*, esta función verifica ciertas condiciones de validez antes de indicar si es necesario volver a las entradas o salidas sobre la tubería:
 - multiplexado en lectura: es necesario que la tubería no esté vacía y que al menos un proceso tenga acceso a la tubería en escritura;
 - multiplexado en escritura: es necesario que la tubería no esté llena y que al menos un proceso tenga acceso a la tubería en lectura;
 - excepciones: la tubería debe ser accesible al menos por un proceso en lectura y en escritura.
- `connect_read`: esta función se utiliza únicamente con las tuberías con nombre, cuando, al crear una tubería, ningún proceso ha abierto la tubería en escritura. Sólo modifica las operaciones realizables sobre el i-nodo asignando el campo `f_op` a `read_fifo_fops`. Tras esta modificación, se lanza una llamada a la función de lectura en una tubería (`pipe_read`).
- `connect_select`: en el mismo contexto que la función anterior, esta función sólo modifica el campo `f_op` en el caso del multiplexado en lectura.

Se utilizan estas funciones al cerrar descriptores:

- `pipe_read_release`: decrementa el número de procesos con acceso en lectura a la tubería;
- `pipe_write_release`: decrementa el número de procesos con acceso en escritura a la tubería;

- `pipe_rdwr_release`: decrementa el número de procesos con acceso en lectura y escritura a la tubería.

Se utilizan estas funciones en la creación de un nuevo proceso que hereda la tubería:

- `pipe_read_open` incrementa el número de procesos con acceso en lectura a la tubería;
- `pipe_write_open` incrementa el número de procesos con acceso en escritura a la tubería;
- `pipe_rdwr_open` incrementa el número de procesos con acceso en lectura y escritura a la tubería.

Capítulo 11

IPC System V

Primitivas detalladas

**msgctl, msgget, msgsnd, msgrcv,
semctl, semget, semop, shmat,
shmctl, shmget, shmdt**

Este capítulo está dedicado a uno de los mecanismos clave del sistema Unix: los IPC (*Inter Process Communication*), que permiten intercambiar y compartir datos, pero también sincronizar procesos entre sí, con relativa facilidad de uso.

Históricamente, los IPC aparecieron en las versiones System V de Unix, y actualmente se encuentran en todas las variantes de Unix.

1 Conceptos básicos

1.1 Introducción

En los capítulos anteriores, se han abordado las tuberías y las señales. Estos conceptos son limitados en términos de comunicación. Por ejemplo, un proceso cualquiera no puede leer o escribir en una tubería a menos que sea el hijo del proceso que la ha creado. Asimismo, en la gestión de las señales, el único mensaje vehiculado es un simple número, lo que hace inútiles las señales para una transferencia de datos.

Los IPC permiten manipular datos de todo tipo, enviarlos de un proceso a otro, o bien compartirlos. En realidad, los IPC están constituidos por tres mecanismos:

1. Colas de mensajes:

Una cola de mensajes puede considerarse como un buzón. En otras palabras, una aplicación, siempre que tenga los derechos necesarios, puede depositar en ella un mensaje (un número, una cadena de caracteres o incluso el contenido de una estructura de datos cualquiera), y otras aplicaciones pueden leer dicho mensaje.

2. Memoria compartida:

La gestión de memoria compartida permite poner en común una zona de memoria entre varias aplicaciones. Normalmente, cuando se asigna una zona de memoria (con *malloc*, por ejemplo), ésta es local al proceso, lo que significa que ninguna otra aplicación funcionando en el sistema puede acceder a ella. La gestión de memoria compartida permite crear zonas de memoria accesibles por varios procesos tanto en lectura como en escritura.

3. Semáforos:

Los semáforos permiten regular uno de los mayores problemas que se presentan en un sistema multitarea: la sincronización de procesos. En efecto, varios procesos funcionan al mismo tiempo y pueden acceder a los mismos datos. Pueden producirse ciertos problemas, como accesos concurrentes a los datos, bloqueos cruzados, etc. Por ello, los semáforos se han implementado para resolver estos problemas de sincronización.

Es útil destacar una gran diferencia entre los demás sistemas de comunicación y los IPC: estos últimos no utilizan el sistema de gestión de archivos. Cuando se crea o manipula un IPC, no se usa como un archivo y, por tanto, no sirven instrucciones como *open*, *read*, etc. La única manera de manipular un IPC es conociendo su clave de identificación.

1.2 La gestión de claves

Los IPC se basan en un mecanismo bastante peculiar: *la gestión de claves*. Para crear un IPC, o simplemente para acceder a él, hay que contar con un identificador, llamado *clave*. Esta clave es un número que identifica el IPC de manera única en todo el sistema, y es necesario para acceder al recurso.

Para generar una clave se utiliza la función de biblioteca *ftok*.

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

key_t ftok (char *pathname, char proj);
```

Esta función toma como argumento un nombre de archivo. El segundo parámetro permite generar para un mismo archivo varias claves diferentes. A partir del nombre de archivo, devuelve el identificador único del IPC. Este identificador es el que deberá usarse para gestionar el IPC.

El número se calcula con la fórmula siguiente:

$$\text{clave} = (\text{st_ino} \& 0X\text{ffff}) | ((\text{st_dev} \& 0xFF) \ll 16) | (\text{proj} \ll 24)$$

El cálculo se realiza por una combinación entre el número del i-nodo del archivo, el número del dispositivo en el que se encuentra el archivo y el último parámetro, de tal manera que siempre se genera un número único.

Veamos un ejemplo de utilización de esta función:

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

void uso(char *prg)
{
    printf (" %s nombre_archivo_existente \n", prg);
    exit (1);
}

void main (int argc, char **argv)
{
    struct stat datos;
    if (argc != 2)
        uso (argv[0]);
    if (stat (argv[1], &datos) == -1) {
        printf ("Imposible acceder a %s \n", argv[1]);
        exit (2);
    }
    printf ("Número del i-nodo del archivo: %x \n",
            (int) datos.st_ino);
    printf ("Clave con 0: %x \n", (int) ftok (argv[1], '\0'));
    printf ("Clave con a: %x \n", (int) ftok (argv[1], 'a'));
}
```

Veamos un ejemplo de su ejecución:

```
gandalf> % CrearClave zorglub.c
Número del i-nodo del archivo: 38e6
Clave con 0: 238e6
Clave con a: 610238e6
```

1.3 Los derechos de acceso

Como toda zona de memoria, archivo u otro objeto del sistema Unix, un IPC posee derechos de acceso para asegurar la confidencialidad de las informaciones transmitidas. Este sistema de derechos autoriza por ejemplo el acceso a una cola de mensajes a un grupo particular de usuarios.

Cada IPC posee una estructura `ipc_perm` que contiene los datos relativos a los permisos. Es posible modificar estos permisos cuando se crea el IPC. Veamos en detalle esta estructura:

tipo	campo	descripción
key_t	key	Clave única
ushort	uid	Identificador de usuario efectivo del propietario
ushort	gid	Identificador de grupo efectivo del propietario
ushort	cuid	Identificador de usuario efectivo del creador
ushort	cgid	Identificador de grupo efectivo del creador
ushort	mode	Derechos de acceso
ushort	seq	Número de usuarios de la entrada

Encontramos los campos estándar que identifican al propietario, al creador del recurso y los derechos de acceso. Estos derechos son importantes porque permiten a los demás procesos poder leer o escribir datos.

2 Llamadas al sistema

La manipulación de los IPC se efectúa mediante llamadas al sistema. Aunque los tres tipos de IPC son muy diferentes, utilizan llamadas que se parecen mucho. Se pueden agrupar en tres categorías:

- Creación: `msgget`, `semget`, `shmget`
- Control: `msgctl`, `semctl`, `shmctl`
- Comunicación: `msgsnd`, `msgrcv`, `semop`, `shmop`

Se utilizan frecuentemente siete constantes:

tipo de llamadas	significado	constante
Creación	Nueva cola Creación de un IPC si la clave no se usa ya Creación de un objeto si no existe Utilizado para los módulos cargables	IPC_PRIVATE IPC_CREAT IPC_EXCL IPC_KERNELD IPC_NOWAIT
msgsnd, msgrcv, semop	Sin espera	
Control	Supresión de recursos del objeto Fija opciones en ipc_perm Lectura de ipc_perm Devuelve una estructura msginfo, o shminfo, seminfo	IPC_RMID IPC_SET IPC_GET IPC_INFO

2.1 Colas de mensajes

Las colas de mensajes se comparan generalmente con un sistema de buzones. El principio es relativamente simple: un proceso deposita uno o más mensajes en un «buzón». Otro proceso (o varios) puede leer cada uno de los mensajes, en el orden de su llegada, según el tipo de mensajes que desea. Es exactamente como cuando se recoge el correo: se puede preferir empezar por leer por la parte baja de la pila, por arriba, por las facturas o por las postales.

Para manipular una cola de mensajes, además de las llamadas al sistema, se necesitan tres estructuras.

2.1.1 Las estructuras básicas

La estructura `msgqid_ds` corresponde a un objeto de la cola de mensajes. Mediante esta estructura es posible manipular el objeto creado. Está definida en el archivo de cabecera `<linux/msg.h>`, pero por razones de portabilidad basta con incluir el archivo de cabecera `<sys/msg.h>`. He aquí una descripción de esta estructura:

tipo	campo	descripción
struct ipc_perm	msg_perm	Derechos de acceso del objeto
struct msg *	msg_first	Puntero al primer mensaje en la cola
struct msg *	msg_last	Puntero al último mensaje en la cola
time_t	msg_stime	Fecha de la última llamada a <code>msgsnd</code>
time_t	msg_rtime	Fecha de la última llamada a <code>msgrcv</code>
time_t	msg_ctime	Fecha de la última modificación del objeto
struct wait_queue *	wwait	Cola de procesos en espera de escritura
struct wait_queue *	rwait	Cola de procesos en espera de lectura
ushort	msg_cbytes	Número de bytes actualmente en la cola
ushort	msg_qnum	Número de mensajes en la cola
ushort	msg_qbytes	Número máximo de bytes en la cola
ushort	msg_lspid	Número del último proceso que ha efectuado un <code>msgsnd</code>
ushort	msg_lrpid	Número del último proceso que ha efectuado un <code>msgrcv</code>

Este archivo de cabecera contiene también la definición de la estructura `msginfo`, que se utiliza en una llamada a `msgctl` con `IPC_INFO` como argumento. Esta estructura se utiliza en particular en programa `ipcs`, y está reservada especialmente a programas del sistema de estadísticas o de observación de la máquina.

La estructura está constituida por los campos siguientes:

tipo	campo	descripción
int	<code>msgpool</code>	Tamaño en kilobytes de los datos en la cola
int	<code>msgmap</code>	Número de entradas en la tabla de mensajes
int	<code>msgmax</code>	Tamaño máximo de los mensajes (en bytes)
int	<code>msgmnb</code>	Tamaño máximo de la cola de mensajes
int	<code>msgmni</code>	Número máximo de identificadores de colas de mensajes
int	<code>msgsz</code>	Tamaño de segmento de mensaje
int	<code>msgtql</code>	Número de cabeceras de mensajes del sistema
ushort	<code>msgseg</code>	Número máximo de segmentos

La estructura `msgbuf` almacena un mensaje y su tipo. Corresponde al modelo a utilizar para el envío y la recuperación de un mensaje en una cola:

tipo	campo	descripción
long	<code>mtype</code>	Tipo del mensaje
char[1]	<code>mtext</code>	Contenido del mensaje

Esta estructura no se usa nunca en las aplicaciones. En realidad, toda estructura de datos depositada en una cola de mensajes debe tener necesariamente como primer campo el tipo del mensaje. Éste es un número estrictamente positivo, de tipo `long`, que permitirá la selección de un mensaje en la cola en función de su tipo.

Las colas de mensajes se utilizan también por el núcleo en el marco de módulos cargables, y de *kerneld* (véase el capítulo 12, sección 4.1).

2.1.2 Creación y búsqueda de colas de mensajes

La llamada al sistema `msgget` cumple dos funciones:

1. creación de una nueva cola de mensajes;
2. búsqueda de una cola de mensajes existente (creada por otra aplicación, por ejemplo) mediante su clave.

En ambos casos, sólo se puede usar si se posee una clave. El prototipo de esta llamada al sistema es el siguiente:

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

```
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key_t clave, int opcion);
```

El primer argumento corresponde a la clave de la cola de mensajes que ya existe, o de la que se desea crear. Si se pasa como clave el valor `IPC_PRIVATE`, se crea una cola. Si se pasa una clave diferente, se presentan dos posibilidades:

1. La clave no es utilizada por otra cola de mensajes: en este caso, es necesario indicar `IPC_CREAT` como opción. En este caso se creará la cola, con la clave pasada como parámetro. Las opciones pueden fijar ciertos derechos de acceso.
2. La clave se utiliza en una cola de mensajes. Es necesario que se pase `IPC_CREAT` o bien `IPC_EXCL` como parámetro. En este caso, a continuación se puede leer o escribir en la cola de mensajes, si los derechos de acceso lo permiten.

Si todo ocurre correctamente, `msgget` devuelve el identificador de la cola de mensajes. En caso contrario, `errno` posee el valor siguiente:

error	significado
EACCES	Existe una cola de mensajes para la clave, pero el proceso que llama no tiene derechos de acceso sobre la cola de mensajes
EXIST	Existe ya una cola de mensajes para la clave, y se han fijado las opciones <code>IPC_CREAT</code> e <code>IPC_EXCL</code>
ETIMR	La cola de mensajes está marcada como destinada a destruirse
ENOENT	No existe ninguna cola de mensajes para la clave, y la opción <code>IPC_CREAT</code> no se ha indicado
ENOMEM	Se habría podido crear una cola de mensajes, pero el sistema no tiene suficiente memoria para la nueva estructura
ENOSPC	Se ha alcanzado el número máximo de colas de mensajes

Esta llamada devuelve el identificador de la cola de mensajes.

2.1.3 Control de las colas de mensajes

Tras haber creado una cola de mensajes, es posible manipularla modificando por ejemplo los permisos de acceso a la cola. Ante todo, hay que saber que los IPC se gestionan en el núcleo por la tabla de colas de mensajes. La llamada al sistema `msgctl` permite acceder y modificar ciertos campos de esta tabla para las colas a las que se tiene acceso.

El prototipo de esta llamada es:

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



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

```
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

El primer argumento *msqid* corresponde al identificador de la cola de mensajes dados por la llamada a *msgget*. El segundo argumento, *cmd*, indica el tipo de la operación a efectuar sobre la cola de mensajes. Ésta es la lista de operaciones que se pueden realizar sobre la cola de mensajes:

- **MSG_STAT** (o **IPC_STAT**): copia la tabla asociada a la cola de mensajes en la dirección apuntada por *buf* de tipo estructura *msqid_ds*.
- **IPC_SET**: permite fijar ciertos miembros de la estructura *msqid_ds*. Esta operación actualiza automáticamente el campo *msg_ctime* que conserva la fecha de la última modificación de la entrada en la tabla de procesos. Es posible acceder a tres campos de la estructura: *msg_perm.uid*, *msg_perm.gid* y *msg_perm.mode*. Puede modificarse un campo suplementario, pero únicamente por parte del superusuario: *msg_qbytes*.
- **IPC_RMID**: permite destruir la cola de mensajes y los datos que contiene. Sólo puede destruir una cola de mensajes un proceso cuyo identificador de usuario efectivo corresponda al superusuario, al creador o al propietario de la cola de mensajes.
- **MSG_INFO** (o **IPC_INFO**): llena la estructura *struct msginfo* pasada como parámetro. Esto se usa por ejemplo en el programa *ipcs*.

2.1.4 Emisión de mensajes

La llamada al sistema *msgsnd* permite enviar un mensaje a una cola de mensajes. Su prototipo es el siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int msgsnd (int msqid, struct msgbuf *msgp, int msgsize, int
            msgopt);
```

Para que esta operación se desarrolle normalmente, es necesario haber obtenido el identificador de la cola de mensajes, pero también poseer los derechos necesarios para escribir en la cola de mensajes. El segundo parámetro corresponde a los datos que se quiere enviar a la cola de mensajes. El parámetro *msgsize* corresponde al tamaño del objeto que se envía a la cola.

Si se pasa como opción `IPC_NOWAIT`, sólo tiene efecto cuando la cola está llena. En este caso, la llamada no es bloqueadora (a diferencia de una llamada sin este parámetro) y se devuelve el error `EAGAIN`.

Esta es la lista de errores que puede devolver esta llamada:

<i>error</i>	<i>significado</i>
EAGAIN	No puede enviarse el mensaje porque la cola de espera está llena y la opción IPC_NOWAIT ha sido activada
EACCES	No hay derechos de escritura
EFAULT	La dirección apuntada por <i>msgp</i> no es accesible
EINVAL	La cola ya no existe: ha sido destruida
ENINTR	El proceso ha recibido una señal y por tanto la llamada al sistema ha fallado
EINVAL	Identificador erróneo de la cola, o el tipo del dato a enviar a la cola de mensajes no es positivo
ENOMEM	No hay suficiente memoria para realizar una copia del objeto

Esta llamada devuelve el valor 0 en caso de éxito.

2.1.5 Recepción de mensajes

La llamada *msgrcv* permite leer en la cola de mensajes, y posee el prototipo siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgrcv (int msqid, struct msgbuf *msgp, int msgsz, long
            msgtyp, int msgflg);
```

Esta llamada al sistema permite, si los derechos del proceso son suficientes, recuperar un mensaje que se copiará en *msgp*. La zona de memoria apuntada por *msgp* tiene como tamaño máximo *msgsz*.

Pueden darse dos tipos de opciones en el campo *msgflg*:

- **MSG_NOERROR**: si el tamaño del mensaje es superior al tamaño especificado en el campo *msgsz*, y si la opción **MSG_NOERROR** está posicionada, el mensaje se truncará. La parte sobrante se pierde. En el caso en que no esté esta opción, el mensaje no se retira de la cola y la llamada fracasa devolviendo como error **E2BIG**.
- **IPC_NOWAIT**: esta opción permite evitar la espera activa. Si la cola no está nunca vacía, se devuelve el error **ENOMSG**. Si esta opción no está activa, la llamada se suspende hasta que un dato del tipo solicitado entre en la cola de mensajes.

El tipo del mensaje a leer debe especificarse en el campo (*msgtyp*):

- Si *msgtyp* es igual a 0, se lee el primer mensaje de la cola, es decir, el mensaje más antiguo, sea cual sea su tipo.
- Si *msgtyp* es **negativo**, entonces se devuelve el primer mensaje de la cola con el tipo menor, inferior o igual al valor absoluto de *msgtyp*.
- Si *msgtyp* es **positivo**, se devuelve el primer mensaje de la cola con un tipo estrictamente igual a *msgtyp*. En el caso en que esté presente la opción *MSG_EXCEPT*, se devolverá el primer mensaje con un tipo diferente.

Estos son los errores que pueden resultar del uso de esta llamada al sistema:

<i>error</i>	<i>significado</i>
EINVAL	El identificador de cola de mensajes no es válido, <i>mtype</i> es inferior a cero, o bien <i>msgsz</i> es inferior a cero o mayor que el tamaño máximo de un mensaje <i>MSGMAX</i>
EFAULT	La zona de memoria apuntada por <i>msgp</i> no es accesible
ETIMED	El proceso esperaba un mensaje y la cola de mensajes se ha destruido
ENACCS	El proceso no tiene los derechos necesarios para acceder a la cola de mensajes
EINVAL	El tamaño del mensaje es mayor que <i>msgsz</i> y no se ha activado la opción <i>MSG_NOERROR</i>
ENOMSG	Se ha activado la opción <i>IPC_NOWAIT</i> y no se ha encontrado ningún mensaje en la cola
EINTR	El proceso esperaba un mensaje y ha recibido una señal

2.1.6 Un ejemplo de uso

Tras haber detallado las diferentes llamadas al sistema, le proponemos un ejemplo simple de uso. El principio es crear un servidor que copiará en una misma cola de mensajes dos archivos pasados en la línea de mandatos. Se crean dos clientes, que recuperan un archivo cada uno, y lo guardan. La figura 11.1 resume el ejemplo que seguirá.

El archivo *Copia.h* contiene las declaraciones comunes, el servidor se implementa en el archivo *CopiaServidor.c* y el archivo *CopiaCliente.c* contiene el código fuente del cliente.

```
/* Copia.h */

#ifndef COPIA_H
#define COPIA_H

/* Nombre del archivo utilizado para generar la clave de */
/* la cola de mensajes */
#define ARCHIVO_CLAVE "CopiaServidor"
#define PROJ_FTOK 'a' /* parámetro ftok */
/* Tipo del archivo 1 en la cola de mensajes */
```

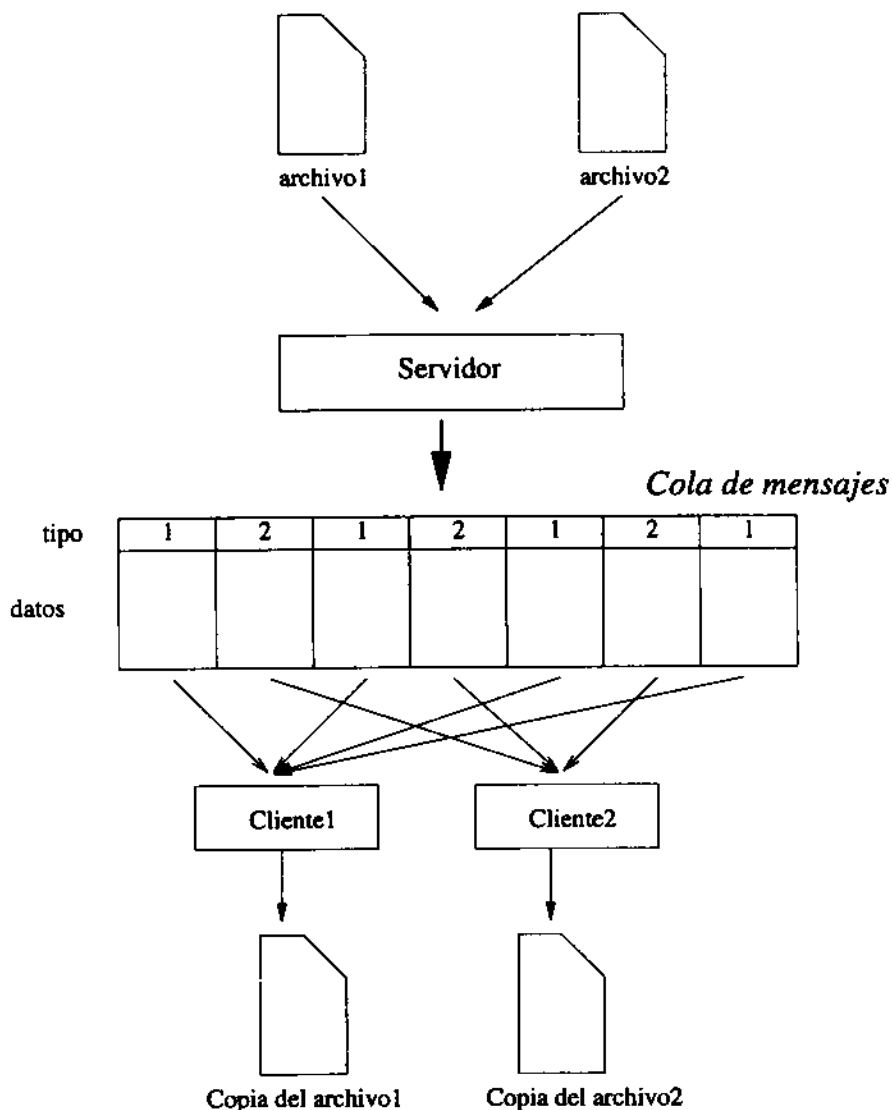


FIG. 11.1 – Ejemplo de manipulación de archivos de mensajes

```
#define TYPE_FIC1 1
/* Tipo del archivo 2 en la cola de mensajes */
#define TYPE_FIC2 2
/* El cliente 1 ha terminado de leer el archivo */
#define TYPE_END1 3
/* El cliente 2 ha terminado de leer el archivo */
#define TYPE_END2 4
#define SIZE_BUF 256      /* Tamaño de los bloques */
```

```
typedef struct data_s data_t;
struct data_s {
    long    tipo;           /* tipo del mensaje */
    int     tam;           /* tamaño de los datos transferidos
                           -1 == EOF */
    char    buf[SIZE_BUF]; /* mensaje */
};

#endif

/* CopiaServidor.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "Copia.h"

static void uso (char *prg)
{
    printf ("Ejemplo de aplicación con colas de mensajes\n");
    printf (" --> Servidor\n");
    printf (" %s <archivo1> <archivo2> \n\n", prg);
    exit (-1);
}

static void unError (char *msg, char *llamada)
{
    fprintf (stderr, "Error Servidor %s :%d \n", msg, errno);
    perror (llamada);
    exit (-1);
}

static void close_all (int id_file, int fd1, int fd2)
{
    msgctl (id_file, IPC_RMID, NULL);
    close (fd1);
    close (fd2);
}
```

```
void main (int argc, char **argv)
{
    int fd1, fd2;
    int id_file;
    key_t clave;
    data_t data1, data2;
    int end1 = 1, end2 = 1;

    if (argc != 3)
        uso (argv[0]);

    /* Apertura de archivos */
    if ((fd1 = open (argv[1], O_RDONLY)) == -1)
        unError ("Error en open1", "open");

    if ((fd2 = open (argv[2], O_RDONLY)) == -1) {
        close (fd1);
        unError ("Error en open2", "open");
    }

    /* Generación de la clave */
    if ((clave = ftok (FICHER_CLEF, PROJ_FTOK)) == -1) {
        close (fd1);
        close (fd2);
        unError ("Error en ftok", "ftok");
    }

    /* Creación de la cola de mensajes */
    if ((id_file = msgget
        (clave, IPC_CREAT | IPC_EXCL | 0666)) == -1) {
        close (fd1);
        close (fd2);
        unError ("Error en msgget", "msgget");
    }

    printf
        ("Servidor: Cola de mensaje(%d) creada: copia de %s y %s \n",
         id_file, argv[1], argv[2]);
    data1.tipo = TYPE_FIC1;
    data2.tipo = TYPE_FIC2;
    data1.tam = data2.tam = -1;

    /* Bucle de copia */
    while ((data1.tam) || (data2.tam)) {
        if (data1.tam) {
            data1.tam = read (fd1, data1.buf,
                             SIZE_BUF * sizeof (char));
            if (data1.tam == -1) {
                close_all (id_file, fd1, fd2);
            }
        }
    }
}
```

```

        unError ("Error durante lectural", "read");
    }
    if (msgsnd (id_file, (struct msgbuf *) &data1,
                sizeof (data_t), 0) == -1)
        unError ("Error durante envío de fic1", "msgsnd");
    }
    if (data2.tam) {
        data2.tam = read (fd2, data2.buf,
                          SIZE_BUF * sizeof (char));
        if (data2.tam == -1) {
            close_all (id_file, fd1, fd2);
            unError ("Error durante lectura2", "read");
        }
        if (msgsnd (id_file, (struct msgbuf *) &data2,
                    sizeof (data_t), 0) == -1)
            unError ("Error durante envío fic2", "msgsnd");
        }
    }

    printf
    ("Servidor: Envío a la cola de mensajes terminado\n");
    printf
    ("Servidor: Esperando que los clientes terminen de leer.\n");
    /* Se espera que los hijos digan que han terminado */
    while ((end1) || (end2)) {
        struct msgbuf  rec;
        if (msgrcv (id_file, &rec, sizeof (struct msgbuf),
                    TYPE_END1, IPC_NOWAIT) != -1)
            end1 = 0;
        if (msgrcv (id_file, &rec, sizeof (struct msgbuf),
                    TYPE_END2, IPC_NOWAIT) != -1)
            end2 = 0;
        }
    printf
    ("Servidor: Desconexión de los dos hijos realizada\n");
    close_all (id_file, fd1, fd2);
}

/* CopiaCliente.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <sys/stat.h>

```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include "Copia.h"

static void uso (char *prg)
{
    printf ("Ejemplo de aplicación con colas de mensajes\n");
    printf (" --> Cliente \n");
    printf (" %s 1|2 <archivo_copia> \n\n", prg);
    exit (-1);
}

static void unError (char *msg, char *llamada, int no_client)
{
    fprintf (stderr, "Error Cliente%d %s :%d \n",
             no_client, msg, errno);
    perror (llamada);
    exit (-1);
}

static void desconexion (int id_file, int no_client)
{
    struct msgbuf desconexion;

    if (no_client == 1)
        desconexion.mtype = TYPE_END1;
    else
        desconexion.mtype = TYPE_END2;
    msgsnd (id_file, &desconexion, sizeof (struct msgbuf), 0);
}

static void close_all (int id_file, int fd, int no_client)
{
    desconexion (id_file, no_client);
    msgctl (id_file, IPC_RMID, NULL);
    close (fd);
}

void main (int argc, char **argv)
{
    int fd;
    int id_file;
    int ret;
    key_t clave;
    data_t data;
```



```
int no_client; /* Cliente 1 o 2 */
int tipo_cliente; /* Identificación del cliente
                  para el servidor */

if (argc != 3)
    uso (argv[0]);

/* Recuperación número de cliente */
no_client = atoi (argv[1]);
if ((no_client != 1) && (no_client != 2))
    uso (argv[0]);

if (no_client == 1)
    tipo_cliente = TYPE_FIC1;
else
    tipo_cliente = TYPE_FIC2;

/* Generación de la clave */
if ((clave = ftok (ARCHIVO_CLAVE, PROJ_FTOK)) == -1)
    unError ("Error en ftok", "ftok", no_client);

/* Conexión a la cola de mensajes */
if ((id_file = msgget (clave, IPC_EXCL)) == -1)
    unError ("Error en msgget", "msgget", no_client);

fprintf
    (stderr, "Cliente%d conectado a la cola de mensajes\n",
     no_client);

/* Apertura del archivo*/
if ((fd = open (argv[2], O_WRONLY | O_CREAT, 0666)) == -1)
    unError ("Error en open", "open", no_client);

/* Bucle de lectura */
while (1) {
    if (msgrcv (id_file, (struct msgbuf *) &data,
               sizeof (data_t), tipo_cliente, 0) == -1)
        unError ("Error durante recepción", "msgrcv", no_client);

    if (data.tam == 0) /* EOF */
        break;
    ret = write (fd, data.buf, data.taille * sizeof (char));
    if (ret == -1) {
        close_all (id_file, fd, no_client);
        unError("Error durante la escritura", "write", no_client);
    }
}
```

```
    }  
    close (fd);  
    printf ("Cliente%d: Desconexión\n", no_client);  
    desconexion (id_file, no_client);  
}
```

A continuación se presenta un ejemplo de uso.

```
gandalf> CopiaServidor CrearClave.c Copia.h &  
Servidor: Cola de mensajes creada: copia de CrearClave.c y  
          Copia.h  
  
gandalf> CopiaCliente 1 archiv01 &  
Cliente1 conectado a la cola de mensajes  
  
gandalf> CopiaCliente 2 archiv02 &  
Cliente2 conectado a la cola de mensajes  
  
Servidor: Envío a la cola de mensajes terminado  
Servidor: Esperando que los clientes finalicen la lectura...  
Cliente2: Desconexión  
Cliente1: Desconexión  
Servidor: Desconexión de los dos hijos realizada
```

2.2 Semáforos

Linux utiliza semáforos de manera interna, para sincronizar los procesos en modo núcleo (véase el capítulo 4, sección 5.3.3). También ofrece a los procesos funciones de manipulación de semáforos.

En ciertos casos, un proceso necesita poseer varios recursos para proseguir su acción, por ejemplo deberá acceder tal vez a una memoria intermedia de datos y a un segmento de memoria compartida si desea desplazar los datos de un lugar a otro. Será necesario que utilice dos semáforos y ejecute dos operaciones P (una para la memoria intermedia y otra para la memoria) a fin de poder disponer de los dos recursos. Esta situación puede provocar un bloqueo cruzado en el caso siguiente: un proceso posee el acceso exclusivo a la memoria intermedia y desea acceder a la memoria mientras que otro posee el acceso exclusivo a la memoria y desea utilizar la memoria intermedia.

Para remediar este problema, las operaciones P y V no se efectúan con un solo semáforo sino con una tabla de semáforos. Así, un proceso puede proseguir su ejecución únicamente si las operaciones se han desarrollado correctamente en todos los elementos de la tabla. Desde el punto de vista del usuario, esto debe realizarse, evidentemente, de manera atómica, es decir, en una sola llamada al sistema.

Los semáforos de System V implementan todas estas funcionalidades con extensiones que permiten esperar que un semáforo esté a cero.

2.2.1 Las estructuras básicas

Estas estructuras se definen en el archivo de cabecera `<linux/sem.h>` pero como con las colas de mensajes, basta con incluir el archivo `<sys/sem.h>`. Cuando se crea un semáforo System V, se crea un grupo de semáforos. La estructura `semid_ds` es la estructura de control para un grupo de semáforos. Contiene información del sistema, punteros a las operaciones a realizar sobre el grupo, y un puntero hacia las estructuras `sem` que se almacenan en el núcleo y contienen las informaciones de cada semáforo.

tipo	campo	descripción
struct ipc_perm	sem_perm	Permisos
time_t	sem_otime	Fecha de la última operación
time_t	sem_ctime	Fecha del último cambio por <code>semctl</code>
struct sem*	sem_base	Puntero al primer semáforo del grupo
struct sem_queue *	sem_pending	Operaciones en espera de realización
struct sem_queue **	sem_pending_last	Última operación en espera
struct sem_undo *	undo	Operaciones anuladas en caso de terminación
ushort	sem_nsems	Número de semáforos del grupo

La estructura `sembuf` corresponde a una operación sobre un semáforo (incrementar, decrementar o esperar un valor nulo), y por tanto se usa en la llamada `semop`. Contiene los campos siguientes:

tipo	campo	descripción
ushort	sem_num	Número de semáforo en el grupo
short	sem_op	Operación sobre el semáforo
short	sem_flg	Opciones

La estructura `semun` es una unión, se utiliza en la llamada `semctl` para almacenar o recuperar informaciones sobre los semáforos.

tipo	campo	descripción
int	val	Valor para <code>SETVAL</code>
struct semid_ds *	buf	Memoria de datos para <code>IPC_STAT</code> e <code>IPC_SET</code>
ushort *	array	Tabla para <code>GETALL</code> y <code>SETALL</code>
struct seminfo *	__buf	Memoria de datos para <code>IPC_INFO</code>
void *	__pad	Puntero de alineación de la estructura

Como se puede constatar, se puede modificar el valor del semáforo: el signo del campo `val` es quien define la acción:

- si `val` es negativo, es una operación P;

- si *val* es positivo, es una operación V;
- si *val* es nulo, el proceso se bloquea hasta que el valor del semáforo sea nulo (esta particularidad se utiliza por ejemplo para concertar encuentros entre procesos).

Encontramos también la definición de la estructura *seminfo* que permite conocer los valores límite o actuales del sistema mediante una llamada a *semctl*. Estas llamadas no se realizan, generalmente, de modo directo, sino que están reservadas a las utilidades del sistema como el mandato *ipcs* (véase la sección 3.2.1). En Linux, sólo cuatro campos tienen un valor significativo por llamada a *semctl* con *IPC_INFO* como argumento y únicamente dos campos son significativos en una llamada con *SEMINFO*. La tabla siguiente describe los campos significativos de la estructura:

tipo	campo	descripción
int	<i>semmap</i>	No se usa
int	<i>semnmi</i>	Número máximo de grupos de semáforos (<i>SEMMSL</i> =128) (<i>IPC_INFO</i>)
int	<i>semmns</i>	Número máximo de semáforos (<i>SEMMSL</i> = <i>SEMNI</i> * <i>SEMMSL</i>) (<i>IPC_INFO</i>)
int	<i>semmnu</i>	No se usa
int	<i>semmsl</i>	Número máximo de semáforos por grupo (<i>SEMMSL</i> =32) (<i>IPC_INFO</i>)
int	<i>semopm</i>	No se usa
int	<i>semume</i>	No se usa
int	<i>semusz</i>	Número de grupos de semáforos actualmente definidos (<i>SEMINFO</i>)
int	<i>semvmx</i>	Valor máximo del contador de semáforos (<i>SEMVMX</i> =32767) (<i>IPC_INFO</i>)
int	<i>semaem</i>	Número de semáforos actualmente definidos (<i>SEMINFO</i>)

2.2.2 Creación y búsqueda de grupos de semáforos

La llamada al sistema *semget* permite la creación de un grupo de semáforos, o bien la recuperación del identificador de un grupo ya existente. Esta función se comporta como la función *msgget* pero para una tabla de semáforos. Posee un argumento suplementario que indica el número de semáforos a crear en la tabla. Hay que destacar que los semáforos se numeran a partir de cero en el grupo. El prototipo de la función es el siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget (key_t key, int nsems, int semflg);
```

Esta llamada devuelve el identificador de un grupo de semáforos o bien el valor -1 si se produce un error. En este último caso, la variable *errno* toma uno de los valores siguientes:

<i>error</i>	<i>significado</i>
EACCES	Existe un grupo de semáforos para la clave, pero el proceso no tiene los derechos necesarios para acceder al grupo
EXIST	Existe un grupo de semáforos para la clave, pero están activadas las opciones IPC_CREAT e IPC_EXCL
EINVAL	El grupo de semáforos ha sido borrado
ENOENT	El grupo de semáforos no existe y no está activada la opción IPC_CREAT
ENOMEM	El semáforo podría crearse pero el sistema no tiene más memoria para almacenar la estructura
ENOSPC	El semáforo podría crearse pero se sobrepasarían los límites para el número máximo de grupos (SEMSETI) de semáforos o el número máximo de semáforos (SEMMSL)

2.2.3 Operaciones sobre los semáforos

La sección 2.2.1 presentaba tres operaciones posibles sobre un semáforo: incremento, decremento o espera de nulidad. La llamada al sistema *semop* permite realizar estas tres operaciones sobre una selección de semáforos de un grupo. El prototipo de la llamada es el siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semop (int semid, struct sembuf *sops, unsigned nsops);
```

El primer argumento de esta llamada representa el grupo sobre el cual las operaciones tendrán lugar. El segundo argumento es una tabla de estructuras *sembuf* que contiene la lista de operaciones. Finalmente, el último argumento es el número de operaciones a realizar en esta llamada (en realidad, esto da el tamaño de la tabla apuntada por el segundo argumento).

Todas estas operaciones deben efectuarse de manera atómica. En los casos en que esto no sea posible, o bien el proceso se suspende hasta que sea posible, o bien, si el indicador **IPC_NOWAIT** está activado (campo *sem_flg* de la estructura *sembuf*) por una de las operaciones, la llamada al sistema se interrumpe sin que se realice ninguna operación.

Por cada operación efectuada, el sistema controla si el indicador **SEM_UNDO** está posicionado. Si es así, crea una estructura para conservar el rastro de esta operación y poder anularla al finalizar la ejecución del proceso.

Si la llamada al sistema se desarrolla correctamente el valor devuelto es 0; si no es -1 y *errno* toma uno de los valores siguientes:

<i>error</i>	<i>significado</i>
E2BIG	El argumento <code>nsops</code> es mayor que <code>SEMOPM</code> , y se sobrepasa el número máximo de operaciones autorizadas para una llamada
EACCES	El proceso que llama no tiene los derechos de acceso a uno de los semáforos especificados en una de las operaciones
EAGAIN	El indicador <code>IPC_NOWAIT</code> está activado y las operaciones no han podido realizarse inmediatamente
EFAULT	La dirección especificada por el campo <code>sops</code> no es válida
EINVAL	El número del semáforo (campo <code>sem_num</code>) es incorrecto para una de las operaciones (negativo o superior al número de semáforos en el grupo)
ENOTDIR	El semáforo no existe
ENOTR	El proceso ha recibido una señal cuando esperaba el acceso a un semáforo
EINVAL	O el grupo del semáforo solicitado no existe (argumento <code>semid</code>), o bien el número de operaciones a realizar es negativo o nulo (argumento <code>nsops</code>)
ENOMEM	El indicador <code>SEM_UNDO</code> está activado y el sistema no puede asignar memoria para almacenar la estructura de anulación
ERANGE	El valor añadido al contador del semáforo sobrepasa el valor máximo autorizado para el contador <code>SEMVMX</code>

2.2.4 El control de los semáforos

La llamada al sistema `semctl` permite la consulta, modificación o supresión de un grupo de semáforos. También permite inicializar los semáforos y obtener información sobre el número de semáforos en espera de aumento o de nulidad. La declaración de la llamada es la siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (int semid, int semnun, int cmd, union semun arg);
```

En función del valor de `cmd`, el parámetro `semmun` representa el ordinal o bien el número de semáforos; asimismo, `arg` se interpretará de modo distinto. El parámetro `cmd` indica pues la operación a realizar, y su valor puede ser:

- **IPC_STAT**: permite obtener las informaciones respecto a un grupo de semáforos. `semmun` se ignora, `arg` es un puntero a la zona que contiene las informaciones. El proceso debe tener derechos de lectura para realizar la operación.
- **IPC_SET**: permite modificar ciertos valores de la estructura del grupo de semáforos. Los campos modificables son `sem_perm.uid`, `sem_perm.gid` y `sem_perm.mode`. El campo `sem_ctime` se actualiza automáticamente. El proceso debe ser el creador o bien el propietario del grupo, o bien el superusuario.

- **IPC_RMID**: permite destruir el grupo de semáforos. El proceso debe ser el creador o el propietario del grupo, o bien el superusuario. Todos los procesos en espera sobre uno de los semáforos del grupo se despiertan y reciben el error **EIDRM**.
- **GETPID**: permite devolver el valor de `sempid` del semáforo `semnum`. Se trata del identificador del último proceso que haya realizado la llamada al sistema `semop` sobre este semáforo. El proceso debe tener derechos de acceso en lectura sobre el grupo.
- **GETNCNT**: permite devolver el valor de `semncnt` del semáforo `semnum`. Corresponde al número de procesos que esperan un aumento del contador del semáforo. El proceso debe tener el derecho de acceso en lectura sobre el grupo.
- **GETZCNT**: permite devolver el valor de `semzcnt` del semáforo `semnum`. Corresponde al número de procesos que esperan que el contador del semáforo llegue a nulo. El proceso debe tener el derecho de acceso en lectura sobre el grupo.
- **GETVAL/SETVAL**: permite leer/posicionar el valor del semáforo `semnum` del grupo `semid`. `arg` contiene el valor leído/modificado del semáforo. En el caso de la modificación, las estructuras de anulación de operaciones se borran, los procesos en espera sobre el semáforo se despiertan si el valor es nulo o positivo, y el campo `sem_ctime` se actualiza. El proceso debe tener los derechos de lectura/escritura.
- **GETALL/SETALL**: permite leer/posicionar los valores de los semáforos del grupo `semid`. `arg` contiene el resultado/el valor de la operación: el parámetro `semnum` se ignora. El proceso que ejecuta la llamada debe tener acceso en lectura/escritura sobre el grupo.

La llamada al sistema devuelve `-1` en caso de error, y la variable `errno` contiene uno de los valores siguientes:

<i>error</i>	<i>significado</i>
EACCES	El proceso que llama no tiene derechos de acceso necesarios para realizar la operación
EFAULT	La dirección especificada por <code>arg.buf</code> o <code>arg.array</code> no es válida
EIDRM	El grupo de semáforos ha sido borrado
EINVAL	El valor de <code>semid</code> o de <code>cmd</code> es incorrecto
EPERM	El proceso no tiene los derechos necesarios para la operación (mandato IPC_SET o IPC_RMID)
ERANGE	<code>cmd</code> tiene el valor SETALL o SETVAL y el valor del contador de uno de los semáforos a modificar es inferior a cero o superior a SEMVMX

En caso de éxito, la llamada al sistema devuelve un valor positivo o nulo dependiendo del parámetro `cmd`:

GETPID	Valor de <code>sempid</code>
GETVAL	Valor de <code>semval</code>
GETNCNT	Valor de <code>semncnt</code>
GETZCNT	Valor de <code>semzcnt</code>

2.3 Memorias compartidas

Una innovación importante aportada por los IPC es la gestión de memoria compartida (*Shared Memory*). En un programa estándar, una zona de memoria asignada es propia del proceso que la ejecuta. Ningún otro proceso puede acceder a ella. El principio de la memoria compartida es permitir a los procesos compartir una parte de su espacio de direccionamiento.

Las memorias compartidas proporcionan un sistema muy potente para compartir zonas de memoria, pero obligan a implementar un mecanismo de sincronización de accesos. En efecto, no es posible que dos procesos intenten escribir al mismo tiempo y en el mismo punto de la memoria compartida: es más que probable que la coherencia de los datos se pierda. Una solución para resolver este problema es utilizar los semáforos.

La manipulación de una memoria compartida sólo puede efectuarse por medio de dos estructuras particulares.

2.3.1 Las estructuras básicas

Las estructuras `shmid_ds` y `shm_inf` se definen en el archivo de cabecera `<linux/shm.h>`. Sin embargo, basta con incluir el archivo `<sys/shm.h>` para poder acceder a ellos. La estructura `shmid_ds` corresponde a una entrada en la tabla de memorias compartidas. Veámosla en detalle:

tipo	campo	descripción
<code>struct ipc_perm</code>	<code>shm_perm</code>	Derechos de acceso
<code>int</code>	<code>shm_segsz</code>	Tamaño del segmento (bytes)
<code>time_t</code>	<code>shm_atime</code>	Fecha de última vinculación
<code>time_t</code>	<code>shm_dtime</code>	Fecha de última desvinculación
<code>time_t</code>	<code>shm_ctime</code>	Fecha de la última modificación
<code>unsigned short</code>	<code>shm_cpid</code>	Número del proceso creador
<code>unsigned short</code>	<code>shm_lpid</code>	Número del proceso que ha efectuado la última operación
<code>short</code>	<code>shm_nattch</code>	Número de vinculaciones
<code>unsigned short</code>	<code>shm_npages</code>	Tamaño de los segmentos (número de páginas de memoria)
<code>unsigned long *</code>	<code>shm_pages</code>	Tabla de punteros a las ventanas de memoria
<code>struct</code>	<code>attaches</code>	Descriptores para las vinculaciones
<code>vm_area_struct *</code>		

Los tres últimos campos son privados: son utilizados por el núcleo para organizar las memorias compartidas.

El mismo archivo de cabecera define la estructura `shminfo` (que no hay que confundir con la estructura `shm_info` utilizada únicamente por el núcleo). Se utiliza en una

llamada a *shmctl* con *IPC_INFO* como argumento. Esta estructura es utilizada por ejemplo por el programa *ipcs*. Se usa raramente, excepto en programas de sistema de estadísticas o de observación de la máquina.

He aquí su detalle:

tipo	campo	descripción
int	shmmax	Tamaño máximo del segmento (bytes)
int	shmin	Tamaño mínimo del segmento (bytes)
int	shmni	Número máximo de segmentos
int	shmseg	Número máximo de segmentos por proceso
int	small	Número máximo de segmentos en número de páginas de memoria

2.3.2 Creación y búsqueda de una zona de memoria compartida

La llamada al sistema *shmget* crea una nueva zona de memoria compartida o bien permite obtener un acceso a una memoria compartida existente. El prototipo de esta llamada es el siguiente:

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key_t key, int size, int option);
```

El parámetro *size* corresponde al tamaño del segmento de memoria compartida que se desea. El tercer parámetro corresponde a los parámetros estándar de creación de un IPC.

Resulta útil precisar que si esta llamada al sistema se utiliza para obtener una referencia a una zona de memoria compartida ya existente, el tamaño especificado debe ser inferior o igual al de la memoria existente.

En caso contrario, el tamaño asignado debe ser múltiplo de *PAGE_SIZE*, que corresponde al tamaño de una página de memoria (4 KB en la arquitectura x86).

Esta llamada al sistema devuelve en caso de éxito el identificador de la memoria compartida. En caso de error, la variable *errno* puede tomar uno de los valores siguientes:

error	significado
ENOMEM	Memoria insuficiente
EINVAL	Parámetros no válidos (<i>shmin</i> > <i>size</i> o tamaño > <i>shmmax</i>)
ENOENT	No existe ningún segmento de memoria compartida para la clave dada
EXIST	Se ha dado el valor <i>IPC_CREAT - IPC_EXCL</i> y el segmento de memoria compartida ya existe
EDRM	El segmento de memoria se ha marcado como destruido
EACCESS, EPERM	Permisos de acceso insuficientes

EFAULT
EINVAL

Parámetros incorrectos
Todos los identificadores de memorias compartidas están en uso, o bien la asignación de una memoria compartida del tamaño especificado sería mayor que **SHMALL**

2.3.3 Vinculación de una zona de memoria

La llamada al sistema *shmat* permite vincular una memoria compartida a un proceso. Esta operación consiste en realidad en vincular una zona de memoria al espacio de direccionamiento virtual del proceso que llama.

El prototipo de esta llamada al sistema es el siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

void *shmat (int shmid, const void *shmaddr, int option);
```

El segundo parámetro *shmaddr* permite especificar la dirección de la memoria compartida:

- si *shmaddr* vale NULL, el sistema operativo intenta encontrar una zona de memoria libre (es el método más seguro);
- si no, el sistema operativo intenta vincular la memoria compartida a la dirección especificada. Si se especifica la opción **SHM_RND**, el sistema intenta vincular la zona de memoria a una dirección múltiplo de **SHMLBA** lo más próxima posible a la especificada.

Hay que precisar que esta llamada al sistema permite el uso de una opción especial: **SHM_RDONLY**. Esta opción permite especificar que el proceso sólo puede acceder al segmento de memoria en lectura. Si no es así, el segmento se vincula en lectura y escritura. No existe un medio para realizar una vinculación en escritura exclusiva.

Esta llamada al sistema actualiza los parámetros siguientes de la estructura *shmid_ds*:

- *shm_atime* recibe la fecha actual;
- *shm_lpid* recibe el pid del proceso que llama;
- *shm_nattch* se incrementa en una unidad.

shmat devuelve la dirección del segmento de memoria que se puede manipular. Si *shmat* devuelve NULL, *errno* recibe uno de los valores siguientes:

<i>error</i>	<i>significado</i>
EACCESS	Permisos de acceso insuficientes
EINVAL	Clave o dirección no válida
ENOMEM	Memoria insuficiente
ETOOM	Segmento marcado como destruido

2.3.4 Desvincular una zona de memoria

La llamada al sistema *shmdt* le permite a un proceso desvincular una zona de memoria compartida de su espacio de direccionamiento.

El prototipo de esta llamada es el siguiente:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmdt (const void *shmaddr);
```

Esta llamada al sistema actualiza los campos siguientes de la estructura *shmid_ds*:

- *shm_dtime* recibe la fecha actual;
- *shm_lpid* recibe el pid del proceso que llama;
- *shm_nattch* se decrementa en una unidad.

2.3.5 Control de las zonas de memoria compartidas

La llamada *shmctl* permite controlar la gestión de un segmento de memoria compartida. El prototipo de esta llamada es el siguiente:

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

Veamos los diferentes mandatos posibles, que corresponden a usos distintos:

- **IPC_STAT**: permite obtener las informaciones respecto al segmento de memoria compartida. Estas informaciones se copian en la zona de memoria apuntada por *buf*. Hay que destacar que el proceso debe tener el derecho de leer el segmento de memoria para que esta llamada tenga éxito.

- **IPC_SET**: permite aplicar cambios que el usuario ha efectuado en los campos `uid`, `gid` o `mode` del campo `shm_perms`. Sólo se utilizan los 9 bits de menor peso. El campo `shm_ctime` también se modifica. Sólo el propietario, el creador y el superusuario tienen derecho a efectuar esta operación.
- **IPC_RMID**: permite marcar un segmento de memoria compartida como destruido. Este segmento se destruirá efectivamente cuando se suprima la última vinculación.

Hay que añadir a estos mandatos dos opciones específicas de Linux. Estas dos opciones sólo pueden ser usadas por el superusuario, y permiten o, por el contrario, impiden el *swap* de un segmento de memoria compartida:

- **SHM_LOCK**: impide el *swap* de un segmento de memoria compartida;
- **SHM_UNLOCK**: permite el *swap* de un segmento de memoria compartida.

2.3.6 Interacción con otras llamadas al sistema

Las zonas de memoria compartidas de un proceso se tienen en cuenta en otras llamadas al sistema:

- *fork*: el proceso hijo hereda segmentos de memoria compartidos que se vinculan a su padre;
- *exec*, *exit*: todos los segmentos se desvinculan (pero no se destruyen).

2.3.7 Ejemplo de uso

Este apartado proporciona un ejemplo para ilustrar el uso de los segmentos de memoria compartidos junto con semáforos. Un segmento de memoria compartido entre varios procesos es un recurso crítico, por lo que es necesario proteger su acceso. Los semáforos son un medio adaptado a la gestión de estos accesos.

Consideramos una zona de memoria compartida. Esta zona puede accederse en lectura o en escritura. Las restricciones impuestas para que las informaciones leídas o escritas por el proceso sean coherentes son:

- varios procesos pueden leer simultáneamente en el segmento;
- cuando un proceso escribe, debe ser el único con acceso al segmento.

Una solución a este problema consiste en utilizar dos semáforos, uno para el acceso en lectura y otro para el acceso en escritura. El semáforo de acceso en escritura es un

semáforo binario, porque el acceso en escritura es exclusivo. El semáforo de acceso en lectura es un semáforo *N*ario, donde *N* corresponde al número autorizado de accesos simultáneos en lectura.

El uso de los dos semáforos no está disociado porque el acceso al segmento en escritura está vinculado a los accesos en lectura. Este problema se resuelve por el uso de un grupo de semáforos que incluye a los dos semáforos anteriores.

El archivo *Comun.h* contiene la definición del grupo, así como las constantes necesarias para la creación de claves de acceso a los diferentes IPC.

```
#ifndef COMUN_H
#define COMUN_H

#define NOM "Comun.h"
#define PROJ_FTOK_SEM 'a' /* parámetro ftok */
#define PROJ_FTOK_MEM '0' /* parámetro ftok */
#define TAM 20 /* Tamaño de la zona compartida */
#define N 5 /* Número de lectores simultáneos */

/* Valores de inicialización del grupo:
 * 1 Escritor
 * N lectores simultáneos
 */

ushort init_sem[]={1,N};

/* petición y liberación de acceso de la zona en escritura */
struct sembuf pide_escritura[]={{0,-1,SEM_UNDO},{1,-N,SEM_UNDO}};
struct sembuf para_escritura[]={{0,+1,SEM_UNDO},{1,N,SEM_UNDO}};

/* petición y liberación de acceso de la zona en lectura */

struct sembuf pide_lectura= {1,-1,SEM_UNDO};
struct sembuf para_lectura = {1,+1,SEM_UNDO};

#endif
```

Otros dos archivos componen el ejemplo:

- *Escritor.c*, que crea una zona compartida y ejecuta el escritor;
- *Lector.c*, que vincula la zona compartida y crea el número de lectores pasado como parámetro.

```
/* Escritor.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#include "Comun.h"

int shmid, semid;

static void uso ()
{
    printf (
"Ejemplo de aplicación de semáforos y memorias compartidas\n");
    printf (" --> Lectores/Escritor\n");
    printf (" Escritor  \n\n");
    exit (-1);
}

static void unError (char *msg)
{
    fprintf (stderr, "Error Escritor: %s :%d \n", msg, errno);
    exit (-1);
}

void escritura (char *orig, char *dest, int nbytes)
{
    int i;

    semop (semid, pide_escritura, 2);
    printf ("Inicio de escritura en la zona compartida de %d"
           " bytes por el proceso %d\n", nb_octets, getpid ());
    for (i = 0; i < nbytes; i++)
        dest[i] = orig[i];

    sleep (1);
    printf ("Fin de escritura en la zona compartida "
           "por el proceso %d\n", getpid ());
    semop (semid, stoppe_ecriture, 2);
}

void init_escritor ()
```

```
{
    int clave_mem, clave_sem;

    /* Generación de la clave de acceso a la memoria compartida */
    if ((clave_mem = ftok (NOM, PROJ_FTOK_MEM)) == -1)
        unError ("Error en ftok memoria compartida");

    /* Generación de la clave de acceso al semáforo */
    if ((clave_sem = ftok (NOM, PROJ_FTOK_SEM)) == -1)
        unError ("Error en ftok semáforo");

    if ((shmid = shmget (clave_mem, TAM, IPC_CREAT | IPC_EXCL |
0644)) == -1)
        unError ("Error en shmget");

    if ((semid = semget (clef_sem, 2, IPC_CREAT | IPC_EXCL |
0666)) == -1)
        unError ("Error en semget");

    if (semctl (semid, 2, SETALL, init_sem) == -1)
        unError ("Error en semctl");
}

void escritor ()
{
    char          buf[20];
    char          *addr;
    int           i, j;

    if ((addr = shmat (shmid, NULL, 0)) == (char *) -1)
        unError ("Error en shmat");

    for (j = 0; j < 4; j++) {
        for (i = 0; i < 20; i++)
            buf[i]++;
        escritura (buf, addr, 20);
    }
}

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

    if ((argc != 1)) {
        uso ();
        exit (1);
    }
}
```

```
    }
    init_escritor ();
    escritor ();
    sleep (5);
    if (semctl (semid, 0, IPC_RMID, 0) == -1)
        unError ("Error en semctl");
    if (shmctl (shmid, IPC_RMID, NULL) == -1)
        unError ("Error en shmctl");
    exit (0);
}

/* Lector.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>

#include "Commun.h"

int shmid, semid;

static void uso ()
{
    printf ("Ejemplo de aplicación de semáforos y "
            "memorias compartidas\n");
    printf (" --> Lectores/Escritor\n");
    printf (" Lector <num_lectores>\n\n");
    exit (-1);
}

static void unError (char *msg)
{
    fprintf (stderr, "Error Lector: %s :%d \n", msg, errno);
    exit (-1);
}

void lectura (char *orig, char *destino, int nbytes)
{
    int i;

    semop (semid, &pide_lectura, 1);
    printf (
```



```
"Inicio de lectura en la zona compartida de %d bytes"
" por el proceso %d\n" ,nbytes, getpid ());

for (i = 0; i < nbytes; i++)
    destino[i] = orig[i];
sleep (2);
printf ("Fin de lectura en la zona compartida "
        "por el proceso %d\n", getpid ());

semop (semid, &stoppe_lecture, 1);
sleep (1);
}

void init_lectores ()
{
    key_t clave_mem, clave_sem;
    /*Generación de la clave de acceso a la memoria compartida*/
    if ((clave_mem = ftok (NOM, PROJ_FTOK_MEM)) == -1)
        unError ("Error en ftok memoria compartida");

    /* Generación de la clave de acceso al semáforo*/
    if ((clave_sem = ftok (NOM, PROJ_FTOK_SEM)) == -1)
        unError ("Error en ftok semáforo");

    if ((shmidx = shmget (clave_mem, TAM, 0)) == -1)
        unError ("Error en shmget");

    if ((semid = semget (clave_sem, 2, 0)) == -1)
        unError ("Error en semget");
}

void lector ()
{
    char          buf[20];
    char          *addr;
    int           i;

    if ((addr = shmat (shmidx, NULL, SHM_RDONLY)) == (char *) -1)
        unError ("Error en shmat");

    sleep (1);
    for (i = 0; i < 4; i++)
        lectura (addr, buf, 20);
    exit (0);
}
```

```

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

    if ((argc < 2) || (argc > 2)) {
        uso ();
        exit (1);
    }
    init_lectores ();
    nb_process = atoi (argv[1]);

    for (i = 1; i <= nb_process; i++) {
        switch (fork ()) {
            case 0:
                lector ();
                break;
            case -1:
                unError ("Error en fork");
            default:
                break;
        }
    }
    exit (0);
}

```

En estos dos programas, el uso de la llamada al sistema *sleep* sirve únicamente para desincronizar los procesos. No hay ninguna restricción sobre el orden de ejecución de las lecturas y escrituras.

La ejecución del ejemplo para un escritor y tres lectores permite constatar que el programa funciona correctamente. Las lecturas de la zona compartida pueden ser entrelazadas, mientras que las escrituras son siempre exclusivas.

```

scylla (2)> ./Escritor & ; ./Lector 3
[1] 1172
Inicio de escritura en la memoria compartida de 20 bytes por el proceso
1172
scylla (2)> Fin de escritura en la memoria compartida por el proceso 1172
Inicio de escritura en la memoria compartida de 20 bytes por el proceso
1172
Fin de escritura en la memoria compartida por el proceso 1172
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1176
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1175
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1174
Fin de lectura en la memoria compartida por el proceso 1174
Fin de lectura en la memoria compartida por el proceso 1175
Fin de lectura en la memoria compartida por el proceso 1176

```

```
Inicio de escritura en la memoria compartida de 20 bytes por el proceso
1172
Fin de escritura en la memoria compartida por el proceso 1172
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1174
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1176
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1175
Fin de lectura en la memoria compartida por el proceso 1174
Fin de lectura en la memoria compartida por el proceso 1176
Fin de lectura en la memoria compartida por el proceso 1175
Inicio de escritura en la memoria compartida de 20 bytes por el proceso
1172
Fin de escritura en la memoria compartida por el proceso 1172
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1175
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1174
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1176
Fin de lectura en la memoria compartida por el proceso 1176
Fin de lectura en la memoria compartida por el proceso 1174
Fin de lectura en la memoria compartida por el proceso 1175
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1175
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1174
Inicio de lectura en la memoria compartida de 20 bytes por el proceso 1176
Fin de lectura en la memoria compartida por el proceso 1176
Fin de lectura en la memoria compartida por el proceso 1174
Fin de lectura en la memoria compartida por el proceso 1175
```

3 Conceptos avanzados

3.1 Opción de compilación del núcleo

Aunque los IPC lleven implementados cierto tiempo, hay que especificar su inclusión en la compilación del núcleo. Normalmente, los IPC se incluyen en los núcleos de las distribuciones estándar. Sería absurdo no insertarlos en el núcleo, porque ciertas aplicaciones necesitan estas herramientas de comunicación para funcionar (*perl* por ejemplo).

En la configuración del núcleo (`make config`), conviene responder y a la pregunta siguiente:

```
System V IPC (CONFIG_SYSVIPC) [Y/n]
```

Tras recompilar el núcleo y rearrancar el sistema, los IPC estarán incluidos en el núcleo.

3.2 Los programas *ipcs* e *ipcrm*

3.2.1 *ipcs*

Este mandato permite visualizar las tres tablas de IPC gestionadas por el núcleo.

```
Gandalf(gandalf)--> ipcs
```

----- Shared Memory Segments -----					
shmid	owner	perms	bytes	nattch	status
----- Semaphore Arrays -----					
semid	owner	perms	nsems	status	
----- Message Queues -----					
msqid	owner	perms	used-bytes	messages	
256	dumas	666	13078	54	
257	dumas	600	1378	25	

En el ejemplo anterior, se han creado dos colas de mensajes en el sistema.

El mandato *ipcs* ofrece las opciones estándar de manipulación de los IPC:

- -s: visualización de los semáforos
- -m: visualización de las memorias compartidas
- -q: visualización de las colas de mensajes
- -a: las tres (opción predeterminada)

Además de estas cuatro opciones, este mandato gestiona otras cinco que permiten una visualización diferente:

- -t → *time*: permite obtener ciertas informaciones como la fecha de la última modificación, etc.:

```
gandalf> ipcs - q - t
----- Message Queues Send/Recv/Change Times -----
msqid    owner send  recv  change
0  dumas Jan 3 10:49:30  Not set  Jan 3 10:54:35
```

- -p → *pid*: indica los IPC vinculados a un proceso
- -c → *creator*: da las informaciones sobre el creador del IPC:

```
gandalf> ipcs -q -c
----- Message Queues: Creators/Owners -----
```

```
msqid      perms  cuid  cgid  uid    gid
128 666    dumas etu  dumas etu
```

- `-l` → *limits*: visualiza los recursos límite respecto a la gestión de los IPC:

```
gandalf>ipcs -l
----- Shared Memory Limits -----
max number of segments = 128
max seg size (kbytes) = 16384
max total shared memory (kbytes) = 16777216
min seg size (bytes) = 1

----- Semaphore Limits -----
max number of arrays = 128
max semaphores per array = 32
max semaphores system wide = 4096
max ops per semop call = 32
semaphore max value = 32767

----- Messages: Limits -----
max queues system wide = 128
max size of message (bytes) = 4056
default max size of queue (bytes) = 16384
```

- `-u` → *summary*: resume la situación de los IPC sobre la máquina:

```
Gandalf(beetlejuice)--> ipcs -u
----- Shared Memory Status -----
segments allocated 4
pages allocated 2
pages resident 1
pages swapped 1
Swap performance: 5 attempts 5 successes

----- Semaphore Status -----
used arrays = 3
allocated semaphores = 5

----- Messages: Status -----
allocated queues = 1
used headers = 62
used space = 16368 bytes
```

3.2.2 *ipcrm*

El mandato *ipcrm* permite destruir un IPC. Es necesario ser el creador del IPC o super-usuario para poder efectuar esta operación.

Su sintaxis es simple:

```
ipcrm [ shm | msg | sem ] id
```

Por ejemplo, para destruir la cola de mensajes con el identificador 657 (véase la sección 3.2.1), basta con escribir:

```
gandalf> ipcrm msg 657  
resource deleted
```

La cola de mensajes se suprime automáticamente si quien llama posee los derechos necesarios.

4 Presentación general de la implementación

Los archivos fuente de los IPC System V pueden encontrarse en el directorio *ipc* del árbol del núcleo. Se componen de cuatro módulos:

- *msg.c*: gestión de las colas de mensajes;
- *sem.c*: gestión de los semáforos;
- *shm.c*: gestión de las memorias compartidas;
- *util.c*: funciones de inicialización y gestión de los derechos de acceso.

4.1 Funciones comunes

Una parte de las fuentes se comparte entre los distintos módulos de los IPC. El código común se agrupa en el archivo *util.c*. Este módulo cumple varias funciones:

1. inicializar los tres tipos de IPC (función llamada una sola vez al arrancar el núcleo);
2. verificar los derechos de acceso;
3. en el caso en que los IPC no se hayan integrado en la compilación del núcleo (*cosa que no es aconsejable porque numerosos programas los necesitan*), este módulo permite efectuar únicamente las declaraciones de las funciones.

La inicialización (función *ipc_init*) consiste en lanzar la función de inicialización específica de cada uno de los IPC. Se llama al inicializarse el núcleo por la función *main* situada en el archivo *init/main.c*.

La función de gestión de los derechos de acceso `ipcperms` se encarga, entre otras cosas, de verificar que se cuenta con los derechos necesarios para acceder al IPC deseado.

4.2 Algoritmos

Los IPC se implementan en forma de tres tablas (una por tipo de IPC) de tamaño `MSGMNI`, `SEMMNI` y `SHMMNI`. Estas tres tablas de tamaño constante son tablas de punteros a las estructuras de manipulación de los IPC (`msqid_ds`, `semid_ds` y `shmid_ds`).

La inicialización de las estructuras de los IPC, al arrancar la máquina, consiste en llamar a la función `ipc_init`. El código de esta función tiene un interés limitado porque contiene únicamente la llamada a cada una de las tres funciones de inicialización `shm_init`, `msg_init` y `sem_init`, situadas en sus archivos respectivos.

Esta inicialización asigna el valor `IPC_UNUSED` a cada casilla de la tabla. Esta fase marca pues todos los recursos IPC como libres. Cada elemento de la tabla puede tener tres tipos de valores:

- `IPC_UNUSED`: recurso libre;
- `IPC_NOID`: o el recurso se está asignando, o bien ha sido destruido;
- la dirección de la estructura que contiene los datos relativos al recurso gestionado.

La figura 11.2 representa la tabla de memorias compartidas. Hay que precisar que la representación es exactamente la misma para los otros dos tipos de IPC, excepto en el tipo.

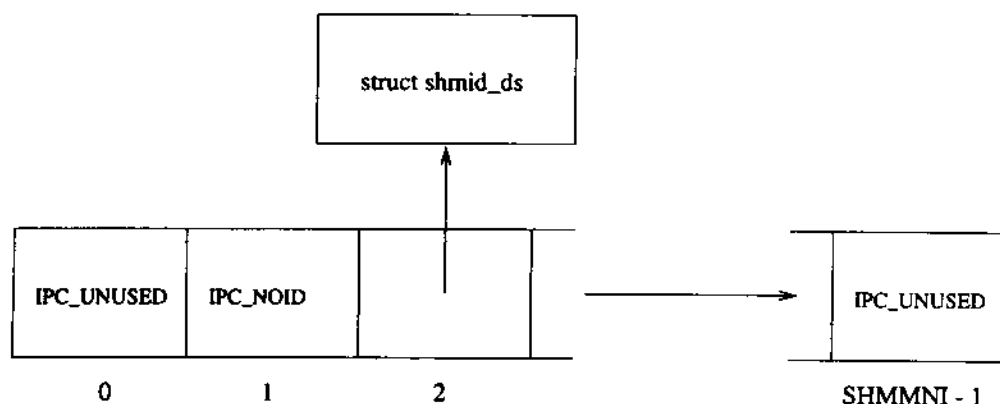


FIG. 11.2 – Tabla de memorias compartidas

4.3 Gestión de las claves

Las diferentes llamadas al sistema relacionadas con los IPC utilizan una clave para identificar el recurso. El problema es asociar esta clave con el índice de la tabla correspondiente. Esta operación se efectúa mediante las funciones `findkey` que se encuentran en cada uno de los tres módulos.

Esta función recorre cada una de las entradas de la tabla deseada. Se distinguen tres casos:

- `IPC_UNUSED`: se pasa al valor siguiente.
- `IPC_NOID`: este valor se da cuando se está modificando una estructura. En este caso, el proceso queda en espera hasta que la operación finalice.
- Se trata de una estructura de datos: en este caso, la clave de esta estructura se encuentra en el campo `sem_perm.key` (sea cual sea el tipo de IPC utilizado). Basta pues con efectuar una comparación.

Si el valor de la clave no corresponde a ninguna clave existente, esta función devuelve `-1`.

5 Presentación detallada de la implementación

5.1 Colas de mensajes

El archivo `msg.c` contiene el conjunto de las funciones que implementan las colas de mensajes. Además de la implementación de la gestión de las colas de mensajes, encontramos también la implementación de las llamadas al sistema: `sys_msgsnd`, `sys_msgrcv`, `sys_msgget` y `sys_msgctl`.

Las colas de mensajes son necesarias también para el proceso *kerneld* que permite la carga a petición de módulos en el núcleo. Un cierto número de funciones está destinado a ello: `kerneld_exit`, `kd_timeout` y `kerneld_send`. Su uso no se detalla aquí.

El archivo contiene también la función de inicialización (llamada al arrancar el sistema) y la de finalización (llamada al terminar la ejecución de cada proceso), que están relacionadas con las colas de mensajes.

5.1.1 Representación interna de las colas de mensajes

Las colas de mensajes se ordenan en la tabla `msgque`. Esta tabla contiene punteros a la estructura `msqid_ds`. Sin embargo, esta tabla tiene un tamaño limitado, que se fija al valor de la constante `MSGMNI`.

La cola de mensajes de índice n está constituida en realidad por elementos como los derechos de acceso. Pero los dos elementos fundamentales son dos punteros a la cola:

- un puntero al primer elemento de la cola;
- un puntero al último elemento de la cola.

Toda la gestión de las colas de mensajes puede resumirse en una simple gestión de listas encadenadas.

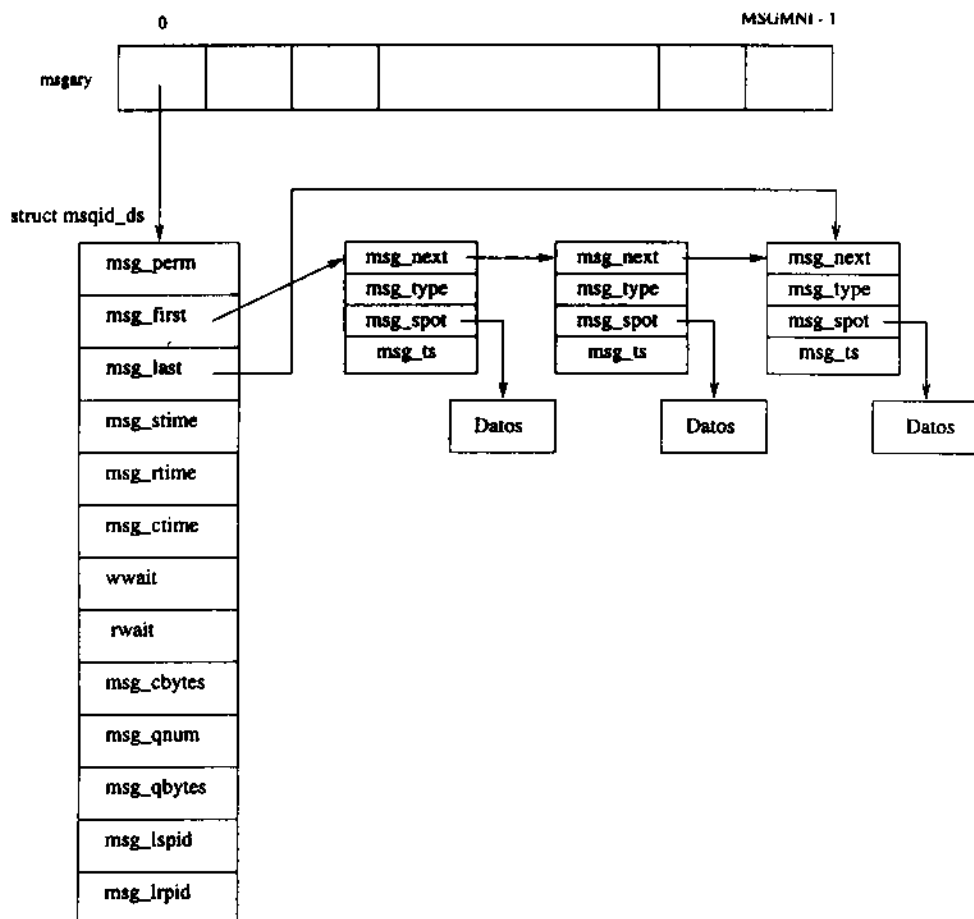


FIG. 11.3 – Representación interna de las colas de mensajes

5.1.2 La inicialización

La función `ipc_init` llama a la función de `msg_init` al arrancar el sistema en la inicialización de los IPC. Ésta guarda el valor `IPC_UNUSED` en cada casilla de la tabla `semary` y pone un cierto número de variables internas a 0, como por ejemplo el espacio de memoria ocupado por los datos almacenados en las colas de mensajes.

5.1.3 Creación de una cola de mensajes

La función `sys_msgget` permite la creación de una nueva cola de mensajes. En este caso, efectúa una llamada a la función `newque` que crea una nueva cola de mensajes si no existe, o bien modifica el estado de la cola de espera ya existente. La operación de creación termina por la asignación de los diferentes derechos de acceso.

5.1.4 El envío de un mensaje

La llamada `msgsnd` corresponde en realidad a la función `real_msgsnd`, que realiza en realidad una simple inserción de elementos en la lista de datos. Evidentemente, esta operación sólo se efectúa si se respetan las condiciones en materia de espacio de memoria y de autorizaciones de acceso.

Al final de la llamada, se actualiza el campo `msg_cbytes` que representa el espacio de memoria ocupado por los datos de la cola, así como `msg_qnum` que representa el número de mensajes presentes en la cola.

5.1.5 La recepción de un mensaje

La recepción, resultado de la llamada a `msgrcv`, se trata por la función `sys_msgrcv`. En realidad, esta función llama a la función `real_msgrcv` que es la que realmente gestiona la recepción de un mensaje en la cola. Tras las verificaciones de uso (derechos de acceso, existencia de la cola deseada), se recorre la cola en busca del tipo deseado. Una vez localizado, el elemento afectado se retira de la cola y se devuelve.

5.1.6 El control de una cola de mensajes

El control de una cola de mensajes se efectúa mediante la llamada al sistema `msgctl`. La implementación de esta llamada es bastante simple: la gran mayoría de los mandatos que se pueden pasar a esta llamada son de orden informativo.

El mandato más particular `IPC_SET` permite fijar ciertos parámetros situados en la estructura `msqid_ds`. Esta operación se efectúa con una simple copia en memoria mediante la función `memcpy_fromfs`. Este mandato efectúa también la actualización de los derechos y tiempos de acceso a la cola de mensajes.

El único mandato realmente interesante es la destrucción de una cola de mensajes. Esta operación se efectúa mediante el mandato `IPC_RMID`. En este caso, la función `free` realiza la liberación del recurso. La liberación se efectúa en tres tiempos: primero se busca la cola de mensajes especificada y se verifica su existencia, luego se destruye cada una de las zonas de memoria gestionadas por la cola y, finalmente, la cola de mensajes. Esta operación se parece a una operación de destrucción en una lista encadenada.

5.2 Semáforos

El archivo fuente `sem.c` contiene el conjunto de funciones que implementan los semáforos. Se encuentran las funciones `sys_semget`, `sys_semctl` y `sys_semop`. El archivo contiene también la función de inicialización (llamada al arrancar el sistema) y la función de finalización (llamada al final de la ejecución de cada proceso).

5.2.1 Representación interna de los semáforos

Los semáforos se almacenan por el sistema en la tabla `semapry`. Esta tabla contiene punteros a la estructura `semid_ds`. Su tamaño indica pues el número máximo de grupos de semáforos que se pueden crear: `SEMNI`. Cada estructura `semid_ds` contiene tres listas:

- la lista de semáforos del grupo (`base`), almacenada en memoria en forma de una tabla justo detrás del descriptor de semáforos;
- la lista de operaciones en espera (`sem_pending`) gestionada en forma de una lista doblemente encadenada;
- la lista de peticiones anulables (`undo`).

5.2.2 La inicialización

La función de inicialización `sem_init` es llamada por la función `ipc_init` al arrancar el sistema en la inicialización de los IPC. Coloca el valor `IPC_UNUSED` en cada casilla de la tabla `semapry` e inicializa los contadores de los semáforos a 0.

5.2.3 Creación de semáforos

La función `sys_semget` gestiona la creación de nuevos semáforos. Llama a la función `newary`, que busca una entrada libre en la tabla `semary` y asigna dinámicamente memoria para el nuevo semáforo. Esta función también inicializa las informaciones relacionadas con el tiempo y la pertenencia al semáforo.

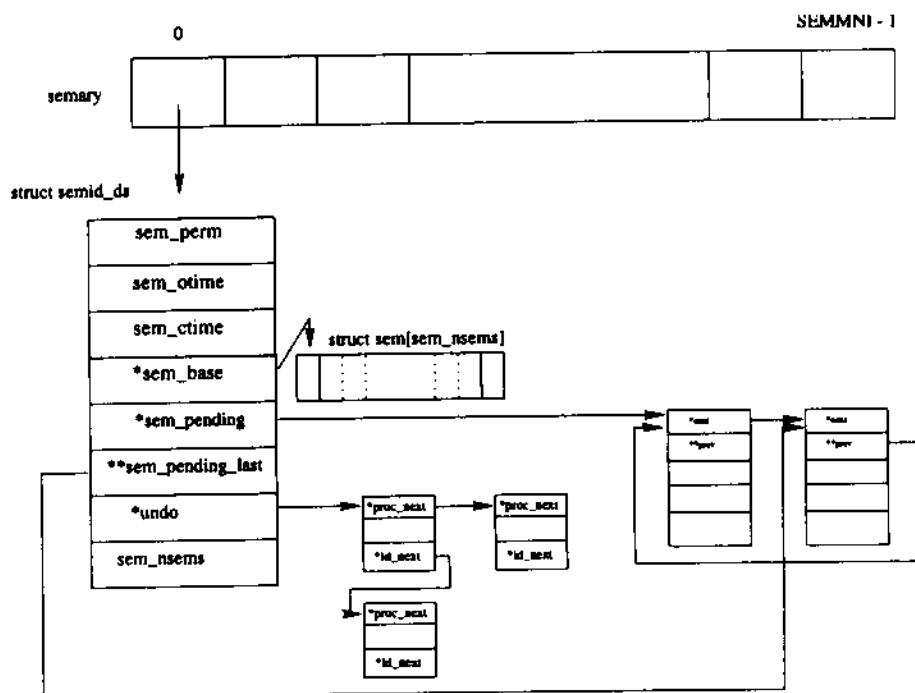


FIG. 11.4 – Representación interna de los semáforos

5.2.4 Control de los semáforos

La función `sys_semctl` realiza el control de los semáforos. En la mayor parte de casos (operaciones `GET*`, `IPC_STAT` o `IPC_SET`), es muy simple porque lee o modifica informaciones relacionadas con los semáforos. Estas operaciones consisten en una transferencia de datos de/hacia el espacio de usuario hacia/de el espacio del núcleo. La operación `IPC_RMID` destruye el semáforo y los recursos asociados. Las dos operaciones algo más complejas son `SETVAL` y `SETALL`. Estas operaciones modifican el valor del contador de un semáforo, por lo que puede ser necesario despertar procesos en espera de acceso al semáforo, en el caso en que se incremente el contador o bien cuando pasa a cero. Éste es el papel de la función `update_queue`. Esta función se detalla en la sección siguiente.

5.2.5 Modificación de los valores de semáforos

Las modificaciones de los valores de los semáforos se realizan llamando a *semop*. Es la llamada de realización más compleja, porque hay que gestionar varios problemas.

Una operación sobre un semáforo puede bloquear un proceso, por lo que es necesaria una cola de espera para almacenar los procesos en espera de acceso al semáforo. Esta cola se implementa en forma de una lista doblemente encadenada. Se accede por los campos *sem_pending* y *sem_pending_last* de la estructura *semid_ds*. Esta lista es también accesible para cada proceso por la tabla de procesos (campo *semsleeping* de la estructura *task_struct*). Las funciones *insert_into_queue* y *remove_from_queue* permiten la manipulación de la lista, respectivamente insertan y suprimen un elemento de la lista.

El conjunto de operaciones a realizar debe ser atómico. La verificación de realización de las operaciones se hace por la función *try_semop*. Esta función devuelve 0 si las operaciones son posibles, 1 si el proceso debe dormirse, o el error *EAGAIN* si la petición se ha hecho con el indicador *IPC_NOWAIT*.

El sistema debe gestionar la anulación de las solicitudes si el indicador *SEM_UNDO* está activado. Para ello, tras llamar a la función *try_semop*, el sistema construye una lista encadenada de las operaciones anulables por el proceso. Esta lista es accesible por el campo *semund* de la estructura del semáforo y, para cada proceso, es accesible por la tabla de procesos de manera que libere los recursos que ya no usen.

A continuación, el sistema realiza las operaciones solicitadas (llamada a la función *do_semop* si la función *try_semop* ha tenido éxito) o bien pone el proceso en espera en la lista encadenada y lo duerme (función *sleep_interruptible*).

Tras llamar a la función *do_semop*, puede ser necesario despertar procesos dormidos. Esto se realiza por la función *update_queue*. Esta función recorre la cola de espera asociada al semáforo pasado como parámetro e intenta realizar las operaciones solicitadas por un proceso dormido; si es posible, el proceso dormido vuelve a la cola de espera de procesos a punto para su ejecución (función *wake_up_interruptible*). Esta función repite el proceso mientras pueda despertar al menos un proceso en la lista.

5.2.6 La finalización

La finalización no afecta al fin de existencia de los semáforos sino a la finalización de los procesos. Cuando un proceso termina, el sistema llama a la función *sem_exit*

para anular las operaciones que el proceso haya efectuado sobre los semáforos (con el indicador `SEM_UNDO`). La función verifica si el proceso no está dormido y en espera de acceso a un semáforo, mediante el campo `semsleeping` de la estructura `task_struct`, y lo retira de la cola si es así. Al liberar recursos la anulación de operaciones, la función llama a `update_queue` para despertar los procesos en espera de acceso a dichos recursos.

5.3 Memorias compartidas

El archivo `shm.c` contiene todas las funciones que implementan las memorias compartidas. La gestión de las memorias compartidas se efectúa a través de un cierto número de llamadas al sistema: `sys_shmget`, `sys_shmctl`, `sys_shmat` y `sys_shmdt`.

Además de las funciones de inicialización de la estructura de memorias compartidas, se definen también las funciones de asignación y liberación de un segmento de memoria compartida: `newseg` y `killseg`.

Otras funciones más específicas de las memorias compartidas se encuentran también en este módulo: las funciones que permiten hacer *swap* sobre una zona de memoria compartida: `shm_swap_in` y `shm_swap`.

La gestión de las memorias compartidas utiliza una lista circular de las diversas vinculaciones realizadas. La gestión de esta lista se realiza mediante dos operaciones: `insert_attach` que permite insertar un elemento, y `remove_attach` que permite suprimir uno.

5.3.1 Representación interna de las memorias compartidas

Las memorias compartidas se implementan en forma de una tabla de punteros (`shm_segs`) sobre estructuras de tipo `shmid_ds`. Esta tabla contiene como máximo `SHMMNI` elementos.

Los elementos fundamentales de la estructura de la memoria compartida son:

- el tamaño del segmento en bytes, y en número de páginas;
- una tabla de punteros a las diferentes *ventanas*;
- una lista circular de las vinculaciones de la memoria compartida.

La representación interna se esquematiza en la figura 11.5.

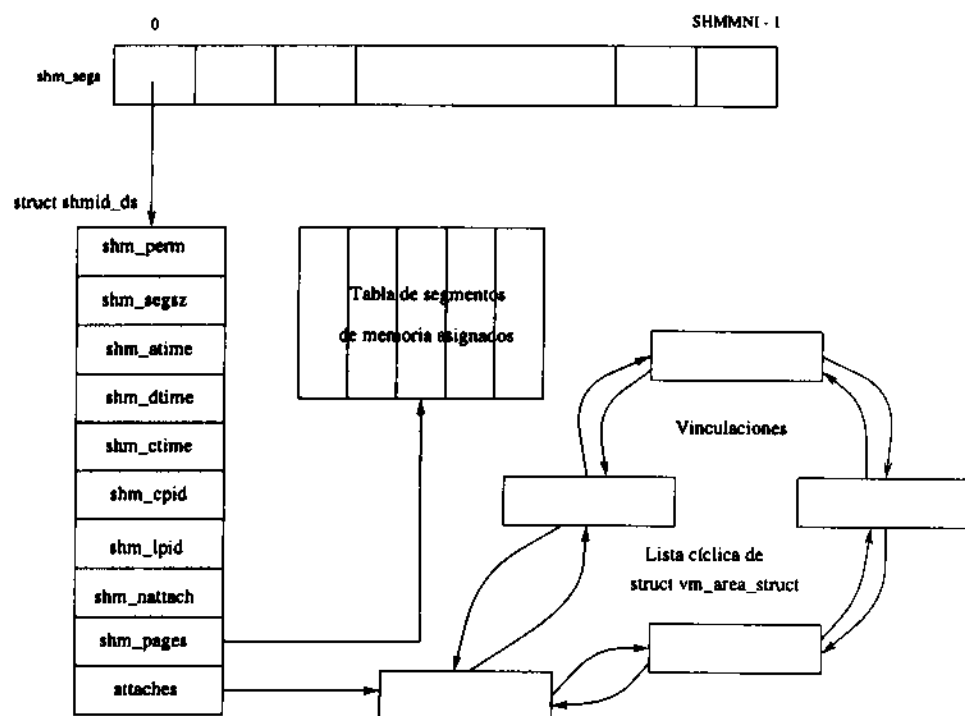


FIG. 11.5 – Representación interna de las memorias compartidas

5.3.2 La inicialización

La función de inicialización `shm_init` se llama al arrancar el sistema, en la inicialización de los IPC. Coloca el valor `IPC_UNUSED` en cada casilla de la tabla `shm_ary` y pone un cierto número de variables internas a 0, como por ejemplo el número de segmentos utilizados.

5.3.3 Creación de una zona de memoria compartida

La creación de una nueva zona de memoria compartida se implementa en el código de la llamada `shmget`. Si la memoria compartida asociada a la clave no existe, una llamada a la función `newseg` creará una nueva memoria compartida. Esta función asigna una nueva estructura `shm_id_ds`, así como los punteros a los segmentos de memoria. La llamada devuelve la referencia a esta memoria compartida.

En caso de éxito de la llamada a la función `findkey`, que consiste en buscar si una memoria compartida ha sido ya asignada con la clave pasada como parámetro, entonces se devuelve la identificación de la memoria compartida.

5.3.4 Vinculación de una zona de memoria

La llamada *shmat* vincula un segmento de memoria compartida a un proceso. Para ello se utiliza el sistema de memoria virtual. Se crea la variable *shmd*, de tipo *vm_area_struct*. Esta estructura permite indicar el tipo de memoria empleada (véase el capítulo sobre la gestión de la memoria). Mediante la llamada a la función *shm_map*, la zona de memoria se proyecta en el espacio del proceso.

La variable se inserta en la lista de vinculaciones de la memoria compartida, mediante la llamada a la función *insert_attach*. Esta lista permite conocer el grupo de procesos que pueden acceder a una zona de memoria compartida dada.

5.3.5 Desvinculación de una zona de memoria compartida

La desvinculación de una zona de memoria compartida se realiza llamando a *shmdt*. La función correspondiente del núcleo *sys_shmdt* llama a la función *do_munmap* definida en el archivo fuente *mm/mmap.c* (véase el capítulo 8, sección 6.6.1).

Esta operación, aunque pasando por la gestión de la memoria, se efectúa realmente en el módulo de las memorias compartidas, gracias al sistema de la memoria virtual. La variable *shm_vm_ops* de tipo *vm_operations_struct*, declara la función de liberación de la zona de memoria compartida que se asigna: *shm_close*.

Esta función retira de la lista de vinculaciones el segmento de memoria asignada llamando a *remove_attach*, y destruye el segmento llamando a *killseg* si ya no se utiliza y la opción *SHM_DEST* está activa (lo que corresponde a una llamada a esta función tras una petición de destrucción del segmento).

La función *remove_attach* retira la zona de memoria de la lista circular.

La función *killseg* destruye cada una de las páginas llamando a la función de liberación de una página en memoria *free_page*.

5.3.6 Control de las zonas de memoria compartidas

El control de una zona de memoria compartida (*shmctl*) consiste en la mayoría de opciones en efectuar copias de informaciones. Sin embargo, la opción *IPC_RMID* provoca siempre la destrucción de la memoria compartida llamando a la función *killseg*.

Pero hay dos opciones particulares de Linux porque afectan a la posibilidad de cambiar el modo de gestión de la memoria compartida en cuanto al *swap*. Si se pasa la opción `SHM_UNLOCK`, entonces la memoria compartida se desbloquea, es decir, que puede ser de nuevo objeto de *swap*. Si, por el contrario, se pasa la opción `SHM_LOCK`, la memoria compartida no puede serlo. Estas dos operaciones se efectúan cambiando la opción en el campo `mode` de los permisos de la memoria compartida, mediante la constante `SHM_LOCKED`.

5.3.7 Herencia de las zonas de memoria compartidas

Tras una llamada a *fork*, las zonas de memoria compartidas deben ser heredadas por el proceso hijo. Esta operación se realiza al recorrer las zonas de memoria del padre al crear el proceso hijo. La función `do_fork` llama a la función `copy_mm` que llama a la función `dup_mmap`. Mediante el campo `mm->mmap->vm_ops`, la función `do_fork` encuentra la función a llamar para tener en cuenta el nuevo proceso y sus zonas de memoria compartidas. Se llama a la función `shm_open`, que añade una vinculación a esta zona ya asignada.

Los módulos cargables

Primitivas detalladas

`get_kernel_syms, create_module,
delete_module, init_module`

1 Conceptos básicos

1.1 Presentación

Fundamentalmente, el núcleo Linux está constituido por múltiples componentes que no todos son necesarios para los usuarios en todo momento. Por ello es por lo que al crearse el núcleo, Linux solicita al usuario que especifique los elementos que desea incluir en el núcleo. Evidentemente, el objetivo de esta operación es insertar únicamente los gestores necesarios en función de la configuración de la máquina y de su uso. De este modo, el tamaño del núcleo es lo más reducido posible en función de la máquina. Cuanto menor es el tamaño del núcleo, queda más memoria disponible para el usuario. Además, el arranque de una máquina con un núcleo especialmente adaptado, es decir, únicamente con los gestores de dispositivos que posee la máquina, es mucho más rápido.

Sin embargo, cualquier modificación del núcleo, como la adición o la supresión de un gestor de dispositivo, un sistema de archivos, implica la recompilación del núcleo. Esto era cierto en las primeras versiones de Linux, hasta la implementación de los módulos cargables.

El principio de los módulos cargables es generar en un primer momento un núcleo mínimo, y cargar los gestores de manera dinámica en función de las necesidades. Esto permite, en un momento dado, tener un núcleo «extendido», tal como se muestra en la

figura 12.1. Este sistema de módulos cargables existe también en otros sistemas operativos Unix como por ejemplo Solaris, pero bajo una forma diferente.

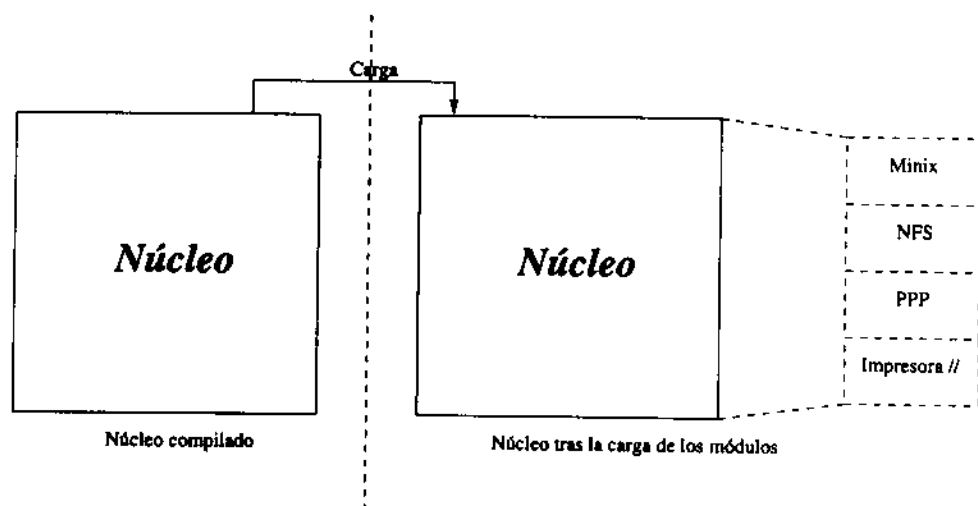


FIG. 12.1 – Carga de módulos

La figura 12.1 evidencia la capacidad de modularidad del núcleo que permiten los módulos cargables. En un uso corriente, el o los usuarios de la máquina no necesitan soporte de *NFS*, *Minix*, *PPP* o la *impresora en paralelo*. Sin embargo, durante la sesión de trabajo, un usuario puede necesitar uno o más de estos recursos. En este caso, basta con cargar esos módulos. Los módulos «se integrarán dinámicamente al núcleo». Cuando su uso haya terminado, basta con descargarlos.

1.2 Compilación

La generación de módulos se efectúa de manera particular. Linux permite a la mayoría de elementos que lo componen integrarse al núcleo o no, y de manera definitiva o bien en forma de módulo.

Sin embargo, hay que destacar que el núcleo debe estar configurado para que gestione los módulos. Para ello, es necesario activar la opción siguiente.

```
*
* Loadable module support
*
Enable loadable module support (CONFIG_MODULES) [Y/n/?]
```

Si esta opción no está activa, el uso de los módulos cargables no será posible en la máquina así configurada.

Respecto a la compilación de los propios módulos, si se desea instalar por ejemplo el sistema de archivos *MS-DOS* en forma de módulos, en la configuración, basta con especificarlo con la letra **M** de **Module** en la selección:

```
DOS FAT fs support (CONFIG_FAT_FS) [M/n/y/?] M
MSDOS fs support (CONFIG_MSDOS_FS) [M/n/?] M
VFAT (Windows-95) fs support (CONFIG_VFAT_FS) [M/n/?] M
```

Una vez lanzada la compilación del núcleo, es necesario compilar los módulos. La generación completa del núcleo debe efectuarse, pues, según estas etapas:

```
gandalf# make config
gandalf# make dep
gandalf# make clean
gandalf# make zImage
gandalf# make modules
gandalf# make modules_install
```

Una vez efectuadas estas operaciones, los módulos se encuentran en forma de archivos objeto situados en el directorio `/lib/modules`.

1.3 Operaciones en línea de mandatos: la carga manual

Linux proporciona un primer método de uso de los módulos: se trata de la versión inicial de estos módulos. El principio es permitir al superusuario cargar y descargar manualmente los módulos según las necesidades. Esta técnica es bastante pesada de manipular porque toda operación debe efectuarse manualmente. Además, el usuario normal no tiene derechos suficientes para efectuar esta operación.

Linux proporciona principalmente tres mandatos que permiten manipular los módulos: *insmod*, *lsmod* y *rmmod*.

El mandato *insmod* permite efectuar la carga del módulo. *lsmod* se limita a mostrar el contenido del archivo `/proc/modules`. *rmmod* descarga el módulo deseado.

El mandato *lsmod* indica no sólo el nombre de cada uno de los módulos cargados en memoria, sino también el número de páginas de memoria ocupadas por el módulo, así como el número de procesos que utilizan este módulo. Una página de memoria ocupa 4 KB, por lo que resulta fácil calcular la memoria economizada cuando estos módulos no están cargados.

Estos mandatos son de manipulación simple, como se demuestra en el ejemplo siguiente:

```
gandalf# insmod fat
gandalf# lsmod
Module:          #pages:Used by:
fat:              6          0
gandalf# rmmod fat
gandalf# lsmod
Module:          #pages:Used by:
```

Sin embargo, estas operaciones (salvo `lsmod`) tienen como inconveniente estar reservadas al superusuario y ser de realización pesada.

1.4 Carga dinámica

El sistema de carga dinámica es reciente en el sistema Linux. Permite automatizar las cargas de los distintos módulos en función de la demanda. Su implementación precisa la activación en la compilación de la opción `CONFIG_KERNELD` y los IPC System V. El demonio *kerneld* utiliza las colas de mensajes para comunicarse con el núcleo: la carga o descarga de un módulo la realiza *kerneld*, pero las órdenes se envían desde el núcleo mediante una cola de mensajes especial (véase el apartado sobre la implementación para mayor detalle).

La implementación de esta técnica en una máquina implica el lanzamiento del programa *kerneld* al arrancar el sistema. También es necesario efectuar ciertas operaciones para construir la lista de módulos cargables instalados. Para más información, consulte [Dumas 1996] y la documentación entregada con el paquete de módulos, donde encontrará ejemplos de archivos preparados para su uso.

Los módulos cargables dinámicamente necesitan dos programas al inicializarse la máquina:

- *depmod*, que permite generar un archivo de dependencias basado en los símbolos encontrados en el conjunto de los módulos. Estas dependencias se usarán ulteriormente en el segundo mandato;
- *modprobe*, que permite cargar un módulo o un grupo de módulos pero también cargar los módulos básicos necesarios para el correcto inicio de la máquina (como NFS, etc.).

En el ejemplo siguiente, el núcleo sólo cuenta con el soporte del sistema de archivos MS-DOS en forma de un módulo. Sin embargo, vamos a montar una partición en formato MS-DOS de manera totalmente transparente.

```

gandalf# lsmod
Module:          #pages:Used by:

gandalf# mount -t msdos /dev/fd0 /mnt

gandalf# lsmod
Module:          #pages:Used by:
msdos            2                1 {autoclean}
fat              6                [msdos] 1 {autoclean}

gandalf# ls -al /mnt
drwxr-xr-x  2 root  root          7168 Jan 1  1970 .
drwxr-xr-x 21 root  root          1024 Jun 1 03:34 ..
-rwxr-xr-x  1 root  root        19693 Jun 4 17:45 bouquin.sty
-rwxr-xr-x  1 root  root        3012 Jun 4 17:45 livre.tex

gandalf# umount /mnt

gandalf# lsmod
Module:          #pages:Used by:
msdos            2                0 {autoclean}
fat              6                [msdos] 0 {autoclean}

gandalf# sleep 60

gandalf# lsmod
Module:          #pages:Used by:

```

En la operación de montaje del sistema de archivos, los dos módulos necesarios para el uso del sistema de archivos *MS-DOS* se cargan de manera automática por el demonio *kernel.d*. Inmediatamente después de desmontar la partición, el número de referencias es nulo porque, en este caso, ningún proceso utiliza los módulos. Éstos se descargarán de manera automática unos segundos más tarde.

El lapso entre la no-utilización y la descarga se fija de modo predeterminado en 60 segundos, pero es posible fijar otra duración, utilizando la opción *delay=duración*. Una vez el demonio se lanza con esta opción, los módulos se descargan respetando este parámetro.

Pueden encontrarse numerosos detalles respecto al uso de este demonio en [Storner 1996].

2 Conceptos avanzados

2.1 La realización de un módulo cargable

2.1.1 Presentación

Los módulos cargables son remarcables, en el sentido que permiten construir un núcleo mínimo y cargar/descargar una parte del núcleo según las necesidades de los usuarios que usan la máquina.

Sin embargo, esto no es más que una faceta de este sistema, porque también se usa frecuentemente para desarrollar ciertos gestores de dispositivos. En el caso de un desarrollo tradicional, hay que modificar el núcleo, recompilarlo y rearrancar el sistema para, finalmente, probar las modificaciones, lo que hace perder mucho tiempo. Los núcleos permiten recompilar únicamente el módulo y cargarlo en el núcleo. Tras haber probado su funcionamiento, basta con descargarlo, modificarlo y así sucesivamente. Así no es necesario rearrancar el sistema. Desde luego, es preciso que el módulo no produzca errores en el núcleo, ya que si es así el reinicio se hace obligatorio.

El breve ejemplo que sigue presenta la realización de un módulo cargable capaz de dar la hora a un dispositivo indicado (*/dev/time* por ejemplo). La hora se expresará en número de segundos transcurridos desde el 1 de enero de 1970.

2.1.2 Ejemplo

El código fuente consiste en reservar el uso del dispositivo con un número mayor 60 y un número menor 0. La llamada de un *read* sobre este archivo consiste en devolver la hora actual.

```
/* Este módulo cargable permite dar la hora con un simple
   gandalf# cat /dev/time */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/module.h>
#include <linux/sched.h>
```

```
/* Números mayor y menor */
#define HOUR_MAJOR 60
#define HOUR_MINOR 0
/* Operaciones de entrada/salida en el dispositivo */
static int time_fs_read (struct inode *inode,
                        struct file *file, char *buf, int nbytes);

static struct file_operations time_fops =
{
    NULL,
    time_fs_read,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};

/* Inicialización */
int hora_init (void)
{
    if (register_chrdev (HOUR_MAJOR, "time", &time_fops)) {
        printk ("time: unable to get major %d\n", HOUR_MAJOR);
        return -EIO;
    }
    printk ("time module loaded (major=%d)\n", HOUR_MAJOR);
    return 0;
}

#ifdef MODULE
/* Inicialización del módulo */
int init_module (void)
{
    return hora_init ();
}

/* Descarga del módulo */
void cleanup_module (void)
{
    unregister_chrdev (HOUR_MAJOR, "time");
    printk ("time module unloaded\n");
}
#endif
```



```
/* Lectura del dispositivo: basta con devolver la hora actual
*/
static int time_fs_read (struct inode *inode, struct file
                        *file, char *buf, int nbytes)
{
    unsigned int    minor = MINOR (inode->i_rdev);
    char            tmp[32];

    if (nbytes == 0)
        return 0;

    if (minor != HOUR_MINOR)
        return -ENODEV;

    MOD_INC_USE_COUNT; /* un proceso más utiliza este módulo */

    sprintf (tmp, "%d\n", (int) (CURRENT_TIME));
    memcpy_tofs (buf, tmp, strlen (tmp));

    MOD_DEC_USE_COUNT; /* Utilización terminada*/

    return strlen (tmp) + 2; /* Número de bytes utilizados
                             en la memoria intermedia */
}
```

2.1.3 Compilación y ejecución

La compilación se efectúa de manera bastante simple. Se trata en principio de compilar los archivos objeto y seguidamente generar el archivo:

```
gandalf# make
gcc -c -D__KERNEL__ -I/usr/src/linux-pre2.0.8/include -Wall
-Wstrict-prototypes -O2 -fomit-frame-pointer -fno-strength-reduce
-pipe -m486 -malign-loops=2 -malign-jumps=2 -malign-functions=2 -DCPU=686
-DMODULE module-hora.c

ld -m elf_i386 -r -o mimodulo.o module-hora.o
```

Las opciones importantes son `-DMODULE` y `-D__KERNEL__`, que especifican que el código será generado en forma de un módulo cargable. Las otras opciones de este ejemplo son las que se utilizan en la compilación del núcleo (más exactamente, la máquina utilizada en este ejemplo es un PC dotado de un procesador 686).

La creación del archivo, la carga del módulo y la ejecución se hacen muy simplemente:

```
gandalf# mkmod /dev/time c 60 0
gandalf# insmod module-hora
gandalf# lsmod
Module:                #pages:Used by:
module-hora            1                0
gandalf# cat /dev/time | head --lines=1
833371919
gandalf# rmmod module_hora
```

Se trata en un primer momento de crear el dispositivo con el número mayor 60 y el número menor 0. Seguidamente, se trata de usar los mandatos especialmente dedicados a la manipulación de los módulos cargables.

Se han dejado ciertos rastros en este ejemplo. En el uso anterior, los mensajes siguientes pueden encontrarse en el archivo de rastreo. Esto depende de la configuración del demonio *syslogd* pero a menudo se trata de archivos */var/adm/messages* o */var/adm/syslog*.

```
May 29 13:33:25 gandalf kernel: time module loaded (major=60)
May 29 13:33:30 gandalf kernel: time module unloaded
```

2.2 Las llamadas al sistema específicas de los módulos

Existen cuatro llamadas al sistema para la manipulación de los módulos cargables, pero no están definidos en ninguna biblioteca. Estas llamadas al sistema las utilizan los programas de manipulación de módulos cargables, como *insmod*, etc. Hay que destacar que sólo el superusuario puede usarlos. Las declaraciones de las llamadas, las estructuras y las macroinstrucciones se encuentran en el archivo de cabecera *<linux/module.h>*.

2.2.1 *get_kernel_syms*

Esta llamada al sistema cuenta con varias funcionalidades: puede devolver el número de símbolos disponibles en el núcleo, o bien devolver uno de estos símbolos.

El prototipo de esta llamada es el siguiente:

```
#include <linux/module.h>

int get_kernel_syms (struct kernel_sym *table);
```

Si la tabla pasada como parámetro es *NULL*, la llamada al sistema devolverá el número de símbolos que están disponibles actualmente en el núcleo. Se puede usar esta técnica para asignar el espacio de memoria necesario para crear la tabla receptora de los datos.

La estructura `kernel_sym` posee la definición siguiente:

tipo	campo	descripción
unsigned long	value	Valor del símbolo
char [SYM_MAX_NAME]	name	Nombre del símbolo

Si este parámetro se pasa a la llamada al sistema, todos los símbolos y nombres de módulos conocidos por el núcleo se copian en la tabla. Las entradas se ordenan en el orden LIFO (*Last In First Out*) de los módulos cargados en el núcleo. El campo *value* contiene la dirección en el núcleo de la estructura que describe el módulo.

2.2.2 *create_module*

Esta llamada al sistema sirve para asignar un cierto número de bytes en el espacio de direccionamiento del núcleo para poder colocar el módulo en él. Además, esta llamada crea también las estructuras del núcleo para gestionar este módulo. El módulo existe entonces en el núcleo, pero con el estatus `MOD_UNINITIALIZED`, es decir, que no es utilizable por el momento.

El prototipo de esta llamada es el siguiente:

```
#include <linux/module.h>

int create_module (char *module_name, unsigned long size);
```

2.2.3 *init_module*

Esta llamada corresponde al cargador de un módulo, y permite la carga del módulo en el núcleo. Su prototipo es el siguiente:

```
#include <linux/module.h>

int init_module (char *module_name, char *code,
                 unsigned codesize, struct mod_routines *routines,
                 struct symbol_table *symtab);
```

Esta llamada carga el módulo que tiene por nombre *name* en el núcleo. El código del módulo está contenido en el parámetro *code* de tamaño *codesize*. Hasta después de esta llamada el módulo no es operativo. Los dos parámetros que siguen contienen punteros a las rutinas de inicialización y descarga del módulo. La estructura `mod_routines` está compuesta por los campos siguientes:

tipo	campo	descripción
int (*) (void)	init	Rutina de inicialización
void (*) (void)	cleanup	Rutina de supresión con descarga

El tamaño de los símbolos del módulo pasado como parámetro se expresa por la estructura `symbol_table`, cuya definición damos a continuación:

tipo	campo	descripción
int	size	Tamaño total, incluyendo la tabla de cadenas
int	n_symbols	Número de símbolos
int	n_refs	Número de referencias
struct internal_symbol[0]	symbol	Tabla de símbolos
struct module_ref[0]	ref	Tabla de referencias

Cada símbolo de esta tabla se representa por la estructura `internal_symbol`:

tipo	campo	descripción
void *	addr	Dirección del símbolo
const char *	name	Nombre del símbolo

Cada módulo se referencia de la forma siguiente:

tipo	campo	descripción
struct module *	module	Puntero a la estructura del módulo
struct module_ref *	next	Puntero al módulo siguiente

Se trata de una estructura interna del núcleo.

Para terminar, cada módulo se representa en forma de la estructura siguiente:

tipo	campo	descripción
struct module *	next	Puntero al módulo siguiente
struct module_ref *	ref	Lista de módulos que utilizan este módulo
struct symbol_table *	syntab	Tabla de símbolos del módulo
const char *	name	Nombre del módulo
int	size	Tamaño del módulo en páginas de memoria
void *	addr	Dirección del módulo
int	state	Estado del módulo
void (*) (void)	cleanup	Rutina lanzada en la descarga

Un módulo cargable puede encontrarse en tres estados. Sólo el estado `MOD_RUNNING` permite el uso del módulo.

constante	significado
MOD_UNINITIALIZED	Módulo creado pero no inicializado
MOD_RUNNING	Módulo en curso de utilización
MOD_DELETED	Módulo destruido

2.2.4 *delete_module*

Esta llamada descarga el módulo con determinado nombre del núcleo. Hay que destacar que esta operación puede fallar si algunos procesos siguen accediendo al módulo.

El prototipo de esta llamada es el siguiente:

```
#include <linux/module.h>

int delete_module (char *module_name);
```

3 Implementación de los módulos cargables

3.1 Presentación

La implementación de los módulos cargables se descompone en varias partes:

- la implementación de las llamadas al sistema en *kernel/module.c*;
- la gestión de los archivos virtuales */proc/modules* y */proc/ksyms* en *kernel/module.c*;
- el sistema de comunicación con *kerneld* en *ipc/msg.c*;
- el demonio *kerneld*.

El meollo del dispositivo se encuentra en el archivo *kernel/module.c*. Este módulo gestiona por ejemplo la lista de módulos cargados en el núcleo. Se trata de la variable *module_list* de tipo estructura *module*.

Al arrancar el sistema, se lanza la función *init_modules*. Esta función consiste en inicializar las variables globales (o estáticas) tales como:

- *module_list*: lista de módulos cargables;
- *symbol_table*: tabla de símbolos del núcleo.

3.2 Implementación de las llamadas al sistema

3.2.1 *create_module*

Esta llamada al sistema consiste en asignar la memoria necesaria (se trata de una asignación en la zona de memoria del núcleo) para albergar el código del módulo cargable.

Tras la asignación de memoria, se crea un nuevo elemento de la lista de módulos, con un estado fijado a `MOD_UNINITIALIZED`. Este elemento se inserta entonces en la lista `module_list`.

3.2.2 *init_module*

Esta llamada efectúa la carga física del módulo. La llamada de carga empieza por poner el código, que se pasa como parámetro, en la lista de módulos (campo `addr` que corresponde al espacio de memoria asignada mediante *create_module*). De este modo, el código pasa a formar parte integral del núcleo.

La parte más compleja de la carga de un módulo no reside en la propia carga (que es muy simple, como hemos visto), sino en la actualización de la tabla de símbolos del núcleo.

En efecto, el núcleo Linux contiene una tabla de símbolos de funciones para saber en qué dirección se encuentra tal o cual función en el núcleo. Por tanto, una vez cargado el código del núcleo, hay que actualizar la tabla de símbolos para indicar la dirección en la que se encuentra cada una de las funciones del módulo.

La operación consiste en crear una nueva tabla de símbolos de tipo `symbol_table` que se detalla en la sección 2.2.3. De hecho, se trata de una duplicación de la tabla de símbolos que contiene el módulo porque se pasa como parámetro. Esta tabla se asocia al módulo residente en el núcleo. De este modo, todo símbolo incluido en el módulo se hace accesible a toda función del núcleo.

3.2.3 *delete_module*

La destrucción de un módulo es una operación particularmente delicada. Esta llamada utiliza dos funciones que se detallarán ulteriormente: `get_mod_name` y `find_module` que recuperan respectivamente el nombre del núcleo y un puntero a la estructura del módulo.

Esta llamada puede tener dos comportamientos. Puede descargar un solo módulo, si se especifica su nombre. Si no se especifica un nombre, recorre toda la lista de módulos e intenta descargar los módulos que no se están usando, uno por uno.

Tras recuperar el nombre y el puntero al módulo, y si el módulo no se usa, el campo de estado del módulo se pone a `MOD_DELETED`. Entonces se efectúa la llamada a la función de descarga `free_modules`.

Esta función recorre la lista de todos los módulos y destruye los marcados `MOD_DELETED`. La destrucción de un módulo se efectúa principalmente en dos etapas. En un primer momento, si hay referencias al módulo utilizadas por otros módulos, se efectúa una llamada a `drop_refs`. Esta función ordena las referencias entre los módulos. En un segundo momento, la función `free_modules` destruye la tabla de símbolos del módulo, y suprime el código del módulo. El módulo se considera entonces «descargado» porque su código ya no existe en el núcleo.

3.2.4 *get_kernel_syms*

Esta llamada al sistema recorre, en un primer momento, la lista de módulos cargados, y cuenta el número de símbolos presentes. Esta operación es muy simple porque cada módulo conoce el número de símbolos que posee (campo `n_symbols` de la tabla de símbolos del módulo).

Seguidamente, si la tabla pasada como parámetro no es nula, todos los nombres y todas las direcciones de los diferentes símbolos contenidos en los módulos cargados se copian en él. Esta operación se realiza por un recorrido de la lista. Sólo los módulos en el estado `MOD_RUNNING` se tienen en cuenta.

3.3 Gestión de los archivos virtuales

Los archivos virtuales */pro/modules* y */proc/ksyms* se implementan en el archivo fuente *kernel/module.c*. Estos dos archivos sólo se pueden consultar en lectura.

Al leer el archivo */proc/modules*, se llama a la función `get_module_list`, que recorre la lista de módulos y llena la memoria intermedia pasada como parámetro. Esta memoria intermedia es en realidad el texto que se mostrará en la consulta de este archivo.

Al consultar el archivo *ksyms*, se llama a la función `get_ksyms_list`. Asimismo, un recorrido de la lista de módulos y de cada una de sus tablas de símbolos permite constituir la lista de símbolos.

3.4 Funciones anexas

Ciertas funciones se usan en este módulo (o en el exterior), pero su interés es algo menor:

- `get_mod_name`: copia el nombre de un módulo desde el espacio de usuario al espacio de núcleo;
- `find_module`: devuelve un puntero cuyo nombre se pasa como parámetro al módulo;
- `register_symtab_from`: inserta una tabla de símbolos en la lista.

4 Carga automática de módulos (kerneld)

4.1 Presentación

El sistema que permite cargar de manera automática los módulos cargables es bastante complejo, porque utiliza varios mecanismos. El principio de la carga automática es relativamente simple. El núcleo en un momento dado necesita ciertas funcionalidades que se encuentran en un módulo cargable. Para ello hay que cargarlo. La operación de carga se efectúa realmente mediante un proceso en modo usuario, *kerneld*. El orden de carga se envía del núcleo al demonio a través de una cola de mensajes (véase el capítulo 11, sección 2.1).

Como se esquematiza en la figura 12.2, la operación de carga se efectúa en varias etapas:

- El núcleo necesita efectuar la carga de un módulo. Para ello, utiliza la función `request_module`.
- Se envía un mensaje a la cola de mensajes. El contenido de este mensaje está constituido por la constante `KERNELD_REQUEST_MODULE`, así como por el nombre del módulo deseado.
- El demonio lee el mensaje y efectúa la carga del módulo llamando al programa *modprobe*.
- El demonio envía eventualmente una respuesta al núcleo.

Esta forma de cargar los módulos presenta ciertas ventajas. Puede darse el caso que el demonio tenga varias órdenes de carga en la cola de mensajes y, en este caso, los módulos se cargan todos a la vez. Además, el código del núcleo es relativamente reducido porque las operaciones de carga son externas al núcleo. Finalmente, la gestión de errores se reduce en gran medida. Sin embargo, el hecho que sea un demonio ejecutado en modo usuario que, por tanto, puede interrumpir quien efectúa el trabajo de carga, puede provocar la puesta en espera de los procesos que han provocado la carga. Esta espera finaliza cuando el demonio consiga finalmente cargar los módulos deseados.

4.2 Detalle de la implementación

La implementación se distribuye en los módulos *ipc/msg.c* y *<linux/kernld.h>*. Aunque este último es un archivo de cabecera, se definen en él ciertas funciones. La mayor parte de la implementación se encuentra de hecho en el demonio, cuyos recursos se distribuyen por separado. La sede primaria de distribución es <http://www.pi.se/blox/modules/>.

Veamos en detalle las funciones contenidas en el archivo de cabecera.

- **request_module**: carga un módulo cuyo nombre se pasa como parámetro. Se utiliza para ello el mandato `KERNELD_REQUEST_MODULE`. Además, una opción particular (`KERNELD_WAIT`) permite salir de la función únicamente cuando el módulo se carga, o bien se produce un error. Este mandato va destinado típicamente a transformarse en una llamada a *modprobe* por el demonio.
- **release_module**: solicita la descarga de un módulo. Esta operación consiste en utilizar el mandato `KERNELD_RELEASE_MODULE`. El demonio lanza entonces el programa *rmmod* para efectuar la operación.
- **delayed_release_module**: si el módulo no se usa durante cierto tiempo (en principio 60 segundos), el módulo se descarga. El mandato utilizado para realizar esta operación es `KERNELD_DELAYED_RELEASE_MODULE`, lo que lleva a la ejecución de *rmmod*.
- **cancel_release_module**: intenta anular una petición de descarga de un módulo. Esta llamada puede realizarse si el núcleo quiere evitar la descarga automática del módulo, mediante el mandato `KERNELD_CANCEL_RELEASE_MODULE`.
- **kssystem**: efectúa una llamada al sistema denominada «inversa». Permite transmitir ciertas informaciones al demonio, mediante el mandato `KERNELD_SYSTEM`.

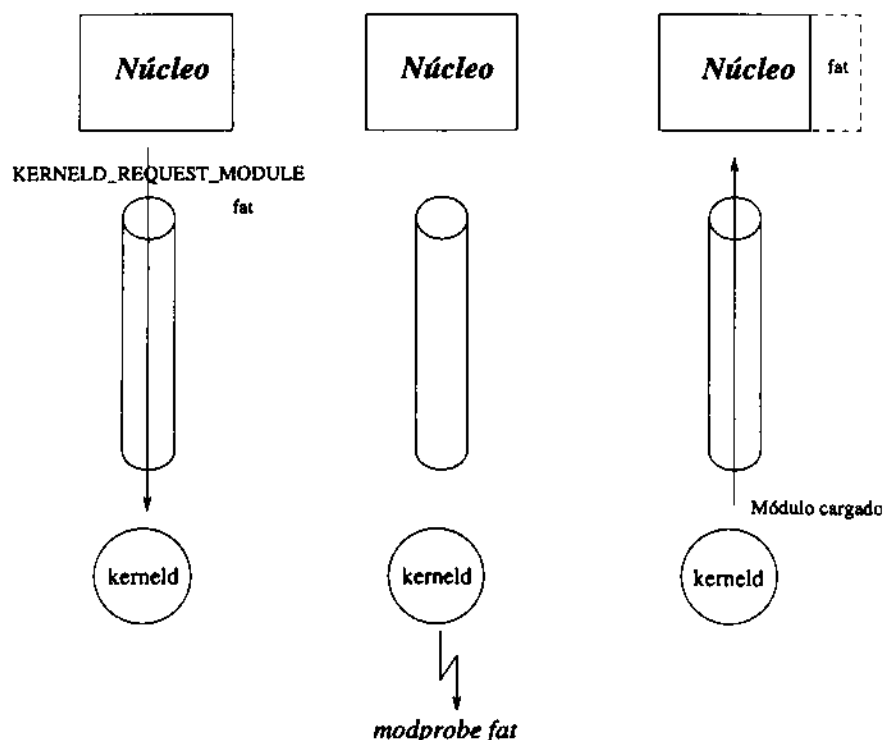


FIG 12.2 – Comunicación entre el núcleo y kerneld

- **kerneld_route**: especialmente diseñada para la red, esta función permite crear una entrada en la tabla de enrutamiento, y se usa por ejemplo en conexiones de red que usen el protocolo PPP. El mandato utilizado es **KERNELD_REQUEST_ROUTE**.
- **kerneld_blanker**: permite utilizar un protector de pantalla externo, mediante el mandato **KERNELD_BLANKER**.

Este archivo de cabecera define también la estructura **kerneld_msg**, que contiene el mensaje que transita entre el núcleo y el demonio. Esta estructura posee los campos siguientes:

tipo	campo	descripción
long	mttype	Tipo del mandato
long	id	Si este campo no es nulo, el núcleo espera una respuesta
short	version	Versión del protocolo. Si este campo está a 0, se trata del protocolo antiguo. El protocolo utilizado actualmente usa el número 2
short	pid	Pid del proceso que provoca la carga del módulo
char *	text	Paso de información

Todas las funciones expuestas anteriormente se basan principalmente en la función `kernel_d_send`, que se encuentra en el módulo `ipc/msg.c`. Esta última es la única que no ha sido detallada en el capítulo sobre los IPC. Consiste en enviar el mensaje deseado a la cola de mensajes de tipo `IPC_KERNELD`. Se trata de una cola de mensajes reservada a la comunicación entre el núcleo y el demonio. El envío y la recepción de mensajes se efectúan por medio de las funciones `real_msgsnd` y `real_msgrcv`. Estas dos funciones se han detallado en el capítulo 11, sección 5.1.4. Los datos que circulan en esta cola de mensajes se expresan a través de la estructura `kernel_d_msg`.

La implementación del demonio *kernel_d* sobrepasa el marco de esta obra, porque es externa al núcleo. Sin embargo, su funcionamiento es muy simple. Puede resumirse de la forma siguiente: mediante una espera activa en lectura sobre la cola de mensajes, recibe las órdenes de lanzamiento o descarga de módulos. Los datos pasados en la cola de mensajes se expresan a través de la estructura `kernel_d_msg`, que se detalla en la sección 4.2. Sin embargo, el demonio no carga realmente los módulos: utiliza los programas previstos a tal efecto.

Capítulo 13

Administración del sistema

Primitivas detalladas

`adjtimex, gettimeofday, gethostname,
getdomainname, reboot, setdomainname,
sethostname, settimeofday, stime,
syslog, sysctl, time, uname`

1 Conceptos básicos

El núcleo Linux mantiene ciertas informaciones de una manera permanente, pues ciertas informaciones son vitales no sólo para el correcto funcionamiento del núcleo, sino también para el correcto desarrollo de los procesos.

1.1 Informaciones destinadas a los procesos

Ciertos datos se ponen a disposición de los procesos. Al estar situados en el espacio de direccionamiento del núcleo, están a la vez protegidos y son fácilmente accesibles. Encontramos aquí el nombre de la máquina, del ámbito y del sistema operativo, así como el número de versión, la fecha en que fue compilado, etc.

Hay que destacar que el nombre de ámbito es bastante ambiguo. Se trata tanto del nombre del ámbito en el DNS (*Domain Name System*) como del nombre de NIS (*Networked Information Service*, antiguamente llamado Páginas amarillas; véase [Moreno 1995] para más detalles). Se presenta un problema en la manipulación para

los programas que se sirven de esta información: un programa puede utilizar el nombre de ámbito en un caso u otro, pero el efecto no es el mismo.

Estos datos son accesibles directamente mediante los mandatos *hostname*, *domainname* y *uname*.

Los dos primeros cumplen una función particular:

- dar el valor:

```
gandalf%% hostname
gandalf
```

- fijar el valor, si se trata del superusuario:

```
gandalf# domainname freenix.fr
gandalf# domainname
freenix.fr
```

El mandato *uname* permite dar estas informaciones, y posee numerosas opciones. Por ejemplo, para acceder a todas las informaciones, se escribirá:

```
gandalf# uname -a
Linux gandalf 2.0.14 #1 Wen Aug 21 00:37:52 MET DST 1996 i486
```

La llamada al sistema que permite acceder a estos datos es *uname*.

1.2 Informaciones que controlan la ejecución

Diferentes tipos de informaciones que controlan la ejecución se conservan en el núcleo:

- *El tiempo*: se trata del número de segundos transcurridos desde el 1 de enero de 1970 00:00 UTC. Se usan principalmente dos llamadas para acceder a este dato: *gettimeofday* y *settimeofday*. Sin embargo, Linux proporciona otras dos llamadas por razones de compatibilidad: *time* y *stime*.
- *El modo de visualización de los mensajes del núcleo*: el núcleo Linux devuelve mensajes para informar al administrador de las operaciones efectuadas sobre la máquina. La llamada al sistema *syslog* permite definir el nivel de detalle de los mensajes a mostrar.
- *Las informaciones sobre el estado de la máquina*: ciertas informaciones son útiles para conocer el estado de rendimiento y de funcionamiento de la máquina. Se trata de informaciones como la memoria libre, la carga, el *swap* utilizado, etc. La llama-

da al sistema *sysinfo* da estas informaciones. Una parte de las informaciones puede visualizarse mediante los mandatos *uptime* y *free*.

- **Sincronización de relojes:** existe un sistema que permite sincronizar el reloj del sistema: *xntpd*. Los relojes de los ordenadores atrasan (o adelantan) a largo plazo. Cuando se cuenta con varias máquinas, es muy difícil conseguir tener la misma hora. *xntpd* permite sincronizarlos. Este demonio utiliza una llamada al sistema particular: *adjtimex*. El núcleo conserva ciertos datos útiles como la precisión del reloj, su frecuencia, etc. Este sistema se basa en el protocolo *NTP* (*Network Time Protocol*) que se detalla en [Mills 1992].

1.3 Cambio de estado del núcleo

Una llamada al sistema particular modifica el estado de la máquina: *reboot*. ¡Puede resultar útil rearrancar la máquina de vez en cuando (para cambiar de versión de núcleo, por ejemplo)! Diversos mandatos Unix permiten efectuar esta operación (*reboot*, *shutdown*, etc.). Hay que destacar una especificidad de Linux que permite reiniciar la máquina mediante el célebre Ctrl-Alt-Supr (opción conocida con el sobre-nombre de «saludo de los tres dedos»).

1.4 Configuración dinámica del sistema

1.4.1 Presentación

Linux proporciona una llamada al sistema muy potente, *sysctl*, que permite configurar de manera dinámica ciertos parámetros del sistema, como el nombre de la máquina, el número máximo de archivos abiertos, la gestión del *swap*, etc.

1.4.2 Otro método: */proc/sys*

La configuración dinámica del sistema puede realizarse también mediante el sistema de archivos virtual */proc* cuyo funcionamiento se ha detallado en el capítulo 6, sección 6.6. Todas las informaciones que pueden ser accesibles por la llamada al sistema *sysctl* están disponibles en forma de archivos en el directorio */proc/sys*.

Por ejemplo, ciertas informaciones relativas a la máquina son accesibles directamente:

- */proc/sys/kernel/hostname*: nombre de la máquina

```
gandalf# cat hostname
gandalf
```

- `/proc/sys/kernel/domainname`: nombre del ámbito

```
gandalf# cat version
#1 Wed Aug 21 00:37:52 MET DST 1996
```

2 Llamadas al sistema de administración

Linux permite a los procesos de usuario manipular las informaciones mantenidas por el núcleo. Todo proceso puede acceder en lectura a estos datos pero sólo los procesos privilegiados pueden modificarlos.

2.1 Informaciones respecto a la estación

Las diversas informaciones respecto a la propia estación, es decir, su nombre, su ámbito, la versión del sistema..., pueden obtenerse mediante la llamada al sistema `uname`. Esta llamada utiliza la estructura `utsname` definida en el archivo de cabecera `<sys/utsname.h>`. Los campos que componen esta estructura son los siguientes:

tipo	campo	descripción
char *	<code>sysname</code>	Nombre del sistema operativo
char *	<code>nodename</code>	Nombre de la estación
char *	<code>release</code>	Número de la versión del sistema operativo
char *	<code>version</code>	Número de revisión del núcleo y fecha de compilación
char *	<code>machine</code>	Identificador de la plataforma de hardware
char *	<code>domainname</code>	Nombre del ámbito al cual pertenece la estación

La sintaxis de la llamada al sistema `uname` es la siguiente:

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

int uname (struct utsname *buf);
```

Las informaciones se devuelven en la estructura cuya dirección se pasa como parámetro. La llamada al sistema `uname` devuelve el valor 0 en caso de éxito, o el valor -1 en caso de fallo (el único código de error que puede devolverse es `EFAULT` en el caso en que el parámetro `buf` contenga una dirección inválida).

El programa siguiente muestra los datos devueltos por *uname*. Es un equivalente simplificado del mandato *uname*.

```
#include <stdio.h>
#include <sys/utsname.h>

void main (void)
{
    struct utsname  buf;
    int             err;

    err = uname (&buf);
    if (err != 0)
        printf ("Error uname\n");
    else {
        printf ("sysname      = %s\n", buf.sysname);
        printf ("nodename     = %s\n", buf.nodename);
        printf ("release      = %s\n", buf.release);
        printf ("version      = %s\n", buf.version);
        printf ("machine      = %s\n", buf.machine);
        printf ("domainname   = %s\n", buf.domainname);
    }
}
```

La ejecución de este programa provoca el siguiente resultado:

```
bbj>ShowUname
sysname= Linux
nodename      = bbj
release= 1.2.13
version= #1 Fri Aug 11 18:50:10 1995
machine= i486
domainname    = freenix.fr
```

Un proceso privilegiado puede modificar ciertas informaciones obtenidas por una llamada a *uname*. Las llamadas al sistema *sethostname* y *setdomainname* permiten modificar respectivamente el nombre de la estación y el nombre de su ámbito. La sintaxis de estas llamadas al sistema es la siguiente:

```
#include <unistd.h>

int sethostname (const char *name, size_t len);
int setdomainname (const char *name, size_t len);
```

Estas dos llamadas al sistema toman como argumento el nuevo nombre de la estación o del ámbito (parámetro *name*), así como el número de caracteres que componen dicho

nombre (parámetro `len`). En caso de fallo, el error `EPERM` indica que el proceso que llama no posee los privilegios requeridos para modificar estas informaciones, y el error `EINVAL` indica que el tamaño del nombre es superior al límite del sistema.

El programa siguiente modifica el nombre de la estación y de su ámbito según los parámetros que se le pasan: primero muestra estas informaciones, las modifica y las muestra de nuevo.

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

void main (int argc, char *argv[])
{
    struct utsname  buf;
    int             err;

    if (argc != 3) {
        printf ("Uso: %s nombre_de_estación nombre_de_ámbito\n",
                argv[0]);
        exit (1);
    }
    err = uname (&buf);
    if (err != 0)
        printf ("Error uname\n");
    else {
        printf ("nodename    = %s\n", buf.nodename);
        printf ("domainname = %s\n", buf.domainname);
    }
    err = sethostname (argv[1], strlen (argv[1]));
    if (err != 0)
        printf ("Error sethostname\n");
    err = setdomainname (argv[2], strlen (argv[2]));
    if (err != 0)
        printf ("Error setdomainname\n");
    err = uname (&buf);
    if (err != 0)
        printf ("Error uname\n");
    else {
        printf ("nodename    = %s\n", buf.nodename);
        printf ("domainname = %s\n", buf.domainname);
    }
}
```

Su ejecución provoca la visualización siguiente:

```
bbj# ./ShowUtsname station domaine
nodename      = bbj
domainname    = freenix.fr
nodename      = station
domainname    = domaine
```

También existen dos funciones llamadas *gethostname* y *getdomainname*, que permiten obtener el nombre de la estación y el de su ámbito. Estas dos funciones se implementan en la biblioteca C y no son llamadas al sistema tratadas por el núcleo: *gethostname* y *getdomainname* se limitan a devolver las informaciones devueltas por la llamada al sistema *uname*. Su sintaxis es la siguiente:

```
#include <unistd.h>

int gethostname (char *name, size_t len);
int getdomainname (char *name, size_t len);
```

2.2 Control de la ejecución

2.2.1 El tiempo

Las llamadas al sistema *gettimeofday* y *settimeofday* permiten recuperar el tiempo transcurrido desde el 1 de enero de 1970, o por el contrario fijar este lapso. Otras dos llamadas al sistema pueden utilizarse para ello: *time* y *stime*. Estas dos llamadas sólo se conservan por razones de compatibilidad, porque *gettimeofday* y *settimeofday* pueden realizar la misma operación.

La sintaxis de estas dos llamadas es la siguiente:

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

int gettimeofday (struct timeval *tv, struct timezone *tz);
int settimeofday (const struct timeval *tv, const struct time
zone *tz);
```

Se usan dos estructuras, que influyen en el desarrollo de la llamada al sistema en función de su presencia. Para poder manipular estas dos estructuras, hay que incluir el archivo *<sys/time.h>* (estas dos estructuras se declaran en realidad en el archivo *<linux/time.h>*).

La estructura `timeval` permite expresar el tiempo en número de segundos y de microsegundos transcurridos desde el 1 de enero de 1970. Su formato se define en el capítulo 4, sección 2.3.

La otra estructura permite fijar o pedir la diferencia horaria (huso horario), mediante la estructura `timezone`:

tipo	campo	descripción
int	tz_tz_minuteswest	Número de minutos de diferencia respecto al oeste del meridiano de Greenwich
int	tz_dsttime	Diferencia respecto al huso horario

Veamos un ejemplo para recuperar las informaciones necesarias:

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

void main ()
{
    struct timeval tv;
    struct timezone tz;

    if (gettimeofday (&tv, NULL) == -1) {
        perror ("gettimeofday");
        exit (-1);
    }
    printf ("Núm. de segundos: %d Núm. de microsegundos: %d\n",
           tv.tv_sec, tv.tv_usec);

    if (gettimeofday (NULL, &tz) == -1) {
        perror ("gettimeofday");
        exit (-1);
    }
    printf ("Núm. de minutos (Oeste Greenwich): %d Huso: %d\n",
           tz.tz_minuteswest, tz.tz_dsttime);
}
```

La ejecución de este programa puede dar informaciones diferentes según la configuración de la máquina, y el período del año en que se ejecute este programa:

```
gandalf# ./gettimeofday
Núm. de segundos: 846708854 Número de microsegundos: 626083
Núm. de minutos (Oeste Greenwich): -60 Huso: 1
gandalf#
```

Este ejemplo se ha ejecutado a finales de octubre. En esta época, sólo hay una hora de diferencia horaria, como lo muestra el número -60. Si este programa se ejecutara en verano, se indicarían dos horas de diferencia.

Los errores que pueden producirse en esta llamada son los siguientes:

<i>error</i>	<i>significado</i>
EINVAL	Parámetros no válidos

La llamada al sistema *settimeofday* sólo puede ser ejecutada por el superusuario. Para fijar la hora o el huso horario, basta con realizar la operación inversa. Esta llamada puede generar los errores siguientes:

<i>error</i>	<i>significado</i>
EPERM	Sólo puede ser ejecutado por el superusuario
EINVAL	Parámetros no válidos

Las llamadas *time* y *stime* tienen los prototipos siguientes:

```
#include <time.h>

time_t time(time_t *t);
int stime (type_t *t);
```

time devuelve el número de segundos transcurridos desde el 1 de enero de 1970 y *stime* fija esta duración. *stime* sólo puede ser ejecutado por el superusuario.

2.2.2 Modo de visualización de los mensajes del núcleo

La llamada al sistema *syslog* no debe confundirse con la función de biblioteca *syslog* que envía un mensaje al demonio *syslogd*. Esta llamada al sistema permite configurar el modo de visualización de los mensajes del sistema y manipular la memoria intermedia de mensajes, y es específica de Linux.

El prototipo de esta llamada es el siguiente:

```
#include <unistd.h>
#include <linux/unistd.h>

_syscall3 (int, syslog, int, type, char *, bufp, int, len);
int syslog (int type, char *bufp, int len);
```

El primer parámetro permite especificar el mandato que se quiere realizar:

- 0: detiene la gestión de mensajes;

- 1: inicia la gestión de mensajes;
- 2: copia los mensajes en la memoria intermedia `bufp` que posee el tamaño `len`;
- 3: copia los últimos mensajes concurrentes de 4 KB;
- 4: lo mismo pero suprime los mensajes leídos de más;
- 5: vacía la memoria intermedia de mensajes;
- 6: desactiva la visualización en la consola;
- 7: activa la visualización en la consola;
- 8: fija el nivel de visualización de mensajes en la consola.

Hay que destacar que la única llamada que un usuario sin privilegios puede efectuar es la 3.

Sin embargo, no se puede usar esta llamada directamente, porque la función de la biblioteca se utiliza prioritariamente en el enlazado. Veamos un ejemplo de su uso:

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

/* se define una llamada con el mismo número */
#define __NR_monsyslog __NR_syslog
static inline
    _syscall3 (int, monsyslog, int, type, char *, b, int, len);

void main ()
{
    char buf[4 * 1024];

    if (monsyslog (3, buf, 4 * 1024) == -1) {
        perror ("syslog");
        exit (-1);
    }
    printf ("%s \n", buf);
}
```

El ejemplo anterior equivale al mandato *dmesg* que muestra todos los mensajes almacenados por el núcleo. Veamos un extracto del resultado obtenido:

```

gandalf# ./demosyslog
<4>Console: 16 point font, 400 scans
<4>Console: colour VGA+ 80x25, 1 virtual console (max 63)
<4>Calibrating delay loop.. ok - 32.87 BogoMIPS
<4>Memory: 14680k/16384k available (796k kernel code, 384k reserved, 524k
<4>Linux version 1.3.100 (dumas@gandalf) (gcc version 2.7.2) #1 Sat May 11
15:08:56 MET DST 1996
<6>Serial driver version 4.12 with no serial options enabled
<6>tty00 at 0x03f8 (irq = 4) is a 16450
<6>tty01 at 0x02f8 (irq = 3) is a 16450
<6>lpt at 0c0378, (polling)
<4>Sound initialization started
<4><SoundBlaster 16 4.11> at 0x260 irq 5 dma 1,5
<7>Max size:314389 Log zone size:2048
<7>First datazone:68 Root inode number 139264
<4>ISO9660 Extensions: RRIP_1991A

```

La cifra situada entre corchetes corresponde al nivel del mensaje. Este nivel es el utilizado en la llamada a la función *syslog* de la biblioteca estándar:

constante	significado
KERN_EMERG (0)	Sistema no utilizable
KERN_ALERT (1)	Mensaje de alerta
KERN_CRIT (2)	Mensaje crítico
KERN_ERR (3)	Mensaje de error
KERN_WARNING (4)	Advertencia
KERN_NOTICE (5)	Mensajes diversos
KERN_INFO (6)	Información
KERN_DEBUG (7)	Depuración

Esta llamada al sistema puede generar los errores siguientes:

error	significado
EPERM	Uso de <i>syslog</i> con un mandato que sólo puede usar el superusuario
EINVAL	Parámetro incorrecto
EINTR	Interrupción de la llamada al sistema. No se copia ningún dato en la memoria

2.2.3 Estado de la máquina

El estado de la máquina se obtiene utilizando la llamada al sistema *sysinfo*. Esta llamada se utiliza por programas como *uptime*.

Su prototipo es el siguiente:

```

#include <linux/kernel.h>
#include <linux/sys.h>

int sysinfo (struct sysinfo *info);

```

La estructura `sysinfo` posee los campos siguientes:

tipo	campo	descripción
long	uptime	Número de segundos transcurridos desde el arranque de la máquina
unsigned long [3]	loads	Carga media de la máquina durante el último minuto, los últimos cinco y los últimos quince
unsigned long	totalram	Memoria disponible
unsigned long	freeram	Memoria libre
unsigned long	sharedram	Memoria compartida
unsigned long	bufferram	Memoria utilizada por las memorias intermedias
unsigned long	totalswap	Tamaño total del <i>swap</i>
unsigned long	freeswap	Espacio disponible en el <i>swap</i>
unsigned short	procs	Número actual de proceso
char [22]	_f	Alineación para que la estructura ocupe 64 bytes

Esta llamada al sistema se usa de manera muy simple:

```
#include <linux/kernel.h>
#include <linux/sys.h>
#include <stdio.h>
#include <stdlib.h>

void main ()
{
    struct sysinfo si;

    if (sysinfo (&si) == -1) {
        perror ("sysinfo");
        exit (-1);
    }
    printf ("Resultado de la llamada a sysinfo \n");
    printf ("Número de segundos desde el arranque: %ld \n",
        si.uptime);
    printf ("Carga 1m (%ld) 5m (%ld) 15 (%ld) \n",
        si.loads[0], si.loads[1], si.loads[2]);
    printf ("Memoria (en KB) total %ld Libre %ld Compartida "
        "%ld Intermedia %ld\n",
        si.totalram / 1024, si.freeram / 1024,
        si.sharedram / 1024, si.bufferram / 1024);
    printf ("Tamaño del swap %ld libre %ld \n",
        si.totalswap / 1024, si.freeswap / 1024);
    printf ("Número de procesos %d \n", si.procs);
}
```

En la ejecución, el formato se ha definido en KB porque resulta más legible:

```

gandalf# demosysinfo
Resultado de la llamada a sysinfo
Número de segundos desde el arranque: 1512025
Carga 1m (38752) 5m (30336) 15 (20224)
Memoria (en KB) total 14680 Libre 716 Compartida 11624 Memorias interme-
dias 312
Tamaño del swap 34812 libre 30116
Número de procesos 49

```

El único error que puede generar esta llamada es EFAULT si el puntero pasado como parámetro no es correcto.

2.2.4 Sincronización de relojes

La llamada al sistema *adjtimex* la utiliza principalmente el sistema de sincronización *xntp*. El objetivo de esta llamada es permitir ajustar la hora de la máquina, modificando los parámetros necesarios del núcleo.

El prototipo de esta llamada es el siguiente:

```

#include <sys/timex.h>

int adjtimex (struct timex *buf);

```

Esta llamada toma como parámetro la estructura *timex* cuya declaración compleja se encuentra en el archivo *<linux/timex.h>*. Veamos sus detalles:

tipo	campo	descripción
unsigned int	modes	Selección del modo
long	offset	Tiempo a añadir (microsegundos)
long	freq	Frecuencia
long	maxerror	Error máximo (microsegundos)
long	esterror	Error estimado (microsegundos)
int	status	Mandato o estado del reloj
long	constant	Constante
long	precision	Precisión del reloj (microsegundos) (lectura exclusiva)
long	tolerance	Tolerancia de la frecuencia del reloj (lectura exclusiva)
struct timeval	time	Hora (lectura exclusiva)
long	tick	Número de microsegundos entre dos tics de reloj
long	ppsfreq	Frecuencia (lectura exclusiva)
long	jitter	
int	shift	Duración de un intervalo (lectura exclusiva)
long	stabil	Estabilidad (lectura exclusiva)
long	jitcnt	Desplazamiento de límite (lectura exclusiva)
long	calcnt	Calibre del intervalo (lectura exclusiva)
long	errcnt	Calibre del error (lectura exclusiva)
long	stbcnt	Límite de la estabilidad (lectura exclusiva)

Esta llamada al sistema puede generar los errores siguientes:

<i>error</i>	<i>significado</i>
EFAULT	La memoria intermedia apunta a una zona de memoria que no permite la escritura
EPERM	El campo mode es distinto de cero y la llamada no proviene del superusuario
EINVAL	Se han asignado valores fuera de límites a offset, status o tick

2.3 Cambio de estado del núcleo

La llamada al sistema *reboot* permite a un proceso privilegiado controlar el rearranque del núcleo.

La sintaxis de *reboot* es la siguiente:

```
#include <unistd.h>

int reboot (int magic, int magic_too, int flag);
```

Los dos primeros parámetros (*magic* y *magic_tool*) deben tener los valores 0xfeedad y 672274793 para que la llamada sea tenida en cuenta. El parámetro *flag* especifica la acción a efectuar según su valor:

- 0x01234567: rearranque inmediato del sistema;
- 0xCDEF0123: parada inmediata de todos los procesos y parada del sistema;
- 0x89ABCDEF: activación del rearranque material de la estación al pulsar las teclas Ctrl-Alt-Supr;
- 0: desactivación del rearranque material de la estación al pulsar las teclas Ctrl-Alt-Supr.

Esta llamada puede generar los errores siguientes:

<i>error</i>	<i>significado</i>
EINVAL	Parámetros incorrectos
EPERM	Sólo el superusuario tiene derecho a utilizar esta llamada al sistema

2.4 Configuración dinámica del sistema

Linux permite al superusuario el cambio del comportamiento del sistema, sin tener que recompilar el núcleo ni reconfigurar ciertas partes de la máquina. Esta característica se le especifica a Linux y debe usarse con precaución, porque influye en el funcionamiento de la máquina.

La llamada al sistema *sysctl* posee el prototipo siguiente:

```
#include <unistd.h>
#include <linux/unistd.h>
#include <linux/sysctl.h>

_syscall1 (int, _sysctl, struct __sysctl_args *, args);
int _sysctl (struct __sysctl_args, *args);
```

Esta llamada utiliza la estructura *__sysctl_args* cuya definición es la siguiente:

tipo	campo	descripción
int *	name	Tabla de enteros que describe los datos a leer o modificar
int	nlen	Tamaño de dicha tabla
void *	oldval	NULL o la dirección donde deben guardarse los valores anteriores
size_t *	oldlenp	Tamaño disponible para colocar los valores anteriores
void *	newval	NULL o dirección de los nuevos valores
size_t	newlen	Tamaño de los nuevos valores

El nombre de los datos a los que se desea acceder se expresa por constantes. Se trata en un primer momento de diseñar el tipo de las variables:

constante	significado
CTL_ANY	Cualquier tipo
CTL_KERN	Información general sobre el núcleo y su funcionamiento
CTL_VM	Gestión de la memoria virtual
CTL_NET	Red
CTL_PROC	Información sobre los procesos
CTL_FS	Sistemas de archivos
CTL_DEBUG	Depuración
CTL_DEV	Dispositivos
CTL_MAXID	Último identificador

Cada categoría posee luego sus propias variables que permiten acceder a los diferentes valores. Sólo se detallan **CTL_KERN** y **CTL_VM**:

• **CTL_KERN:**

constante	significado
KERN_OSTYPE	Nombre del sistema (cadena), aquí Linux
KERN_OSRELEASE	Versión del núcleo (cadena)
KERN_OSREV	Número de versión (int)
KERN_VERSION	Fecha de generación del núcleo (cadena)
KERN_SECUREMASK	Máscara de derechos máximos (estructura)
KERN_PROF	Información para la depuración (tabla)
KERN_NODENAME	Nombre de la máquina
KERN_DOMAINNAME	Nombre del ámbito
KERN_MNINODE	Número de i-nodos asignados
KERN_MAXINODE	Número de i-nodos máximo
KERN_NRFILE	Número de descriptores de archivo utilizados

KERN_MAXFILE	Número máximo de archivos abiertos
KERN_MAXID	
KERN_SECURELVL	Nivel de seguridad del sistema
KERN_PANIC	timeout para un panic (int)
KERN_REALROOTDEV	Dispositivo real a montar como raíz tras initrd
KERN_NFSNAME	Nombre de la raíz NFS
KERN_NFSRADDRS	Dirección de la máquina con la raíz NFS

• CTL_VM:

<i>constante</i>	<i>significado</i>
VM_SWAPCTL	Configura la gestión del <i>swap</i> (estructura)
VM_KSWAPD	Configura el sistema de paginación (estructura)
VM_FREEMPG	Fija el umbral de páginas libres (estructura)
VM_BUF_FLUSH	Controla la gestión del <i>búfer caché</i> (estructura)
VM_MAXID	

Veamos un ejemplo de uso que da el nombre de ámbito de la máquina:

```
#include <linux/unistd.h>
#include <linux/types.h>
#include <linux/sysctl.h>
#include <stdio.h>
#include <stdlib.h>

#define DOMAINSIZ 50

_syscall1 (int, _sysctl, struct __sysctl_args *, args);

int sysctl (int *name, int nlen, void *oldval,
            size_t * oldlenp, void *newval, size_t newlen)
{
    struct __sysctl_args args =
        {name, nlen, oldval, oldlenp, newval, newlen};

    return _sysctl (&args);
}

void main ()
{
    int name[] = {CTL_KERN, KERN_DOMAINNAME};
    char mydomain[DOMAINSIZ];
    int mydomainlength = DOMAINSIZ;

    if (sysctl (name, sizeof (name), mydomain, &mydomainlength,
                0, 0))
        perror ("sysctl");
}
```

```
else
    printf ("domainname: %s \n", mydomain);
}
```

3 Implementación

La implementación de las llamadas al sistema se encuentra en los archivos siguientes:

- *kernel/sys.c* para *sethostname*, *gethostname*, *setdomainname*, *reboot* y *uname*;
- *kernel/printk.c* para *syslog*;
- *kernel/time.c* y *arch/i386/kernel/time.c* para *adjtimex*, *gettimeofday*, *settimeofday*, *time* y *stime*;
- *kernel/sysctl.c* para *sysctl*.

3.1 *sethostname*, *gethostname*, *setdomainname* y *uname*

Estas llamadas al sistema manipulan la variable `system_utsname`. La estructura `new_utsname` define el tipo de esta variable. El tratamiento de estas primitivas consiste en devolver o modificar uno o más de los campos de `system_utsname`.

Por ejemplo, el código de la llamada al sistema *gethostname* es el siguiente:

```
asmlinkage int sys_gethostname(char *name, int len)
{
    int i;

    if (len < 0)
        return -EINVAL;
    i = verify_area(VERIFY_WRITE, name, len);
    if (i)
        return i;
    i = 1+strlen(system_utsname.nodename);
    if (i > len)
        i = len;
    memcpy_toofs(name, system_utsname.nodename, i);

    return 0;
}
```

Tras haber efectuado las verificaciones de uso respecto a la validez de la zona de memoria apuntada (`verify_area`), esta llamada al sistema se reduce a efectuar la copia del campo `system_utsname.nodename` en la zona de memoria apuntada por `name`. En el caso de `sethostname`, se realiza la operación inversa: la copia del contenido de la memoria apuntada por `name` en `system_utsname.nodename`:

```
memcpy_fromfs(system_utsname.nodename, name, len);
```

3.2 *reboot*

La llamada al sistema *reboot* permite, como hemos visto anteriormente, cambiar el estado del núcleo. En realidad, puede descomponerse en dos: parada completa de la máquina o arranque.

Si se trata únicamente de una parada de la máquina (por ejemplo, utilizando el mandato *halt*), la operación se efectúa en dos pasos:

- lanzamiento de `sys_kill(-1, SIGKILL)` que tiene por efecto parar todos los procesos;
- lanzamiento de `do_exit(0)` que destruye todos los recursos asignados, y el núcleo vuelve a quien le ha llamado, lo que provoca la parada de la máquina. En realidad, se trata de un bucle infinito porque no es posible lógicamente realizar la parada material de una máquina.

Si se trata de reaniciar la máquina, *reboot* va seguida por la llamada a la función `hard_reset_now` que realiza de una manera efectiva el arranque de la máquina. Esta función es dependiente de la arquitectura sobre la cual se encuentra. Se sitúa en el archivo `arch/i386/kernel/process.c` en el caso de la arquitectura x86.

3.3 *syslog*

La llamada al sistema *syslog* se implementa en el módulo `kernel/printk.c`. La memoria intermedia que contiene los mensajes al sistema es una tabla de caracteres `log_buf` de tamaño `LOG_BUF_LEN`. Dos variables de tipo `unsigned long` permiten gestionar esta tabla de datos:

- `log_start`: índice de inicio de los mensajes en la tabla;
- `logged_chars`: número de caracteres almacenados.

La llamada *syslog* consiste seguidamente en manipular esta memoria intermedia, duplicándola en la memoria pasada como parámetro, o bien modificando los índices.

3.4 *gettimeofday, settimeofday, time y stime*

Estas cuatro llamadas al sistema tienen por objetivo recuperar o fijar las fechas de la máquina. Todas utilizan la variable global *xtime* de tipo *timeval*.

3.4.1 *time*

Esta llamada utiliza la macroinstrucción *CURRENT_TIME*, que devuelve el campo *tv_sec* de la variable global *xtime*. El campo correspondiente a los microsegundos es siempre nulo.

3.4.2 *stime*

La llamada al sistema *stime* se limita a asignar el valor pasado como parámetro al campo *tv_sec* de *xtime*. Hay que destacar que esta llamada es una operación ininterrumpible, gracias al uso de las funciones *cli* y *sti*.

3.4.3 *gettimeofday*

En un principio, esta llamada recupera los datos respecto a la fecha y hora si el usuario lo desea. Esta operación se realiza mediante la función *do_gettimeofday* situada en *arch/i386/kernel/time.c*.

Esta operación es necesaria para poder tener una fecha extremadamente precisa, por ejemplo para los microsegundos. Linux proporciona dos algoritmos para efectuar este cálculo: *do_slow_gettimeofday* y *do_fast_gettimeofday*. El primero se usa en estas llamadas. El segundo está reservado a la inicialización del sistema operativo.

Tras haber efectuado el cálculo necesario de microsegundos, la llamada al sistema se limita a actualizar la estructura *timeval* pasada como parámetro. Toda esta operación se hace también de manera ininterrumpible.

Finalmente, la llamada al sistema actualiza la variable global *sys_tz* de tipo *time-zone*.

3.4.4 *settimeofday*

Esta operación es mucho más simple que la llamada anterior, y se hace en tres etapas. Los parámetros se copian en las variables locales *new_tv* y *new_tz*.

Una vez actualizada la variable *sys_tz*, la hora de la máquina se recalcula en función de los nuevos datos del huso horario obtenidos mediante la función *warp_clock*.

Finalmente, se actualiza la hora de la máquina en función de la nueva hora pasada como parámetro mediante la función *do_settimeofday*. Ésta actualiza la variable *xtime*.

3.5 *sysctl*

Aunque el código de esta llamada se encuentra en el archivo *kernel/sysctl.c*, su funcionamiento está íntimamente vinculado al sistema virtual de archivos *proc* que se detalla en el capítulo 6, sección 6.6.

Al inicializarse la máquina, la función *sysctl_init* crea el árbol */proc/sys* si el sistema de archivos */proc* está incluido en el núcleo. Esta operación se efectúa simplemente por la llamada a *register_proc_table*. Ésta genera los subdirectorios *kernel*, *vm* y *net*. Asimismo, se llama a la función *unregister_proc_table* al desmontar el sistema de archivos.

La creación de las tablas necesarias para la gestión de los datos se efectúa por la función *register_sysctl_table*. La función inversa se realiza efectuando una llamada a la función *unregister_sysctl_table*. Estas llamadas permiten añadir un grupo de archivos al sistema de archivos.

La llamada al sistema *sysctl* se implementa realmente en la función *do_sysctl*. En un principio, se efectúa una verificación de los datos pasados como parámetro. Luego, mediante llamadas sucesivas a la función *parse_table*, se lanzan las órdenes de búsqueda de los datos o de modificaciones.

Esta función busca la entrada en el sistema de archivos que corresponde a los datos que se desea consultar o modificar. Una vez localizada la entrada, la llamada a la función *do_sysctl_strategy* efectúa las operaciones deseadas. Esta operación se limita a modificar o bien a copiar el contenido deseado, si los derechos lo permiten. Sin embargo, en el caso del archivo *securelevel*, se llama a la función *do_securelevel_strategy*. En efecto, las modificaciones de la variable *securelevel* se

someten a diversas restricciones, y es necesario utilizar una función particular que efectúa los controles.

Este módulo también se encarga de la gestión de entradas/salidas que pueden efectuarse directamente sobre los archivos situados en el árbol `/proc/sys`. Estas operaciones se realizan por las funciones `proc_readsys` y `proc_writesys`. Los derechos de acceso se verifican por la función `proc_sys_permission`. De hecho, en una entrada/salida sobre uno de los archivos del sistema de archivos, se efectúa una llamada a la función `do_rw_proc`.

Los archivos se gestionan en forma de cadenas de caracteres, de tablas de enteros o de estructuras. Las funciones `proc_doststring`, `proc_dointvec` y `proc_dointvec_minmax` se encargan de la gestión de los datos, desde el sistema de archivos. Sin embargo, los diferentes tipos de datos se gestionan ante todo por las funciones genéricas `sysctl_string` y `sysctl_intvec`, así como por las funciones más específicas `do_string`, `do_int` y `do_struct`.

Aunque `sysctl` esté vinculada al sistema de archivos `proc`, esta llamada funciona sin que el sistema de archivos esté montado o compilado. Sin embargo, la facilidad de manipulación de los datos que ofrece el sistema virtual de archivos simplifica en gran medida el trabajo del desarrollador o del usuario.

3.6 *adjtimex*

La llamada al sistema *adjtime* se implementa en el archivo `kernel/time.c`. No detallaremos aquí el algoritmo que consiste en calcular la nueva hora a partir de la hora contenida en la variable global `xtime` en función del contenido de la variable de tipo `timex` pasada como parámetro porque este punto sobrepasa ampliamente el marco de esta obra.

Sin embargo, el principio es poner en una variable `time_adjust` la duración según la cual hay que reajustar el reloj de la máquina. Este valor será tenido en cuenta por la función `update_wall_time_one_tick` del archivo `kernel/sched.c`.

Esta función actualizará el campo `tv_usec` de la variable `xtime`. La regulación del reloj se efectúa muy afinadamente de manera sucesiva. Esta llamada al sistema sólo la usa el demonio *xntpd*.

Apéndice A

Fases de una compilación C

Para presentar de manera rápida las distintas partes de la compilación de un programa en C, realizaremos todo el proceso de la compilación con un programa de ejemplo (cuyo uso es muy limitado):

```
#define MAXIMUM_VOLUME 10

void main (int argc, char **argv)
{
    int volume_total = 0;
    volume_total = MAXIMUM_VOLUME * 2;
}
```

Este programa efectúa una simple multiplicación por dos de una constante.

1 Preprocesador

El paso del preprocesador puede efectuarse de dos maneras:

- `cpp ejemplo.c`
- `gcc -E ejemplo.c`

El resultado es el mismo, y por una buena razón: *gcc* se limita a lanzar el programa *cpp*. El resultado del archivo tratado se envía a la salida estándar.

```
gandalf# cpp ejemplo.c

# 1 "ejemplo.c"

void main(int argc, char **argv)
{
    int volumen_total = 0;
    volumen_total = 10 * 2;
}
```

cpp posee un cierto número de opciones interesantes:

- **-Wall**: indica todos los errores o avisos;
- **-IDIRECTORIO**: indica un directorio donde ir a buscar los archivos de cabecera;
- **-DMACRO**: define una macroinstrucción;
- **-DMACRO=valor**: define una macroinstrucción y su valor;
- **-M**: permite generar las dependencias asociadas a los archivos.

2 Compilador

La compilación es el instante crítico porque en este momento es cuando se efectúa la verificación del código. El código se transforma en código ensamblador. En esta etapa del proceso de compilación se realizan ciertas optimizaciones. Para pasar del código C al código ensamblador, se pueden usar dos mandatos:

- `gcc -S ejemplo.c`
- `cc1 ejemplo.c`

El código ensamblador generado se encontrará en el archivo *ejemplo.s*:

```
.file "exemplo.cpp"
.version      "01.01"
gcc2_compiled.:
.text
.align 16
.globl main
.type main,@function
```

```
main:
pushl %ebp
movl %esp,%ebp
subl $4,%esp
movl $0,-4(%ebp)
movl $20,-4(%ebp)
.L1:
movl %ebp,%esp
popl %ebp
ret
.Lfel:
.size main,.Lfel-main
.ident      "GCC: (GNU) 2.7.2"
```

Las opciones usadas frecuentemente para esta compilación son las siguientes:

- `-O`: nivel de optimización: de 0 (ninguna optimización) a 3 (optimización máxima);
- `-g`: utilizada para la depuración: adición de la tabla de símbolos;
- `-Wall`: indica todos los errores o avisos.

El código puede recompilarse con las diversas opciones de optimización y de depuración para observar las diferencias sobre el código ensamblador obtenido.

3 Ensamblador

La etapa de ensamblado consiste en transformar el código ensamblador en un código binario. Esta operación se realiza por el programa *as*. La lista de las diferentes opciones puede obtenerse utilizando la opción `-help`.

La ejecución se hace de manera simple:

```
gandalf# as -o ejemplo.o ejemplo.s
```

4 Enlazador

La última etapa del proceso de compilación es el enlazado. Esta etapa reúne todos los archivos objeto para generar un archivo ejecutable. La operación se efectúa con el mandato *ld*. Este mandato se efectúa raramente «a mano» porque necesita un buen número de parámetros:

```
gandalf# ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.1 -s/usr/lib/crt1.o
/usr/lib/crti.o /usr/lib/crtbegin.o -L/usr/lib/gcc-lib/i486-linux/2.7.2
-L/usr/i486-linux/lib exemple.o -lgcc -lc -lgcc /usr/lib/crtend.o
/usr/lib/crtn.o

gandalf# ls -al a.out
-rwxr-xr-x    1 dumasusers          2356 Apr 27 13:19 a.out

gandalf# file a.out
a.out: ELF 32-bit LSB executable i386 (386 anad up) Version 1
```

El enlazador combina el archivo objeto *ejemplo.o*, una cabecera (*crt1.o*, *crti.o*, *crtbegin.o*), un fin (*ctrend.o*, *crtn.o*) y las bibliotecas predeterminadas (c y gcc) para crear un archivo ejecutable.

La opción `-m format` corresponde al formato del binario que el desarrollador quiera generar. Para conocer la lista de formatos que soporta el enlazador, hay que utilizar la opción `-V`:

```
gandalf# ld -V
ld version cygnus-2.6 (with BFD 2.6.0.12)
Supported emulations:
  elf_i386
  i386linux
  i386coff
  m68kelf
  sun4
  elf32_sparc
  alpha
```

Apéndice B

Utilización de *gdb*

Hemos creado un programa que tiene por objetivo mostrar dos ocurrencias de la cadena pasada como parámetro para ilustrar el funcionamiento de *gdb*. La ejecución de este programa genera un error:

```
gandalf# occurrences EstoEsUnProgramaQueNoFunciona
Segmentation fault
```

Es necesario, pues, buscar el punto donde se presenta el problema. En un primer momento, basta con lanzar *gdb*:

```
gandalf# gdb occurrences
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i486-unknown-linux), Copyright 1996 Free Software Foundation, Inc...
(gdb)
```

El intérprete de mandatos de *gdb* es relativamente potente. Permite completar y gestionar un historial. Es posible escribir en cualquier momento la palabra clave *help* para obtener ayuda.

El primer mandato que hay que lanzar es *run*, que permitirá ejecutar el programa. Pero esto no tiene gran interés porque está claro que el programa no funciona. En principio, es necesario fijar el argumento que se pasará al programa mediante las palabras clave *set args*. Seguidamente, pondremos un punto de parada en la función *main*. Para ello, lo más simple es escribir *break main*.

Poner un punto de parada en la función `main` significa que en la ejecución, el depurador se detendrá en la función `main`:

```
(gdb) set args EstoEsUnProgramaQueNoFunciona
(gdb) break main
Breakpoint 1 at 0x8048596: file coredumped.c, line 12.
(gdb) run
Starting program: /home/dumas/10...Livre/1...
Eyrolles/Outils/sources/ocurrences EstoEsUnProgramaQueNoFunciona

Breakpoint 1, main (argc=2, argv=0xbffff828) at coredumped.c:12
12         char *new_str=NULL;
(gdb)
```

Como estaba previsto, *gdb* ha parado la ejecución del programa en el punto de parada definido. Se muestra la línea (aquí 12), correspondiente a la línea que se va a ejecutar. En todo momento es posible ver el código del programa mediante el mandato `list`:

```
(gdb) list
static char *doublestr(const char *str);
static void usage(const char *prg);
9
10 void main(int argc, char **argv)
11 {
12     char *new_str=NULL;
13
14     if(argc<2)
15         uso(argv[0]);
16
(gdb)
```

Tras esta presentación general del depurador, salimos en busca del problema. Para proseguir la ejecución, son posibles tres mandatos:

- `step`: ejecución paso a paso. El programa se ejecuta línea por línea, y se entra en todas las funciones encontradas;
- `next`: lo mismo, pero sin entrar en las funciones;
- `continue`: ejecuta el programa hasta el fin o hasta el próximo punto de parada.

Efectuando varias veces la operación `step`, el cursor pasa a encontrarse en la función `doublestr`:

```
(gdb) step
14      if (argc<2)
(gdb) s
17      new_str = doublestr(argv[1]);
(gdb) s
doublestr (str=0xbffff947 "EstoEsUnProgramaQueNoFunciona")
    at coredumped.c:27
27      char *ret_str = NULL;
(gdb) s
28      char *digit_str = NULL;
(gdb) s
32      if (str == NULL)
(gdb) s
35      if ((digit_str = malloc(sizeof(char)))=NULL)
(gdb) s
38      new_length = strlen(str)*2 + strlen(SEPARATOR);
(gdb) s
39      if ((ret_str = malloc(sizeof(char)*new_length))=NULL)
(gdb) s
45      sprintf(digit_str, "(%d)", new_length);
(gdb)
```

Es interesante subrayar que *gdb* indica los parámetros pasados a la función.

El lector atento habrá detectado probablemente el error: en la línea 35, se asigna una zona de memoria del tamaño de un carácter, pero esta zona de memoria se asigna a una cadena de caracteres constituida por un número de caracteres claramente superior. Para corregir este código, basta con efectuar la modificación en el código fuente y reiniciar la compilación.

También es posible conocer el valor de una variable, el contenido de una estructura y el contenido de una cadena a partir de su puntero. Estas operaciones pueden efectuarse de manera muy simple con el mandato *print*:

```
(gdb) print new_length
$1 = 68
(gdb) print str
$2 = 0xbffff947 "EstoEsUnProgramaQueNoFunciona"
(gdb) print *str
$3 = 67 'E'
(gdb)
```

Finalmente, la última operación respecto al depurador que presentamos consiste en encontrar en la lista la ubicación de las llamadas a las funciones. Esta operación se realiza con el mandato *where*, o *backtrace*:

```
(gdb) where
#0 doublestr (str=0xbffff947 "EstoEsUnProgramaQueNoFunciona")
   at coredumped.c:46
#1 0x80485bf in main (argc=2, argv=0xbffff828) at coredumped.c:17
#2 0x8048544 in __crt_dummy__()
(gdb) backtrace
#0 doublestr (str=0xbffff947 "EstoEsUnProgramaQueNoFunciona")
   at coredumped.c:46
#1 0x80485bf in main (argc=2, argv=0xbffff828) at coredumped.c:17
#2 0x8048544 in __crt_dummy__()
```

Este mandato da la lista de las diferentes llamadas que se han efectuado durante la depuración, dando para cada una el nombre del archivo donde se encuentra la función, la línea y la lista de los parámetros pasados.

Apéndice C

Utilización de *make*

1 Funcionamiento de *make*

Al ser ejecutado, *make* utiliza el archivo *makefile* o *Makefile* situado en el directorio actual, analiza su contenido y lanza los mandatos indicados.

Esta herramienta es muy práctica porque permite, entre otras cosas, recompilar sólo lo necesario.

2 Escritura de un archivo *makefile*

El ejemplo que ilustra este apartado se utiliza también en el apéndice D. Este ejemplo usa varios archivos:

- *complejo.c*: módulo que implementa los cálculos sobre números complejos;
- *complejo.h*: interfaz de prototipos del módulo *complejo.c*;
- *dibujo.c*: módulo de representación gráfica de los números complejos;
- *dibujo.h*: interfaz de los prototipos del módulo *dibujo.c*;
- *main.c*: archivo principal.

El archivo *Makefile* asociado es entonces el siguiente:

```
# Archivo makefile para números complejos
CC=gcc
RM=/bin/rm

# Opciones de compilación
```

```
CFLAGS=-g

# Archivos
SRC=complejo.c dibujo.c main.c
OBJ=$(SRC:.c=.o)

PROGRAMA=complejo
LIB=-lm

# Reglas de generación
$(PROGRAMA) : $(OBJ)
$(CC) $(CFLAGS) $(OBJ) -o $(PROGRAMA) $(LIB)

# ¡Y se hizo la propiedad!
clean:
$(RM) -f $(OBJ) $(PROGRAMA)

# Dependencias
complejo.o : complejo.c complejo.h
dibujo.o : dibujo.c dibujo.h complejo.h
main.c : main.c dibujo.h complejo.h
```

La primera parte de este archivo está constituida por declaraciones diversas que permiten una cierta flexibilidad de uso en el caso en que se quiera, por ejemplo, cambiar de compilador. Seguidamente, se definen las opciones que deberán utilizarse en la compilación. Aquí se trata de la opción `-g` para poder depurar eventuales errores.

Viene a continuación la enumeración de los distintos archivos que deben compilarse. La variable `OBJ` está formada por una regla que evita tener que enumerar todos los archivos objeto sabiendo que se trata de los mismos nombres que para los archivos fuente, aparte de la extensión.

La última parte es la regla de creación del ejecutable. Esta regla se ejecuta únicamente si todos los archivos objeto han sido generados. Al fin del archivo se encuentran las dependencias que permitirán a *make* compilar sólo los archivos modificados.

Por ejemplo, supongamos que se hayan generado todos los archivos objeto, pero desde la última compilación un desarrollador ha modificado el archivo *dibujo.h*. En la próxima compilación, sólo se recompilarán ciertos archivos:

```
gandalf# make
gcc -g -c dibujo.c -o dibujo.o
gcc -g -c main.c -o main.o
gcc -g complejo.o dibujo.o main.o -o complejo -lm
```

Gracias a la definición de dependencias, el archivo *complejo.c* no se recompila. Para conocer las dependencias de un programa, basta con lanzar el mandato `gcc -MM dibujo.c`. Se obtiene así la lista de archivos de cabecera utilizados por este módulo.

Apéndice D

Gestión de las bibliotecas

1 Herramientas de creación y de manipulación

Puede ocurrir que se necesite buscar una biblioteca: «¿Cuál es la biblioteca que define la función `sin`?». Todo desarrollador se ha hecho preguntas como ésta. Una solución es utilizar el mandato `nm`. Este programa proporciona la lista de símbolos referenciados en una biblioteca. Esto funciona tanto si la biblioteca es estática como si es dinámica.

Por ejemplo, verifiquemos que `sin` se encuentra en la biblioteca matemática:

```
gandalf# nm -P /lib/libm.so.5 | grep sin
                U __isinf
                U __isinfl
00000000000016d0 T asin
000000000000174c T asinh
0000000000001800 T asinhl
00000000000018b4 T asinl
0000000000005354 T sin
0000000000005378 T sinh
00000000000053d8 T sinhl
0000000000005438 T sinl
```

El resultado se organiza alrededor de tres columnas:

- La dirección en la que se encuentra la función en la biblioteca, si está definida. Si es sólo una llamada o una definición de una función externa, no se inserta ninguna dirección. La resolución se efectúa entonces dinámicamente en la carga.

- Una letra que indica el tipo de la definición del símbolo:
 - U: símbolo no definido.
 - W: símbolo definido pero que puede ser «sobrecargado». Esto permite definir funciones propias para reemplazar las de la biblioteca.
 - T: símbolo definido normalmente.
- El nombre del símbolo.

2 Bibliotecas estáticas

El modo de creación de una biblioteca estática es el mismo tanto si es para ELF como para a.out. Es necesario ante todo generar los archivos objeto. Luego, los programas *ar* y *ranlib* harán el resto. Hay que destacar que en ciertos sistemas (como Solaris) el mandato *ranlib* no existe. Bajo Linux, este mandato equivale a *as -s*.

Veamos un ejemplo de creación de la biblioteca de manipulación de números complejos:

```
gandalf# gcc -c complejo.c -o complejo.o
gandalf# ar cru libcomplejo.a complejo.o
gandalf# ranlib libcomplejo.a
```

ar posee un número de parámetros impresionante. La página del manual detalla cada una de las opciones. Las opciones utilizadas en este ejemplo permiten crear la biblioteca que tiene por nombre *libcomplejo.a* a partir de los archivos objeto especificados.

El mandato *ranlib* que sigue a la generación de la biblioteca permite generar el índice de la biblioteca. Para obtener este índice (que agrupa las funciones y las variables exportadas) basta con hacer:

```
gandalf# nm -s libcomplejo.a

Archive index:
crear_complejo in complejo.o
destruir_complejo in complejo.o
suma_complejo in complejo.o
multiplicar_complejo complejo.o
real_complejo complejo.o
imaginario_complejo complejo.o
algebraico_complejo complejo.o
...
```

Sin embargo, el prototipo no se ha dado, por lo que la consulta de cabecera es necesaria para poder desarrollar.

3 Bibliotecas dinámicas a.out

La generación de bibliotecas compartidas de tipo a.out ha sido siempre una operación difícil. Debido al abandono de este formato, no daremos la descripción de esta operación, pero toda la operativa está descrita en [Engel Youngdale 1992].

4 Bibliotecas dinámicas ELF

El gran progreso aportado por las bibliotecas ELF ha sido la generación de bibliotecas compartidas dinámicas. Para crear una biblioteca dinámica, es necesario compilar los diferentes módulos añadiendo la opción `-fPIC`. Seguidamente, la generación de la biblioteca dinámica se efectúa en forma de un enlazado.

Veamos la generación de la biblioteca de números complejos:

```
gandalf# gcc -c -fPIC complejo.c -o complejo.o
gandalf# gcc -shared -Wl, -soname, libcomplejo.so.1 -o libcomplejo.so.1.0
complejo.o
```

Cada una de las opciones es necesaria y cumple un papel particular:

- `-shared`: especifica al enlazador que debe crear una biblioteca dinámica;
- `-Wl`: permite pasar parámetros particulares al enlazador;
- `-soname`: permite especificar el nombre «propio» de la biblioteca.

Las bibliotecas dinámicas tienen en realidad dos nombres:

- el nombre propio: se guarda en la biblioteca y es buscado por el cargador dinámico en la ejecución de un programa.
- el nombre físico: su utilidad se limita a los caminos de acceso.

Esta distinción puede parecer superflua, pero permite la coexistencia de varias versiones diferentes de la misma biblioteca sobre el mismo sistema.

El nombre de una biblioteca dinámica debe respetar una convención:

`libnombre.so.mayor.menor`

Puede añadirse eventualmente un número suplementario tras el número menor.

En el ejemplo anterior, el nombre físico de la biblioteca es `libcomplejo.so.1.0`, y el nombre propio es `libcomplejo.so`. Si esta biblioteca se añade a un directorio estándar (por ejemplo en `/usr/lib`), el programa `ldconfig` crea un enlace simbólico `libcomplejo.so.1 → libcomplejo.so.1.0` para que la imagen apropiada se encuentre en la ejecución. Para terminar, hay que crear un enlace `libcomplejo.so` hacia la nueva versión.

Cuando el desarrollador corrige errores en la biblioteca o bien añade nuevas funciones (para toda modificación que no afecte a la ejecución de los programas ya existentes), es necesario reconstruir la biblioteca. Basta con incrementar el número menor. Cuando efectúa modificaciones que alteran el funcionamiento de los programas existentes, debe incrementar el número mayor de la biblioteca `libcomplejo.so.1.0`, y darle como nombre propio `libcomplejo.so.2`. Finalmente, debe recrear el enlace `libcomplejo.so` hacia la nueva versión.

5 La carga dinámica de bibliotecas

Una de las ventajas que procura el formato ELF es poder especificar la carga de una biblioteca dinámica, y su descarga cuando la operación haya terminado. Numerosos programas como *perl* o *Java* lo usan activamente.

Estas funcionalidades necesitan el uso de la biblioteca `libdl`, y el archivo de cabecera `<dlfcn.h>`. Todo el principio de la carga dinámica se detalla en [Lu 1995].

Veamos un ejemplo que ilustra esta característica:

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

#ifdef PI
#define PI 3.14159265358979323846
#endif

typedef double (*fun_t) (double);
```

```
void main (int argc, char **argv)
{
    void                *ptr = NULL;
    char                *nombib = "libm.so";
    char                *funcion = "cos";
    fun_t               fun = NULL;
    double              resultado = 0;

    /* Carga de la biblioteca dinámica */
    if ((ptr = dlopen (nombib, RTLD_LAZY)) == NULL) {
        fprintf(stderr, "dlopen(%s) -> %s \n", nombib, dlerror());
        exit (1);
    }
    /* Recuperación de la dirección de la función */
    if ((fun = (fun_t) dlsym (ptr, funcion)) == NULL) {
        fprintf(stderr, "dlsym(%s) -> %s \n", funcion, dlerror());
        exit (1);
    }
    printf ("Llamada de %s en %s \n", funcion, nombib);
    resultado = (*fun) ((double) PI / 4);
    dlclose (ptr);
    printf ("Resultado: %f \n", resultado);
}
```

La compilación de este programa cuyo objetivo es efectuar la operación $\cos \pi/4$ da:

```
gandalf# gcc -c -g -Wall dltest.c -o dltest.o
gandalf# gcc -g dltest.o -o dltest -ldl
gandalf# ldd dltest
        libdl.so.1 => /lib/libdl.so.1.7.14
        libc.so.5 => /lib/libc.so.5.3.9
gandalf# ./dltest
Llamada a cos en libm.so
Resultado: 0.707107
```

El programa no está vinculado a la biblioteca matemática, y sin embargo, se ha llamado a la función *cos* de la biblioteca matemática.

Los prototipos de las funciones son los siguientes:

```
#include <dlfcn.h>

void *dlopen (const char *filename, int flag);

const char *dlerror (void);

void *dlsym (void *handle, char *symbol);

int dlclose (void *handle);
```

La función `dlopen` tiene como segundo argumento una opción que especifica el modo de resolución de los símbolos. Esta opción propone las opciones siguientes: en la carga, se realiza sólo la resolución de símbolos no definidos si es posible (`RTLD_LAZY`), o bien todos los símbolos se resuelven antes del fin de la llamada (`RTLD_NOW`), y si un símbolo no se puede resolver, la llamada falla.

La función `dlsym` permite recuperar un *handler* a la función deseada. Finalmente, la función `dlclose` permite descargar la biblioteca dinámica, si ninguna otra aplicación la usa. Esto significa que tras un `dlclose`, toda llamada a una función de esta biblioteca por parte del programa se saldrá con una parada del programa.

Bibliografía

[Aho *et al.* 1991] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilateurs: principes, techniques et outils*. InterEditions, 1991.

[AT&T1989] AT&T. *Unix SYSTEM V - MANUEL DE Référence du gestionnaire système*. Masson - AT&T, 1989.

[Bach 1993] Maurice J. Bach. *Conception du système Unix*. Masson, Prentice Hall, 4ième edition, 1993.

[Barlow 1995] Daniel Barlow. *ELF HowTo*. Technical report, Linux Documentation Project, Sep 1995. Versión francesa de René Cougnenc.

[Barlow 1996] Daniel Barlow. *GCC HowTo*. Technical report, Linux Documentation Project, Feb 1996. Versión francesa de Eric Dumas (dumas@freenix.fr).

[Beck *et al.* 1996] M. Beck, H. Böhme, M. Dziadza, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison-Wesley, 1996.

[Card *et al.* 1994] Rémy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *Proceedings of the Linux Dutch Symposium*, págs. 34-51, Dec 1994.

[Dijkstra 1965] E. W. Dijkstra. *Cooperating Sequential Processes*. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.

[Dumas 1996] Eric Dumas. *Guide du ROOTard pour Linux*. Technical report, Septembre 1996. Con las contribuciones de Rémy Card, René Cougnenc, Julien Simon, Pierre Ficheux y Nat Makarévitch.

[Ellis and Stroustrup 1990] Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Engel and Youngdale 1992] David Engel and Eric Youngdale. *Using DLL Tools With Linux*. Technical report, Dec 1992. Incluido en el archivo `ftp://tsx-11.mit.edu/pub/linux/packages/GCC/src/tools-2.17.tar.gz`.

- [Froidevaux *et al.* 1993] Christine Froidevaux, Marie-Claude Gaudel, and Michèle Soria. *Types de données et algorithmes*. Ediscience International, 1993.
- [Gallmeister 1995] Bill O. Gallmeister. *Posix 4: programming for the real world*. O'Reilly & Associates, 1995.
- [Goodheart and Cox 1994] Berny Goodheart and James Cox. *The Magic Garden Explained: The Internals of UNIX System V Release 4*. Prentice Hall, 1994.
- [Kernighan and Ritchie 1992] B. W. Kernighan and D.M. Ritchie. *Le Langage C*. Masson, Prentice Hall, 2^e edición, 1992.
- [Kirch 1995] Olaf Kirch. *Administration réseau sous Linux*. O'Reilly & Associates, 1995. Versión francesa realizada por René Cougnenc.
- [Kleiman 1986] S. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the Summer USENIX Conference*, pages 260-269, June 1986.
- [Knowlton 1965] K. C. Knowlton. A fast storage allocator. *Communications of the ACM*, 8(10):623-625, October 1965.
- [Lewine 1991] Donald Lewine. *POSIX Programmer's Guide*. O'Reilly & Associates, 1991.
- [Lu 1995] Hongjiu Lu. *Elf: From the Programmer's Perspective*. Technical report, NYNEX Science and Technology, Inc., May 1995.
- [McKusick *et al.* 1984] Marshall K. McKusick, William Joy, Samuel J. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181-197, August 1984.
- [McKusick *et al.* 1996] Marshall K. McKusick, Keith Bostic, Michael J. Karels, and John S. Quaterman. *The Design and Implementation of the 4.4BSD UNIX Operating System*. Addison Wesley, 1996.
- [Mills 1992] David L. Mills. *RFC1305*. Technical report, University of Delaware, Mars 1992. *Network Time Protocol (Version 3) - Specification, Implementation and Analysis*.
- [Moreno 1995] Jean-Michel Moreno. *UNIX Administration*. Ediscience International, 1995.

- [Silberschatz and Calvin 1994] Abraham Silberschatz and Peter B. Calvin. *Operating System Concepts*. Addison-Wesley, 4th edition, 1994.
- [Stallman and McGrath 1995] Richard M. Stallman and Roland McGrath. *Gnu make*. Technical report, Documentation GNU, Abril 1995.
- [Stallman and Support 1994] Richard M. Stallman and Cygnus Support. *Debugging with GDB*. Technical report, Documentation GNU, Ene 1994.
- [Satallman 1995] Richard M. Stallman. *Using and Porting GNU CC*. Technical report, Documentation GNU, Nov 1995.
- [Storner 1996] Henrik Storner. *Kernel Mini-Howto*. Technical report, Linux Documentation Project, Jun 1996.
- [Tanenbaum 1987] Andrew Tanenbaum. *Operating Systems: Design and Implementation*. Prentice Hall, 1987.
- [TIS 1993] TIS. *Executable and Linkable Format (ELF)*. Technical report, 1993. <ftp://ftp.ibp.fr/pub/linux/tsx-11/packages/GCC/ELF.doctar.gz>.
- [Vahalia 1996] Uresh Vahalia. *Unix Internals: The New Frontiers*. Prentice Hall, 1996.
- [Welsh and Kaufman 1995] Matt Welsh and Lar Kaufman. *Le Système Linux*. O'Reilly & Associates, 1ª edition, 1995. Versión francesa realizada por René Cougnenc.