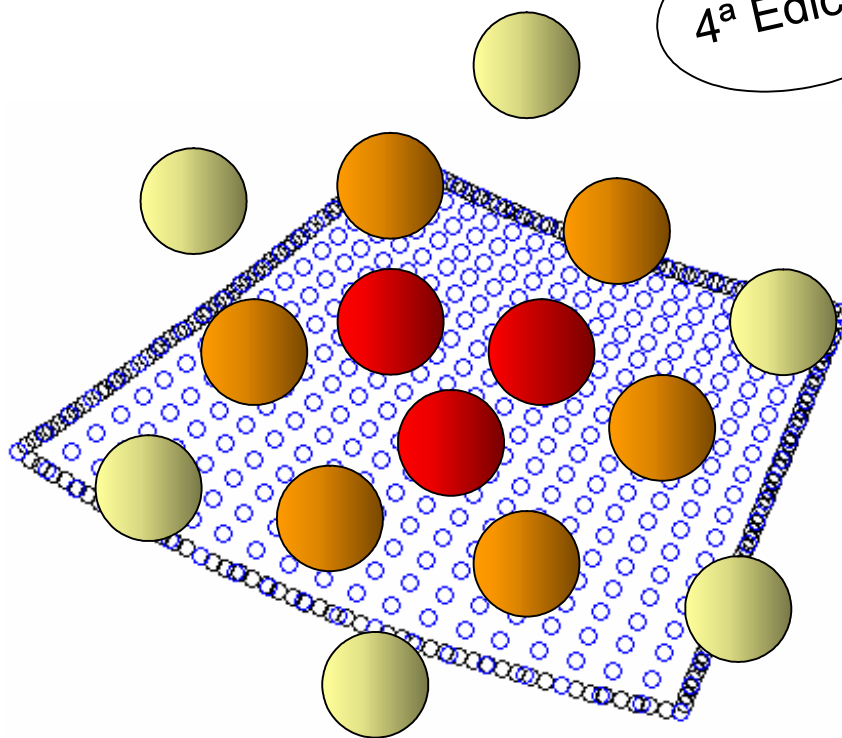


SISTEMAS OPERATIVOS II

APUNTES

4ª Edición



Jose Manuel Díaz - Salvador Ros



U.N.E.D
Ingeniería Técnica en Informática de Sistemas

TEMA 0

UN MOMENTO DE REFLEXIÓN

Creo firmemente que cada vez nuestra tipo de sociedad nos lleva a olvidar a nuestros semejantes mas desfavorecidos o con problemas. La universidad debe ser un foro de conocimientos, es verdad, pero también de humanidad. Analizando cómo en un texto sobre sistemas operativos se podría aportar algo en este sentido, hace un par de años, se publicaron estos apuntes de forma gratuita, hoy la segunda edición está a vuestra disposición y en ella hemos colaborado más profesores.

Nos gustaría que antes de empezar vuestro estudio leyerais estas líneas, tranquilos, sin prisas porque creo que merecen un MOMENTO DE REFLEXIÓN.

Todos sabéis el coste real de venta de un libro (36 a 44 euros), y lo que va a suponeros editar éste desde formato electrónico, (prácticamente el papel o las fotocopias). Es por ello que nos gustaría que estos apuntes sirvieran para un objetivo más comprometido con las personas y solidario y dedicarais parte de lo que os vais a ahorrar a una causa justa.

Sabéis que existen miles de ONG's y causas justas que necesitan de nuestra solidaridad para seguir realizando su labor y subsistir y también entiendo que cada uno de vosotros tendrá una especial sensibilización hacia alguna de ella, yo también. Pero si no es así me voy a permitir hablaros de una causa que conozco: Se trata de la **Federación Menudos Corazones**.

Esta federación **sin ánimo de lucro** intenta ayudar a los niños con **cardiopatías congénitas** y a sus familias. Para aquellos que no sepan lo qué es y lo que supone, sólo tenéis que imaginaros un recién nacido operado de corazón a los pocos días de nacer y que le esperan varias operaciones más a lo largo de su vida. Estas operaciones, en España, sólo se pueden realizar en determinados centros médicos y la estancia media en el hospital es de un mes. Un mes que el niño permanece hospitalizado y las familias, de diversas nacionalidades, desplazadas en una ciudad como Madrid sin poder trabajar y angustiadas por la salud de su hijo.

Si os parece esta una buena causa contactar con ellos en su página Web www.menudoscrazones.org, si no elegid vosotros mismos vuestra propia causa solidaria o simplemente disfrutad de vuestros apuntes.

Tu ayuda es importante

Aprovecho estas líneas para invitar a otros profesores a sumarse a esta iniciativa con alguno de sus apuntes o cualquier otro material que hayan realizado para sus alumnos. Y propongo que esta forma de editar la llamemos:

LIBROS CON CORAZÓN

Gracias a todos.

ISBN: Pendiente de adjudicación

Copyright © 2006 Jose Manuel Díaz - Salvador Ros

Todos los derechos reservados. Prohibida la copia por cualquier medio de alguna de las partes de estos apuntes sin el consentimiento por escrito de los autores.

Edita: Jose Manuel Díaz - Salvador Ros

E.T.S.I Informática.

Universidad de Educación a Distancia (UNED).

C/ Juan del Rosal nº 16. Despachos 5.03 y 5.05.

Madrid 28040

LISTA DE ABREVIATURAS

ANSI	Instituto Nacional Americano de Estándares
BSDx	UNIX Berkeley Software Distribution versión x
DF	Cargar el contenido del marco de página con el contenido de una página de un fichero ejecutable
DZ	Llenar de ceros la página física
E/S	Entrada/Salida
egid	Identificador de grupo efectivo
euid	Identificador de usuario efectivo
FIFO	Primero en entrar, primero en salir
gid	Identificador de grupo
IPC	Comunicación entre procesos
LRU	Usado menos recientemente
nodo-i	Nodo índice
nodo-im	Nodo índice cargado en memoria principal
nodo-v	Nodo virtual
npi	Nivel de prioridad de interrupción
pid	Identificador del proceso
s5fs	Sistema de ficheros estándar del UNIX System V
sfv	Sistema de ficheros virtual
SVRx	UNIX System V versión x
Tabla dbd	Tabla de descriptores de bloques de disco
Tabla dmp	Tabla de datos de los marcos de página
uid	Identificador de usuario

Prefacio

Estos apuntes son el material de estudio básico de la asignatura *Sistemas Operativos II*, que se engloba dentro de los estudios del tercer curso de la Ingeniería Técnica de Sistemas de la Escuela Superior de Informática de la UNED.

En esta asignatura se profundiza en los conceptos impartidos en la asignatura de Sistemas Operativos I. Para ello se introduce las principales estructuras de datos y algoritmos implicadas en la arquitectura del sistema operativo UNIX. Este sistema escrito en lenguaje C está implantado ampliamente en el mercado informático. Asimismo UNIX, es la base del sistema operativo de libre distribución *Linux* que cada vez está comenzando a interesar a un mayor número de usuarios y de empresas.

El objetivo fundamental de esta asignatura es dar una visión interna básica del sistema operativo UNIX, de tal forma que el alumno sea capaz de comprender de forma global el funcionamiento de un sistema operativo. Asimismo se pretende que el alumno comprenda como se interrelacionan todas las estructuras de datos, las llamadas al sistema y los algoritmos que conforman un sistema operativo para garantizar el correcto funcionamiento del mismo. Finalmente, otro objetivo de esta asignatura es que el alumno aprenda el lenguaje de programación C, uno de los más utilizados hoy en día.

El temario de esta asignatura está adecuado para ser estudiado en un cuatrimestre, en consecuencia, los contenidos que se exponen en cada tema han tenido que ser reducidos y, en algunos casos, simplificados. Asimismo, algunos temas se han quedado en el tintero. Así, el temario se ha dividido en nueve temas; puesto que el núcleo de UNIX está escrito principalmente en lenguaje C, en el Tema 1 se realiza una introducción a este útil lenguaje de programación. En el Tema 2, con el objetivo de dar al alumno una idea global de UNIX, se realizan unas consideraciones generales sobre este sistema operativo. En el Tema 3, se dan unas nociones básicas sobre la administración de UNIX. Estos tres primeros temas constituyen el primer bloque de contenidos, su estudio no debería suponer ninguna dificultad para el alumno/a.

En el Tema 4 se estudian principalmente las estructuras de datos que mantiene el núcleo de UNIX para poder soportar la ejecución de distintos programas o procesos. El Tema 5 se dedica a describir los principales algoritmos que el núcleo utiliza para controlar

la ejecución de los procesos. Estos dos temas constituyen el segundo bloque de contenidos, sin duda el más rico conceptualmente, su estudio requerirá de cierto esfuerzo.

El Tema 6 se describe la planificación de procesos en UNIX. Por su parte, en el Tema 7 se estudian los mecanismos de comunicación entre los procesos en UNIX. Estos dos temas, independientes entre sí, constituyen el tercer bloque de contenidos; su estudio no debería suponer mucho mayor esfuerzo que el bloque anterior.

El Tema 8 se dedica a la descripción de los sistemas de ficheros en UNIX. Finalmente, en el Tema 9 se estudia la gestión de memoria en UNIX. Estos dos temas, constituyen el cuarto y último bloque de contenidos; su estudio requerirá un mayor esfuerzo que los bloques anteriores.

Con el objetivo de practicar con los contenidos que vaya aprendiendo en cada tema, recomendamos al alumno/a que se instale en su PC cualquier distribución de Linux, de las muchas existentes en Internet, asegurándose de disponer de un compilador de lenguaje C, por ejemplo, `gcc`.

Con respecto a las ediciones anteriores de estos apuntes, esta cuarta edición supone un salto de calidad bastante considerable, fruto de siete meses intensos de trabajo. Así, se han reestructurado los contenidos con el objetivo de asegurar que el aprendizaje de la asignatura se puede realizar de forma secuencial, de tal forma que al llegar a un tema el alumno disponga de los conocimientos necesarios para su adecuada comprensión. Además hemos puesto especial hincapié en mejorar la explicación de los contenidos del temario, por ello la mayoría de estos apuntes, con respecto a las ediciones anteriores, han tenido que ser reescritos.

Obviamente, serán los alumnos/as y los tutores/as quiénes juzguen si el esfuerzo que hemos realizado en esta 4ª edición de los apuntes ha merecido la pena. El equipo docente estará encantado de recibir en soii@iti.uned.es las sugerencias oportunas con el objetivo de ir mejorando las futuras versiones de estos apuntes. Asimismo si encuentran erratas no duden en comunicarlas, estas se irán publicando en la web de la asignatura <http://www.ctb.dia.uned.es/asig/so2/>

Finalmente, sólo desearle que el estudio de esta asignatura le sea de interés y provecho.

INDICE

LISTA DE SIMBOLOS	i
-------------------------	---

PREFACIO	iii
----------------	-----

TEMA 1: INTRODUCCION AL LENGUAJE DE PROGRAMACION C

1.1 INTRODUCCION	1
1.2 CICLO DE CREACION DE UN PROGRAMA	2
1.3 ESTRUCTURA DE UN PROGRAMA EN C	4
1.4 CONCEPTOS BASICOS DE C	6
1.4.1 Identificadores, palabras reservadas, separadores y comentarios	6
1.4.2 Constantes	7
1.4.3 Variables	8
1.4.4 Tipos fundamentales de datos	9
1.4.5 Tipos derivados de datos	11
1.5 EXPRESIONES Y OPERADORES EN C	18
1.5.1 Operadores aritméticos	18
1.5.2 Operadores de relación y lógicos	19
1.5.3 Operadores para el manejo de bits	20
1.5.4 Expresiones abreviadas	21
1.6 ENTRADA Y SALIDA DE DATOS EN C	21
1.6.1 Entrada de un carácter: función <i>getchar</i>	21
1.6.2 Salida de un carácter: función <i>putchar</i>	21
1.6.3 Introducción de datos: función <i>scanf</i>	22
1.6.4 Escritura de datos: función <i>printf</i>	24
1.6.5 Las funciones <i>gets</i> y <i>puts</i>	25
1.7 INSTRUCCIONES DE CONTROL EN C	26
1.7.1 Propositiones y bloques	26
1.7.2 Ejecución condicional.	26
1.7.3 Bucles	28
1.7.4 Las instrucciones <i>break</i> y <i>continue</i>	29
1.7.5 La instrucción <i>switch</i>	30
1.8 FUNCIONES	31
1.8.1 Definición de una función	31
1.8.2 Prototipos de funciones y acceso a una función	32
1.8.3 Paso de argumentos a una función	34
1.8.4 Punteros a funciones	36

1.8.5 Argumentos de la función <i>main()</i>	38
1.9 MACROS	40
1.10 ASIGNACION DINAMICA DE MEMORIA.....	42

TEMA 2: CONSIDERACIONES GENERALES DEL SISTEMA OPERATIVO UNIX

2.1 INTRODUCCION	45
2.2 HISTORIA DEL SISTEMA OPERATIVO UNIX.....	47
2.2.1 Origenes	47
2.2.2 La distribución BSD de UNIX	48
2.2.3 La distribución System V de UNIX.....	49
2.2.4 Comercialización de UNIX.....	49
2.2.5 Estándares para compatibilidad en UNIX.....	50
2.2.6 Las organizaciones OSF y UI	52
2.2.7 La distribución SVR4 y más allá	53
2.3 ARQUITECTURA DEL SISTEMA OPERATIVO UNIX.....	53
2.4 SERVICIOS REALIZADOS POR EL NUCLEO.....	55
2.5 MODOS DE EJECUCION.....	56
2.5.1 Modo usuario y modo núcleo	56
2.5.2 Tipos de procesos	57
2.5.3 Interrupciones y Excepciones.....	58
2.6 ESTRUCTURA DEL SISTEMA OPERATIVO UNIX	60
2.6.1 Nivel de usuario	61
2.6.2 Nivel del núcleo	61
2.7 EL INTERFAZ DE USUARIO PARA EL SISTEMA DE FICHEROS	64
2.7.1 Ficheros y directorios	64
2.7.2 Atributos de un fichero.....	66
2.7.3 Modo de un fichero.....	68
2.7.4 Descriptores de ficheros.....	71
2.7.5 Operaciones de E/S con un fichero	74

TEMA 3: ADMINISTRACION BASICA DEL SISTEMA UNIX

3.1 INTRODUCCION	81
3.2 PRIMEROS PASOS EN UNIX.....	83
3.2.1 Entrar al sistema.....	83
3.2.2 Consolas virtuales	83
3.2.3 Intérpretes de comandos.....	84
3.2.4 Comandos básicos	85
3.3 CONSIDERACIONES GENERALES DE LOS INTÉRPRETES DE COMANDOS... 92	
3.3.1 Tipos de intérpretes de comandos	92

3.3.2 Variables del intérprete de comandos y el entorno	93
3.3.3 La variable de entorno PATH.....	94
3.3.4 Caracteres comodines	95
3.3.5 Scripts del Intérprete de Comandos	96
3.4 GESTION DE USUARIOS.....	97
3.4.1 Cuentas de usuario.....	97
3.4.2 Creación y eliminación de una cuenta de usuario	99
3.4.3 Modificación de la información asociada a una cuenta de usuario.....	100
3.4.4 Grupos de usuarios.....	100
3.5 CONFIGURACION DE LOS PERMISOS DE ACCESO A UN FICHERO	101
3.5.1 Máscara de modo simbólica	101
3.5.2 Configuración de la máscara de modo de un fichero	105
3.5.3 Consideraciones adicionales	106
3.6 CONTROL DE TAREAS.....	106
3.6.1 Visualización de los procesos en ejecución.....	107
3.6.2 Primer plano y segundo plano	108
3.6.3 Eliminación de procesos	110

TEMA 4: ESTRUCTURACION DE LOS PROCESOS EN UNIX

4.1 INTRODUCCION	113
4.2 ESPACIO DE DIRECCIONES DE MEMORIA VIRTUAL ASOCIADO A UN PROCESO	115
4.2.1 Formato lógico de un archivo ejecutable	115
4.2.2 Regiones de un proceso	116
4.2.3 Operaciones con regiones implementadas por el núcleo	118
4.3 IDENTIFICADORES NUMERICOS ASOCIADOS A UN PROCESO	120
4.3.1 Identificador del proceso	120
4.3.2 Identificadores de usuario y de grupo	121
4.4 ESTRUCTURAS DE DATOS DEL NUCLEO ASOCIADAS A LOS PROCESOS ..	125
4.4.1 Pila del núcleo.....	125
4.4.2 Tabla de procesos	129
4.4.3 Area U.....	130
4.4.4 Tabla de regiones por proceso	132
4.4.5 Tabla de regiones	132
4.5 CONTEXTO DE UN PROCESO	134
4.5.1 Definición	134
4.5.2 Parte estática y parte dinámica del contexto de un proceso	135
4.5.3 Salvar y restaurar el contexto de un proceso	139
4.5.4 Cambio de contexto	140
4.6 TRATAMIENTO DE LAS INTERRUPCIONES.....	141
4.7 INTERFAZ DE LAS LLAMADAS AL SISTEMA.....	143

4.8 ESTADOS DE UN PROCESO	150
4.8.1 Consideraciones generales	150
4.8.2 Estados adicionales.....	154
4.8.3 El estado dormido.....	154

TEMA 5: CONTROL DE PROCESOS EN UNIX

5.1 INTRODUCCION	159
5.2 CREACION DE PROCESOS.....	160
5.3 SEÑALES	168
5.3.1 Generación y tratamiento de señales.....	168
5.3.2 Problemas de consistencia en el mecanismo de señalización.....	177
5.3.3 Llamadas al sistema para el manejo de señales.....	180
5.4 DORMIR Y DESPERTAR A UN PROCESO.....	189
5.4.1 Algoritmo sleep()	189
5.4.2 Algoritmo wakeup().....	192
5.5 TERMINACION DE PROCESOS.....	194
5.6 ESPERAR LA TERMINACIÓN DE UN PROCESO	197
5.7 INVOCACION DE OTROS PROGRAMAS	200

TEMA 6: PLANIFICACION DE LOS PROCESOS EN UNIX

6.1 INTRODUCCION	205
6.2 TRATAMIENTO DE LAS INTERRUPCIONES DEL RELOJ	207
6.2.1 Consideraciones generales	207
6.2.2 Callouts.....	208
6.2.3 Alarmas	210
6.2.4 Llamadas al sistema asociadas con el tiempo	212
6.3 PLANIFICACIÓN TRADICIONAL EN UNIX.....	217
6.3.1 Prioridades de planificación de un proceso	217
6.3.2 Implementación del planificador	221
6.3.3 Manipulación de las colas de ejecución	222
6.3.4 Análisis	225
6.4 SINCRONIZACIÓN	226

TEMA 7: COMUNICACIÓN ENTRE PROCESOS EN UNIX

7.1 INTRODUCCION	229
7.2 SERVICIOS IPC UNIVERSALES.....	230
7.2.1 Señales	230
7.2.2 Tuberías.....	231
7.2.3 Seguimiento de procesos	237
7.3 MECANISMOS IPC DEL SYSTEM V.....	238
7.3.1 Consideraciones generales	238
7.3.2 Semáforos.....	244
7.3.3 Colas de mensajes	251
7.3.4 Memoria Compartida.....	261

TEMA 8: SISTEMAS DE ARCHIVOS EN UNIX

8.1 INTRODUCCION	269
8.2 FICHEROS ESPECIALES.....	270
8.3 MONTAJE DE SISTEMAS DE FICHEROS.....	273
8.3.1 Consideraciones generales	273
8.3.2 Llamadas al sistema y comandos asociados al montaje de sistema de ficheros.....	277
8.4 ENLACES SIMBOLICOS	279
8.5 LA CACHÉ DE BUFFERS DE BLOQUES	283
8.5.1 Funcionamiento básico	286
8.5.2 Cabeceras de los buffers	288
8.5.3 Ventajas	289
8.5.4 Inconvenientes.....	289
8.6 EL INTERFASE NODO-V/SFV.....	290
8.6.1 Una breve introducción a la programación orientada a objetos.....	291
8.6.2 Perspectiva general del interfaz node-v/sfv	293
8.6.3 Nodos virtuales y ficheros abiertos	297
8.6.4 El contador de referencias del nodo-v	299
8.6.5 Objetos dependientes del sistema de ficheros	300
8.6.6 Montaje de un sistema de ficheros	303
8.6.7 Operaciones sobre ficheros	304
8.7 EL SISTEMA DE FICHEROS DEL UNIX SYSTEM V (S5FS).....	309
8.7.1 Organización en el disco del s5fs	309
8.7.2 Directorios.....	310
8.7.3 Nodos-i.....	311
8.7.4 El superbloque	318
8.7.5 Organización en la memoria principal del s5fs	320
8.7.6 Análisis del s5fs	323

TEMA 9: GESTION DE MEMORIA EN UNIX

9.1 INTRODUCCION	325
9.2 POLITICA DE DEMANDA DE PÁGINAS EN EL SVR3	330
9.2.1 Estructuras de datos asociadas a la gestión de memoria mediante demanda de páginas	331
9.2.2 La realización de la llamada al sistema fork en un sistema con paginación	339
9.2.3 Exec en un sistema de paginación	341
9.2.4 Transferencia de páginas de memoria principal al área de intercambio	343
9.2.5 Tratamiento de los fallos de página	346
9.2.6 Explicación desde el punto de vista de la gestión de memoria del cambio de modo de un proceso	355
9.2.7 Localización en memoria del área U de un proceso	357
 BIBLIOGRAFIA	 359

TEMA 1

INTRODUCCION AL LENGUAJE DE PROGRAMACION C

1.1 INTRODUCCION

C es un lenguaje de programación de alto nivel desarrollado por Dennis Ritchie para codificar el sistema operativo UNIX. Las primeras versiones de UNIX se escribieron en ensamblador, pero a partir de 1973 pasaron a escribirse en C. Actualmente, sólo un pequeño porcentaje del núcleo de UNIX se sigue codificando en ensamblador; en concreto aquellas partes íntimamente relacionadas con el hardware. Todas las órdenes y aplicaciones estándar que acompañan al sistema UNIX están escritas también en C.

El lenguaje posee instrucciones que constan de términos que se parecen a expresiones algebraicas, además de ciertas palabras clave inglesas como `if`, `else`, `for`, `do` y `while`. En este sentido, C recuerda a otros lenguajes de programación estructurados como Pascal y Fortran.

El lenguaje C presenta las siguientes características:

- Se puede utilizar para programación a bajo nivel cubriendo así el vacío entre el lenguaje máquina y los lenguajes de alto nivel más convencionales.
- Permite la redacción de programas fuentes muy concisos, debido en parte al gran número de operadores que incluye el lenguaje.
- Tiene un repertorio de instrucciones básicas relativamente pequeño. Aunque incluye numerosas funciones de biblioteca que mejoran las instrucciones básicas. Además los usuarios pueden escribir bibliotecas adicionales para su propio uso.

- Los compiladores de C son frecuentemente compactos y generan programas objeto que son pequeños y muy eficientes.
- Los programas escritos en C son muy portables. C deja en manos de las funciones de biblioteca la mayoría de las características dependientes de la computadora. De esta forma, la mayoría de los programas en C se pueden compilar y ejecutar en muchas computadoras diferentes sin tener que realizar en la mayoría de los casos ninguna modificación en los programas.

1.2 CICLO DE CREACION DE UN PROGRAMA

Un *compilador* es un programa que toma como entrada un texto escrito en un lenguaje de programación de alto nivel, denominado *fuentes*, y da como salida otro texto en un lenguaje de bajo nivel (ensamblador o código máquina), denominado *objeto*. Asimismo, un *ensamblador* es un compilador cuyo lenguaje fuente es el lenguaje ensamblador.

Un compilador no es un programa que funciona de manera aislada, sino que normalmente se apoya en otros programas para conseguir su objetivo: obtener un programa ejecutable a partir de un programa fuente en un lenguaje de alto nivel. Algunos de esos programas son:

- *El preprocesador*. Se ocupa (dependiendo del lenguaje) de incluir ficheros, expandir macros, eliminar comentarios, y otras tareas similares.
- *El enlazador (linker)*. Se encarga de construir el fichero ejecutable añadiendo al fichero objeto generado por el compilador las cabeceras necesarias y las funciones de librería utilizadas por el programa fuente.
- *El depurador (debugger)*. Permite, si el compilador ha generado adecuadamente el programa objeto, seguir paso a paso la ejecución de un programa.
- *El ensamblador*. Muchos compiladores en vez de generar código objeto, generan un programa en lenguaje ensamblador que debe después convertirse en un ejecutable mediante un programa ensamblador.

A la hora de crear un programa en C, se ha de empezar por la edición de un fichero de texto estándar que va a contener el código fuente escrito en C. Este fichero se

nombrar, por convenio, añadiéndole la extensión `.c`. Si se utiliza el editor `vi` disponible en UNIX, la forma de editar el programa desde la línea de comandos del terminal (\$) es:

```
$ vi prog.c
```

Los compiladores de C más utilizados son el `cc` y el `gcc` que se encargan de generar el fichero ejecutable a partir del fichero fuente escrito en C. Para invocarlo, desde la línea de comandos del terminal se teclea:

```
$ gcc prog.c
```

Esta línea de órdenes va a provocar que se genere el fichero `a.out`, que ya es ejecutable. Si se quiere personalizar el nombre del fichero de salida se escribirá la orden

```
$ gcc -o nombre_ejecutable prog.c
```

De esta manera se creará un programa ejecutable con nombre `nombre_ejecutable`.

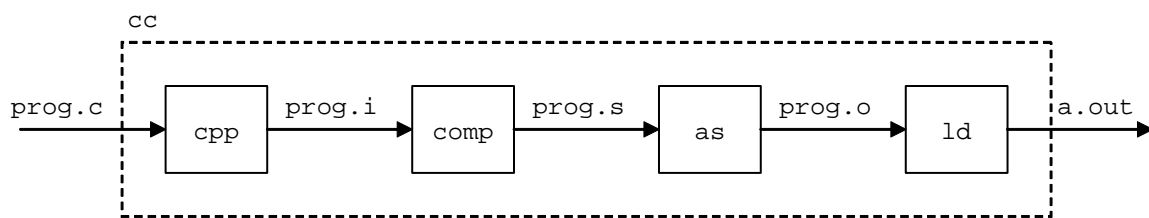


Figura 1.1: Fases del proceso de compilación

Con respecto a `cc` comentar que en realidad no es el compilador sino un interfaz entre el usuario y los programas que intervienen en el proceso de generación de un programa ejecutable (ver Figura 1.1) y que son los siguientes:

- El *preprocesador* `cpp`, que genera un archivo con extensión `*.i`.
- El *compilador* `comp`, que genera un archivo con extensión `*.s` que contiene código fuente ensamblador
- El *ensamblador* `as`, que genera un archivo con extensión `*.o` que contiene código objeto.
- El *enlazador* `ld`, que genera el programa ejecutable con extensión `*.out` a partir de ficheros con código objeto (`.o`) y bibliotecas (`.a`).

1.3 ESTRUCTURA DE UN PROGRAMA EN C.

Todo programa C consta de uno o más módulos llamados *funciones*. Una de estas funciones es la función principal que se llama `main`. El programa siempre comenzará por la ejecución de la función `main`, la cual puede acceder a las demás funciones. Las definiciones de las funciones adicionales se deben realizar aparte, bien precediendo o bien siguiendo a `main`. De forma general, se puede afirmar que la estructura de un programa en C es la que se muestra en el Cuadro 1.1.

```
# directivas del preprocesador.
definición de variables globales.
definición prototipo de la función 1
.
.
definición prototipo de la función N

función main()
{
    definición de variables locales
    código
}
funcion1(parámetros formales)
{
    definición de variables locales
    código
}
.
.
funcionN(parámetros formales)
{
    definición de variables locales
    código
}
```

Cuadro 1.1: Estructura general de un programa en C

En primer lugar, se escriben las *directivas del preprocesador*, que son órdenes que ejecuta el preprocesador para generar el fichero con el que va a trabajar el procesador. Dos son las directivas más utilizadas:

- `#include <xxxx.h>`. Que se emplea para indicar al compilador que recupere el código de un *fichero de cabecera o librería* `xxxx.h` donde están identificadores, constantes, variables globales, macros, prototipos de funciones, etc. La utilización de los archivos de cabecera permite tener las declaraciones fuera del programa principal. Esto implica una mayor modularidad.
- `#define`. Que se emplea para declarar identificadores que van a ser sinónimos de otros identificadores o constantes. También se emplea para declarar *macros*.

En segundo lugar, se escriben las *declaraciones de las variables globales* del programa, en el caso de que existan, que pueden ser utilizadas por todas las funciones del mismo.

En tercer lugar se escriben la *declaración de los prototipos* de las N funciones que se vayan a utilizar en el programa (salvo `main`). En cuarto lugar se escribe el *cuerpo del programa* o función `main`. Y finalmente se precede a escribir las N funciones cuyos prototipos se han definido anteriormente.

♦ Ejemplo 1.1:

Considérese el siguiente programa

```
/* Mi primer programa de C*/
#include <stdio.h>
void main(void)
{
    printf( " HOLA A TODOS " );
}
```

Programa 1.1

La primera línea del programa es un comentario, en C se escriben comenzando con `/*` y terminando con `*/`. La segunda línea es una directiva del preprocesador del tipo `#include` que hace referencia al fichero de cabecera o librería `stdio.h`. Finalmente las restantes líneas corresponden al cuerpo del programa (función `main`), que en este caso solo consta de una sentencia simple del tipo `printf`. De forma general todas las sentencias simples terminan en `;`.

Cuando se ejecute este programa se mostrará por pantalla el mensaje " HOLA A TODOS ". En este ejemplo la única función existente en el programa es la función principal o función `main`.



1.4 CONCEPTOS BASICOS DE C

1.4.1 Identificadores, palabras reservadas, separadores y comentarios

Un identificador es una secuencia de letras o dígitos donde el primer elemento debe ser una letra, o los caracteres `_` y `$`. Las letras mayúsculas y minúsculas se consideran distintas. Los identificadores son los nombres que se utilizan para representar variables, constantes, tipos y funciones de un programa. El compilador sólo reconoce los 32 primeros caracteres del identificador, pero éste puede ser de cualquier tamaño.

El lenguaje C distingue entre letras mayúsculas y minúsculas. Por ejemplo, el identificador `valor` es distinto a los identificadores `VALOR` y `Valor`

Las *palabras reservadas* son identificadores predefinidos que tienen un significado especial para el compilador de C. Las palabras claves siempre van en minúsculas. En la Tabla 1.1 se recogen las palabras reservadas según ANSI C, que es la definición estandarizada del lenguaje C creada por el Instituto Nacional Americano de Estándares (ANSI¹).

<code>auto</code>	<code>double</code>	<code>int</code>	<code>struct</code>
<code>break</code>	<code>else</code>	<code>long</code>	<code>switch</code>
<code>case</code>	<code>enum</code>	<code>register</code>	<code>typedef</code>
<code>char</code>	<code>extern</code>	<code>return</code>	<code>union</code>
<code>const</code>	<code>float</code>	<code>short</code>	<code>unsigned</code>
<code>continue</code>	<code>for</code>	<code>signed</code>	<code>void</code>
<code>default</code>	<code>goto</code>	<code>sizeof</code>	<code>volatile</code>
<code>co</code>	<code>if</code>	<code>static</code>	<code>while</code>

Tabla 1.1: Palabras reservadas según el estándar ANSI C

El lenguaje C también utiliza una serie de caracteres como elementos *separadores*: `{, }, [,], (,), ;, -, >, ..`

En C es posible escribir *comentarios* como una secuencia de caracteres que se inicia con `/*` y termina con `*/`. Los comentarios son ignorados por el compilador.

¹ ANSI es el acrónimo derivado del término inglés “*American National Standards Institute*”.

1.4.2 Constantes

Las constantes se refieren a valores fijos que no se pueden alterar por el programa. El lenguaje C tiene cuatro tipos básicos de constantes:

- *Enteras*. Es un número con un valor entero, consistente en una secuencia de dígitos. Las constantes enteras se pueden escribir en tres sistemas numéricos distintos: decimal, octal y hexadecimal.
- *En coma flotante*. Es un número decimal que contiene un punto decimal o un exponente (o ambos).
- *De carácter*. Es un sólo carácter encerrado con comillas simples.
- *De cadenas de caracteres*. Consta de cualquier número de caracteres consecutivos encerrados entre comillas dobles.

Se denomina *secuencia de escape* a una combinación de caracteres que comienza siempre con una barra inclinada hacia atrás \ y es seguida por uno o más caracteres especiales. Una secuencia de escape siempre representa un sólo carácter, aún cuando se escriba con dos o más caracteres. En la Tabla 1.2 se listan las secuencias de escape utilizadas más frecuentemente:

Carácter	Secuencia de escape
Sonido (alerta)	\a
Tabulador horizontal	\t
Tabulador vertical	\v
Nueva línea	\n
Comillas	\"
Comilla simple	\'
Barra inclinada hacia atrás	\\
Signo de interrogación	\?
Nulo	\0

Tabla 1.2: Secuencias de escape

♦ Ejemplo 1.2:

```
0      1      743    5280   32767 9999          (Constantes enteras decimales)

0x     0X     0X7FFF          0xabcd          (Constantes enteras hexadecimales)
```

0.1 1.0 827.602 2E-8 1.666E12 (Constantes en coma flotante)

'A' 'X' '?' ' ' 'd' (Constantes de carácter)

"Verde" "La respuesta correcta es" (Constantes de cadena de caracteres)



Las constantes se declaran colocando el modificador `const` delante del tipo de datos. Otra forma de definir constantes es usando la directiva de compilación `#define`.

◆ Ejemplo 1.3:

La siguiente sentencia declara la constante `MAXIMO` de tipo entero que es inicializada al valor 9.

```
const int MAXIMO=9;
```

Otra forma equivalente de declararla es con la sentencia:

```
#define MAXIMO 9
```

Esta sentencia se ejecuta de la siguiente forma: En la fase de compilación al ejecutar `#define` el compilador sustituye cada aparición de `MAXIMO` por el valor 9. Además no se permite asignar ningún valor a esa constante. Es importante darse cuenta que esta declaración no acaba en punto y coma ';'



1.4.3 Variables

Una *variable* es un identificador que se utiliza para representar cierto tipo de información dentro de una determinada parte del programa. En su forma más sencilla, una variable es un identificador que se utiliza para representar un dato individual; es decir, una cantidad numérica o una constante de carácter. En alguna parte del programa se asigna el dato a la variable. Este valor se puede recuperar después en el programa simplemente haciendo referencia al nombre de la variable.

A una variable se le pueden asignar diferentes valores en distintas partes del programa. De esta forma la información representada puede cambiar durante la ejecución del programa. Sin embargo, el tipo de datos asociado a la variable no puede cambiar.

Las variables se declaran de la siguiente forma:

```
tipo nombre_variable;
```

Donde `tipo` será un tipo de datos válido en C con los modificadores necesarios y `nombre_variable` será el *identificador* de la misma.

Las variables se pueden declarar en diferentes puntos de un programa:

- Dentro de funciones. Las variables declaradas de esta forma se denominan *variables locales*.
- En la definición de funciones. Las variables declaradas de esta forma se denominan *parámetros formales*.
- Fuera de todas las funciones. Las variables declaradas de esta forma se denominan *variables globales*.

La inicialización de una variable es de la forma:

```
nombre_de la variable= constante;
```

Donde `constante` debe ser del tipo de la variable.

♦ Ejemplo 1.4:

```
/*Declaración de variables*/
char letra;
int entero;
float real;
/*Inicialización de variables*/
letra='a';      /*Se usan comillas simples*/
entero=234;
real =123.333;
```

♦

1.4.4 Tipos fundamentales de datos

En el lenguaje C se consideran dos grandes bloques de datos:

- *Tipos fundamentales*: que son suministrados por el lenguaje.

- *Tipos derivados*, que son definidos por el programador.

Los *tipos fundamentales* se clasifican en:

- *Tipos enteros*. Se utilizan para representar subconjuntos de los números naturales y enteros.
- *Tipos reales*. Se emplean para representar un subconjunto de los números racionales.
- Tipo `void`. Sirve para declarar explícitamente funciones que no devuelven ningún valor. También sirve para declarar punteros genéricos.

1.4.4.1 Tipos enteros

Se distinguen los siguientes tipos enteros:

- `char`. Define un número entero de 8 bits. Su rango es [-128, 127]. También se emplea para representar el conjunto de caracteres ASCII (Código Estándar Americano para el Intercambio de Información).
- `int`. Define un número entero de 16 o 32 bits (dependiendo del procesador).
- `long`. Define un número entero de 32 o 64 bits.
- `short`. Define un número entero de tamaño menor o igual que `int`. En general se cumple que: $\text{tamaño}(\text{short}) \leq \text{tamaño}(\text{int}) \leq \text{tamaño}(\text{long})$.

Los cuatro primeros tipos pueden ir precedidos del modificador `unsigned` que indica que el tipo sólo representa números positivos o el cero.

♦ Ejemplo 1.5:

```
int numero=3;
char letra='c';
unsigned short contador=0;
```

Con las anteriores sentencias se han declarado la variable `numero` de tipo `int` inicializada a 3, la variable `letra` de tipo `char` inicializado a 'c' y la variable `contador` de tipo `unsigned short` inicializado a 0.

1.4.4.2 Tipos reales

Se distinguen los siguientes tipos reales:

- `float`. Define un número en coma flotante de precisión simple. El tamaño de este tipo suele ser de 4 bytes (32 bits).
- `double`. Define un número en coma flotante de precisión doble. El tamaño de este tipo suele ser de 8 bytes (64 bits). Este tipo puede ir precedido del modificador `long`, que indica que su tamaño pasa a ser de 10 bytes.

◆ Ejemplo 1.6:

```
float ganancia=125.23;
double diametro=12.3E53;
```

Con las anteriores sentencias se han declarado la variable `ganancia` de tipo `float` inicializada a 125.3 y la variable `diametro` de tipo `double` inicializada a '12.3E53'.

◆

1.4.5 Tipos derivados de datos

Los tipos derivados se construyen a partir de los tipos fundamentales o de otros tipos derivados. Se van a describir las características de los siguientes: arrays, punteros, estructuras y uniones.

1.4.5.1 Arrays

Un *array* es una colección de variables del mismo tipo que se referencian por un nombre común. El compilador reserva espacio para un array y le asigna posiciones de memoria contiguas. La dirección más baja corresponde al primer elemento y la más alta al último. Se puede acceder a cada elemento de un array con un índice. Los índices, para acceder al array, deben ser variables o constantes de tipo entero. Se define la dimensión de un array como el total de índices que necesitamos para acceder a un elemento particular del array.

La definición formal de una array N-dimensional es la siguiente:

```
tipo_array nombre_array [rango1] [rango1]... [rangoN];
```

A los arrays unidimensionales se les denomina *vectores*, y a los bidimensionales se les denomina *matrices*.

A los arrays unidimensionales de tipo `char` se les denomina *cadena de caracteres*, y a los arrays de cadenas de caracteres (matrices de caracteres) se les denomina *tablas*.

Para indexar los elementos de un array, se debe tener en cuenta que los índices deben variar entre 0 y M-1, donde M es el tamaño de la dimensión a la que se refiere el índice.

◆ **Ejemplo 1.7:**

La declaración de una variable denominada `matriz_A` de números de tipo `float` de dos filas y dos columnas sería de la siguiente forma:

```
float matriz_A[2][2];
```

La declaración de una variable denominada `x` que es un vector de 4 números de tipo `int` sería de la siguiente forma:

```
int x[4];
```

Para el vector `x` anteriormente declarado, el índice variará entre 0 y 3: `x[0]`, `x[1]`, `x[2]`, `x[3]`;

◆

1.4.5.2 Punteros

Un *puntero* es una variable que es capaz de guardar una dirección de memoria, dicha dirección es la localización de otra variable en memoria. Así, si una variable A contiene la dirección de otra variable B decimos que A apunta a B, o que A es un puntero a B.

Los punteros se definen en base a un tipo fundamental o a un tipo derivado. La declaración de un puntero se realiza de la siguiente forma:

```
tipo_base *puntero;
```

Los punteros una vez declarados contienen un valor desconocido. Si se intenta usar sin inicializar podemos incluso dañar el sistema operativo. La forma de inicializarlos consiste en asignarles una dirección conocida.

Existen dos operadores que se utilizan para trabajar con punteros:

- El operador `&`, da la dirección de memoria asociada a una variable y se utiliza para inicializar un puntero.

- El operador `*`, se utiliza para referirse al contenido de una dirección de memoria.

♦ Ejemplo 1.8:

Supóngase la siguiente sentencia

```
int *z;
```

Se trata de la declaración de la variable puntero `z`, que contiene la dirección de memoria de un número de tipo `int`.

Supóngase ahora las siguientes tres sentencias:

```
int z, *pz;
```

```
pz=&z;
```

```
*pz=25;
```

La primera sentencia está declarando un variable `z` de tipo entero y una variable puntero `pz` que contiene la dirección de una variable de tipo `int`.

La segunda sentencia, inicializa la variable puntero `pz` a la dirección de la variable `z`.

Por último la tercera sentencia es equivalente a la sentencia '`z=25;`'. A través de `pz` se puede modificar de una forma indirecta el valor de la variable `z`.

En la Figura 1.2 se representa la relación entre el puntero `pz` y la variable `z`.

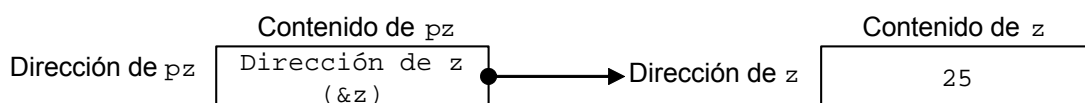


Figura 1.2: Relación entre el puntero `pz` y la variable `z`

El acceso a una variable a través de un puntero se conoce también como *indirección*. Obviamente antes de manipular el contenido de un puntero hay que inicializar el puntero para que referencie a la dirección de dicha variable.

En general no tiene sentido asignar un valor entero a una variable puntero. Pero una excepción es la asignación de 0, que a veces se utiliza para indicar condiciones

especiales. En tales situaciones, la práctica recomendada es definir una constante simbólica `NULL` que representa el valor 0 y usar `NULL` en la inicialización del puntero. Un ejemplo de esta forma de inicialización es

```
#define NULL 0
float *pv=NULL;
```

Por otra parte, los punteros admiten las operaciones de incremento, decremento, suma de una constante entera y diferencia de punteros. Al realizar estas operaciones se está modificando la dirección a la que apunta el puntero.

Los punteros son usados frecuentemente en C ya que tienen una gran cantidad de aplicaciones. Por ejemplo, pueden ser usados para transvasar información entre una función y sus puntos de llamada. En particular proporcionan una forma de devolver varios datos desde una función a través de los argumentos de una función. Los punteros también permiten que referencias a otras funciones puedan ser especificadas como argumentos de una función. Esto tiene el efecto de pasar funciones como argumentos de una función dada.

Los punteros están muy relacionados con los arrays y proporcionan una vía alternativa de acceso a los elementos individuales del array. Se puede acceder a cualquier posición del array usando el nombre y el índice (indexación) o bien con un puntero y la aritmética de punteros.

En general, el nombre de un array es realmente un puntero al primer elemento de ese array. Por tanto, si `x` es un array unidimensional, entonces la dirección al primer elemento del array se puede expresar tanto como `&x[0]` o simplemente como `x`. Además la dirección del segundo elemento del array se puede escribir tanto como `&x[1]` o como `(x+1)`, y así sucesivamente. En general, la dirección del elemento `(i+1)` de la formación se puede expresar bien como `&x[i]` o como `(x+i)`.

♦ **Ejemplo 1.9:**

```
float vector[5];
float *p=vector; /*Inicialización del puntero*/
*p=10.2; /*Esta sentencia equivale a vector[0]=10.2*/
p=p+3;
*p=0.5 ; /*Esta sentencia equivale a vector[3]=0.5*/
```

El anterior conjunto de sentencias muestra como es posible acceder a los elementos del array `vector` que consta de 5 elementos a través del puntero `p`.

Indicar que otra forma de inicialización del puntero `p` es usando la dirección de la primera posición del array

```
float p=&vector[0];
```



1.4.5.3 Estructuras

Una estructura es un agregado de tipos fundamentales o derivados que se compone de varios campos. A diferencia de los arrays, cada elemento de la estructura puede ser de un tipo diferente. La forma de definir una estructura es la siguiente:

```
struct nombre_estructura
{
    tipo1 campo1;
    tipo2 campo2;
    ...
    tipoN campoN;
}
```

Para acceder a los campos de una estructura se utiliza el operador `'.'`.

Es posible declarar:

- Arrays de estructuras.
- Punteros a estructuras. En este caso, el acceso a los campos de la variable se hace por medio del operador `'->'`.

Además, puesto que el tipo de cada campo de una estructura puede ser un tipo fundamental o derivado, también puede ser otra estructura. Con lo que es posible declarar estructuras dentro de estructuras.

◆ Ejemplo 1.10:

```
struct cliente{
    int cuenta;
    char nombre[100];
    unsigned short dia;
    unsigned short mes;
    unsigned int año;
    float saldo;
}
```

```
struct cliente uno;
```

La variable `uno` es una variable declarada del tipo `struct cliente`. Con la sentencia

```
uno.cuenta=12530;
```

Se está asignando al campo `cuenta` de la estructura `uno` el valor 12530.



◆ Ejemplo 1.11:

Sea la siguiente declaración de una estructura

```
struct
{
    float real, imaginaria;
} vectorC[5];
```

La variable `vectorC` es un array unidimensional de números complejos. Para modificar la parte real del elemento 2 (recuérdese que los elementos de un array se comienzan a numerar por el 0), se escribe la sentencia:

```
vectorC[1].real=0.23;
```



◆ Ejemplo 1.12:

Supóngase ahora la siguiente declaración de una estructura:

```
struct altura
{
    char nombre[100];
    float h;
};
struct altura persona, *p;
p=&persona;
p->h=1.65;
```

El puntero `p` apunta a la estructura `persona` del tipo `altura`. Con la última sentencia se está asignando al campo `h` de la estructura `persona` el valor 1.65. Esa sentencia es equivalente a

```
persona.h=1.65;
```



◆ Ejemplo 1.13:

Un ejemplo de una estructura declarada dentro de una estructura sería:

```
struct fecha
{
    int día, mes, año;
}
struct alumno
{
    char nombre[100];
    struct fecha fecha_nacimiento;
    float nota;
};
struct alumno alumno;
```

◆

1.4.5.4 Uniones

Las uniones se definen de forma parecida a las estructuras y se emplean para almacenar en un mismo espacio de memoria variables de distintos tipos. La declaración de una unión es la siguiente:

```
union nombre_unión
{
    tipo1 campo1;
    tipo2 campo2;
    ...
    tipoN campoN;
};
```

El tamaño de la unión va a ser igual al tamaño del mayor de sus campos.

◆ Ejemplo 1.14:

```
union multiuso
{
    float numeroR;
    int numeroZ;
    char letra;
};
```

Con la variable `multiuso` se pueden hacer las siguientes asignaciones:

```
multiuso.numeroR=0.1235;
multiuso.numeroZ=-225;
multiuso.letra='j';
```



1.4.5.5 Alias para los nombres de tipo

Es posible hacer que un identificador sea considerado el nombre de un nuevo tipo, para ello hay que emplear la palabra clave `typedef`, con ella es posible actuar sobre cualquier tipo fundamental o derivado.

◆ Ejemplo 1.15:

```
typedef int entero;
entero y;
```

Con la definición de `entero` como sinónimo de `int`, es posible declarar variables de tipo `entero`.



1.5 EXPRESIONES Y OPERADORES EN C

En una expresión van a tomar parte variables, constantes y operadores. Los operadores establecen la relación entre las variables y las constantes a la hora de evaluar la expresión. Los paréntesis también pueden formar parte de una expresión y se emplean para modificar la precedencia de los operadores.

1.5.1 Operadores aritméticos

Los posibles operadores aritméticos son los que se muestran en la Tabla 1.3

Operador	Acción
-	Resta
+	Suma
*	Multiplicación
/	División ²
%	Resto de la división
--	Decremento
++	Incremento

Tabla 1.3: Operadores aritméticos

² La división de números enteros produce un truncamiento del cociente (por ejemplo, $3/2=1$).

Las expresiones aritméticas se evalúan de izquierda a derecha. Si en una expresión aritmética intervienen variables o constantes de diferentes tipos, el tipo del resultado va a coincidir con el tipo mayor que aparezca en la expresión. Por ejemplo, si se multiplica una variable `float` por una variable `int`, el resultado será `float`.

La suma y la diferencia sobre una misma variable tienen una representación simplificada mediante los operadores `++` y `--`.

♦ Ejemplo 1.16:

```
int x;
++x; /* Es equivalente a x=x+1. Preincremento*/
x++; /* Es equivalente a x=x+1. Postincremento*/
--x; /* Es equivalente a x=x-1. Predecremento*/
x--; /* Es equivalente a x=x-1. Postdecremento*/
```

La diferencia entre la posición prefija y la posición sufija de los operadores anteriores queda puesta de manifiesto en las siguientes sentencias:

```
x=10;
printf("%d\n",++x); /*Incrementa "x" en 1, por lo que imprime 11*/;
x=10;
printf("%d\n",x++); /*Imprime 10 e incrementa x en 1*/;
```

♦

1.5.2 Operadores de relación y lógicos

Tanto los operadores de relación como los operadores lógicos se emplean para formar expresiones booleanas. Recuérdese que una expresión booleana únicamente puede tomar dos valores: Verdadero (`TRUE`) o Falso (`FALSE`). En el lenguaje C, por convenio se considera que si una expresión booleana da como resultado 0 entonces su valor lógico es Falso. Por el contrario si al evaluarla su valor es distinto de 0, entonces su valor lógico es Verdadero. En la Tabla 1.4 y 1.5 se muestran los operadores lógicos y los operadores de relación, respectivamente.

Operador	Significado
<code>&&</code>	AND lógica
<code> </code>	OR lógica
<code>!</code>	Negación lógica

Tabla 1.4: Operadores lógicos

Operador	Relación
>	Mayor
>=	Mayor o igual
<	Menor
<=	Menor o igual
==	Igual
!=	Distinto

Tabla 1.5: Operadores de relación

1.5.3 Operadores para el manejo de bits

El lenguaje C dispone de operadores para la manipulación de bits o constantes enteras. Se debe tener mucho cuidado de no confundir las operaciones a nivel de bit con las operaciones lógicas. En la Tabla 1.6 se muestran los operadores para el manejo de bits.

Operador	Significado	Ejemplo
&	AND	1001&0011=>0001
	OR	1001 0011=>1011
^	XOR	1001^0011=>1010
~	Complemento a 1	~1001=>0110
<<	Desplazamiento a la izquierda	0110<<1=>1100
>>	Desplazamiento a la derecha	0110>>1=>0011 1011>>1=>1101

Tabla 1.6: Operadores para el manejo de bits

Expresión abreviada	Expresión equivalente
x+=y	x=x+y
x-=y	x=x-y
x*=y	x=x*y
x/=y	x=x/y
x&=y	x=x&y
x =y	x=x y
x^=y	x=x^y
x<<=y	x=x<<y
x>>=y	x=x>>y

Tabla 1.7: Expresiones abreviadas

1.5.4 Expresiones abreviadas

El lenguaje C permite utilizar algunas expresiones abreviadas para indicar ciertas operaciones. En la Tabla 1.7 se muestran las expresiones abreviadas más comunes.

1.6 ENTRADA Y SALIDA DE DATOS EN C

El lenguaje C va acompañado de una colección de funciones de biblioteca que incluye un cierto número de funciones de entrada/salida. Como norma general, el archivo de cabecera requerido para la entrada/salida estándar se llama `stdio.h`, entre todas las funciones que contiene algunas de las más usadas son: `getchar`, `putchar`, `scanf`, `printf`, `gets` y `puts`. Estas seis funciones permiten la transferencia de información entre la computadora y los dispositivos de entrada/salida estándar tales como un teclado y un monitor.

En las siguientes subsecciones se describen únicamente las características básicas de las funciones `getchar`, `putchar`, `scanf`, `printf`, `gets` y `puts`.

1.6.1 Entrada de un carácter: función `getchar`

Mediante la función de biblioteca `getchar` se puede conseguir la entrada de un carácter a través del dispositivo de entrada estándar, usualmente el teclado. Su sintaxis es

```
variable = getchar();
```

donde `variable` es alguna variable de tipo carácter declarada previamente.

◆ Ejemplo 1.17:

```
char c;  
c=getchar();
```

En la primera instrucción se declara la variable `c` de tipo carácter. La segunda instrucción hace que se lea del dispositivo de entrada estándar un carácter y entonces se le asigne a `c`.

◆

1.6.2 Salida de un carácter: función `putchar`

Mediante la función de biblioteca `putchar` se puede conseguir la salida de un carácter a través del dispositivo de salida estándar, usualmente el monitor. Su sintaxis es

```
putchar(variable);
```

donde `variable` es alguna variable de tipo carácter declarada previamente.

♦ **Ejemplo 1.18:**

```
char c='a';  
putchar(c);
```

En la primera instrucción se declara la variable `c` de tipo carácter. La segunda instrucción hace que se visualice el valor de `c` en el dispositivo de salida estándar (monitor).

♦

1.6.3 Introducción de datos: función `scanf`

Mediante la función de biblioteca `scanf` se puede introducir datos en la computadora a través del dispositivo de entrada estándar. Esta función permite introducir cualquier combinación de valores numéricos, caracteres sueltos y cadenas de caracteres. La función devuelve el número de datos que se han conseguido introducir correctamente. Su sintaxis es

```
scanf(cadena de control, arg1, arg2, ..., argN);
```

donde `cadena de control` hace referencia a una cadena de caracteres que contiene cierta información sobre el formato de los datos y `arg1, arg2, ..., argN` son argumentos (punteros) que indican la direcciones de memoria donde se encuentran los datos.

En la cadena de control se incluyen grupos individuales de caracteres, con un grupo de caracteres por cada dato de entrada. Cada grupo de caracteres debe comenzar con el signo de porcentaje `%`. En su forma más sencilla, un grupo de caracteres estará formado por el signo de porcentaje, seguido de un *carácter de conversión* que indica el tipo de dato correspondiente. En la Tabla 1.8 se muestran los caracteres de conversión de los datos de entrada de uso común.

Los argumentos pueden ser variables o arrays, y sus tipos deben coincidir con los indicados por los grupos de caracteres correspondientes en la cadena de control. Cada nombre de variable debe ser precedido por un ampersand (`&`), salvo en el caso de los arrays.

Carácter de conversión	Significado del dato
c	Carácter
d	Entero decimal
e	Coma flotante
f	Coma flotante
g	Coma flotante
h	Entero corto
o	Entero octal
s	Cadena de caracteres seguida de un carácter de espaciado.
u	Entero decimal sin signo
[...]	Cadena de caracteres que puede incluir caracteres de espaciado.

Tabla 1.8: Caracteres de conversión de los datos de entrada de uso común

♦ **Ejemplo 1.19:**

```
#include <stdio.h>
main()
{
    char concepto[20];
    int no_partida;
    float coste;
    scanf("%s %d %f", concepto, &no_partida, &coste);
}
```

Dentro de la función `scanf` de este programa, la cadena de control es `"%s %d %f"`. Contiene tres grupos de caracteres. El primer grupo, `%s`, indica que el primer argumento (`concepto`) representa a una cadena de caracteres. El segundo grupo, `%d`, indica que el segundo argumento (`&no_partida`) representa un valor entero decimal, y el tercer grupo, `%f`, indica que el tercer argumento (`&coste`) representa un valor en coma flotante.

Obsérvese, que las variables numéricas `no_partida` y `coste` van precedidas por ampersands dentro de la función `scanf`. Sin embargo, delante de `concepto` no hay ampersand, ya que `concepto` es el nombre del array..

♦

1.6.4 Escritura de datos: función `printf`

Mediante la función de biblioteca `printf` se puede escribir datos en el dispositivo de salida estándar. Esta función permite escribir cualquier combinación de valores numéricos, caracteres sueltos y cadenas de caracteres. Su sintaxis es

```
printf(cadena de control, arg1, arg2, ..., argN);
```

donde `cadena de control` hace referencia a una cadena de caracteres que contiene información sobre el formato de salida y `arg1, arg2, ..., argN` son argumentos que representan los datos de salida.

La cadena de control está compuesta por grupos de caracteres, con un grupo de caracteres por cada dato de salida. Cada grupo de caracteres debe empezar por un signo de porcentaje (%). En su forma sencilla, un grupo de caracteres consistirá en el signo de porcentaje seguido de un carácter de conversión que indica el tipo de dato correspondiente. En la Tabla 1.9 se muestran los caracteres de conversión de los datos de salida de uso común.

Carácter de conversión	Significado del dato visualizado
c	Carácter
d	Entero decimal con signo
e	Coma flotante con exponente
f	Coma flotante sin exponente
g	Coma flotante con o sin exponente según el caso. No se visualizan ni los ceros finales ni el punto decimal cuando no es necesario.
i	Entero con signo
o	Entero octal, sin el cero inicial
s	Cadena de caracteres
u	Entero decimal sin signo
x	Entero hexadecimal sin el prefijo 0x

Tabla 1.9: Caracteres de conversión de los datos de salida de uso común

♦ Ejemplo 1.20:

```
#include <stdio.h>
main()
{
    char concepto[20]="cremallera";
```

```
int no_partida=12345;
float coste=0.05;
printf("%s %d %f", concepto, no_partida, coste);
}
```

Dentro de la función `printf` de este programa, la cadena de control es `"%s %d %f"`. Contiene tres grupos de caracteres. El primer grupo, `%s`, indica que el primer argumento (`concepto`) representa a una cadena de caracteres. El segundo grupo, `%d`, indica que el segundo argumento (`no_partida`) representa un valor entero decimal, y el tercer grupo, `%f`, indica que el tercer argumento (`coste`) representa un valor en coma flotante.

El resultado de la ejecución de estas instrucciones del programa es visualizar en el monitor la siguiente salida:

```
cremallera 12345 0.050000
```

◆

1.6.5 Las funciones `gets` y `puts`

La función de biblioteca `gets` permite leer una cadena de caracteres desde el dispositivo de entrada estándar. Su sintaxis es

```
gets(variable);
```

donde `variable` debe ser una cadena de caracteres

La función de biblioteca `puts` permite visualizar una cadena de caracteres en el dispositivo de salida estándar. Su sintaxis es

```
puts(variable);
```

donde `variable` debe ser una cadena de caracteres

◆ Ejemplo 1.21:

```
/*El siguiente código permite leer y escribir una línea de texto*/
#include <stdio.h>
main()
{
    char linea[80];
    gets(linea);
    puts(linea);
}
```

◆

1.7 INSTRUCCIONES DE CONTROL EN C

1.7.1 Proposiciones y bloques

Se denomina *proposición* a una expresión seguida de punto y coma ';'. Asimismo se denomina *bloque* o *proposición compuesta* a un conjunto de declaraciones y proposiciones agrupadas entre llaves '{','}'.

◆ **Ejemplo 1.22:**

```
{ /* Comienzo bloque*/  
float z; /*Declaración*/  
/*Proposiciones*/  
z=0.256;  
z=z+1;  
printf("%f\n",z)  
} /* Final bloque*/
```

◆

1.7.2 Ejecución condicional.

1.7.2.1 La instrucción if

La instrucción `if` presenta la siguiente sintaxis

```
if(expresión) proposición;
```

La proposición se ejecutará sólo si `expresión` tiene un valor no nulo, es decir es Verdadera. En caso contrario no se ejecutará. La proposición puede ser simple o compuesta.

◆ **Ejemplo 1.23**

```
if (debito>0) credito=0; /*If con proposición simple*/  
if(x<=3.0) { /*If con proposición compuesta*/  
    y=0.25*x ;  
    z=1.26*y;  
    printf("%f\n",y);  
}
```

◆

1.7.2.2 La instrucción *if - else*

La sintaxis de una instrucción *if* que incluye la sentencia *else* es:

```
if (expresión)
    proposición1;
else
    proposición2;
```

Si la expresión tiene un valor no nulo, es decir es Verdadera se ejecuta la proposición1, en caso contrario se ejecuta la proposición2. Tanto proposición1 como proposición2 pueden ser simples o compuestas.

◆ Ejemplo 1.24:

```
if (estado=='S')
    tasa=0.20*pago;
else
    tasa=0.14*pago;
if (debito>0) {
    prestamo=0;
    x=y+z;
}
else {
    x=y-z;
    d=0;
}
```

◆

1.7.2.3 La instrucción *else if*

La sintaxis de una instrucción *else if* es:

```
if (expresión1)
    proposición1;
else if (expresión2)
    proposición2;
...
else if (expresiónn-1)
    proposiciónn-1;
else
    proposiciónn;
```

1.7.3 Bucles

1.7.3.1 La instrucción *for*

La forma general de la instrucción `for` es:

```
for (inicialización; expresión; progresión)
    proposición;
```

donde `inicialización` se utiliza para inicializar algún parámetro (denominado índice) que controla la repetición del bucle, `expresión` representa una condición que debe ser satisfecha para que se continúe la ejecución del bucle, y `progresión` se utiliza para modificar el valor del parámetros inicialmente asignado por `inicialización`. `proposición` se ejecutará mientras `expresión` sea Verdadera. La `proposición` puede ser simple o compuesta.

◆ **Ejemplo 1.25:**

```
int digito;
for (digito=0; digito<=9; ++digito)
    printf("%d\n", digito);
```

◆

1.7.3.2 La instrucción *while*

La forma general de la instrucción `while` es:

```
while (expresión)
    proposición;
```

`proposición` se ejecutará mientras `expresión` sea Verdadera. La `proposición` puede ser simple o compuesta.

◆ **Ejemplo 1.26:**

```
int digito=0;
while (digito<=9){
    printf("%d\n", digito);
    ++digito;
}
```

◆

1.7.3.3 La instrucción **do - while**

La forma general de la instrucción **do - while** es:

```
do
{
    proposición;
} while (expresión);
```

proposición se ejecutará mientras **expresión** sea Verdadera. La primera vez siempre se ejecuta.

◆ **Ejemplo 1.27:**

```
int digito=0;
do{
    printf("%d\n",digito++);
}
while (digito<=9)
```

◆

1.7.4 Las instrucciones **break** y **continue**

La instrucción **break** se utiliza para terminar la ejecución de bucles o salir de una instrucción **switch**. Se puede utilizar dentro de una instrucción **while**, **do - while**, **for** o **switch**. La instrucción **break** se puede escribir sencillamente sin contener ninguna otra expresión o instrucción de la siguiente forma:

```
break;
```

La instrucción **continue** se utiliza para saltarse el resto de la pasada actual a través de un bucle. El bucle no termina cuando se encuentra una instrucción **continue**. Sencillamente no se ejecutan las instrucciones que se encuentran a continuación de **continue** y se salta directamente a la siguiente pasada a través del bucle.

La instrucción **continue** se puede incluir dentro de una instrucción **while**, **do - while** o **for**. Simplemente se escribe sin contener ninguna otra expresión o instrucción de la siguiente forma:

```
continue;
```

♦ Ejemplo 1.28:

```
int x=100;
while (x<=100){
    x=x-1
    if (x<0){
        printf("VALOR NEGATIVO DE X");
        break;
    }
    if (x==50){
        printf("REDUCCION AL 50%");
        continue;
    }
}
```

♦

1.7.5 La instrucción switch

La instrucción `switch` hace que se seleccione un grupo de instrucciones entre varios grupos disponibles. La selección se basa en el valor de una expresión que se incluye en la instrucción `switch`. Su sintaxis es:

```
switch(expresión)
{
    case exp_const1: proposiciones1;
    case exp_const2: proposiciones2;
    ...
    case exp_constn: proposicionesn;
    default: proposiciones;
}
```

♦ Ejemplo 1.29:

```
char eleccion ;
switch (eleccion=getchar()){
case 'r' :
case 'R' :
    printf("ROJO");
    break;
case 'b' :
case 'B' :
    printf("BLANCO");
    break;
```

```
case 'a' :
case 'A' :
    printf("AZUL");
    break;
default:
    printf("\n Entrada erronea");
}
```



1.8 FUNCIONES

Una *función* es un segmento de programa que realiza determinadas tareas bien definidas. Todo programa en C consta de una o mas funciones. Una de estas funciones se debe llamar `main`. La ejecución del programa siempre comenzará por las instrucciones contenidas en `main`.

Si un programa contiene varias funciones, sus definiciones pueden aparecer en cualquier orden, pero deben ser independientes unas de otras. Esto es, una definición de una función no puede estar incluida en otra.

Generalmente, una función procesará la información que le es pasada desde el punto del programa en que se accede a ella y devolverá un solo valor. La información se le pasa a la función mediante unos identificadores especiales llamados *argumentos* (también denominados *parámetros*) y es devuelta por medio de la instrucción `return`. Sin embargo, algunas funciones aceptan información pero no devuelven nada (por ejemplo, la función de biblioteca `printf`), mientras que otras funciones (por ejemplo, la función `scanf`) devuelven varios valores. Una función no puede devolver otra función, ni tampoco un array. La organización de un programa grande en funciones sencillas permite que el programa sea estructurado y más fácil de depurar y mantener.

1.8.1 Definición de una función

Según el estándar ANSI C una función se define de la siguiente forma:

```
tipo nombre_función (tipo1 arg1, tipo2 arg2,...,tipoN argN)
{
    variables_locales;
    proposiciones;
    return(expresión);
}
```

donde:

- `tipo` representa el tipo de datos del valor que devuelve la función. Por defecto si no se especifica nada, el tipo de una función es entero.
- `nombre` es el nombre de la función.
- `tipo1, tipo2, ..., tipoN` representan los tipos de datos de los argumentos `arg1, arg2, ..., argN`.

Los argumentos se denominan *argumentos formales*, ya que representan los nombres de los elementos que se transfieren a la función desde la parte del programa que hace la llamada.

El resto de la definición, es el *cuerpo de la función*, que contiene la definición de diferentes variables locales y diversas proposiciones.

Las variables locales y los parámetros o argumentos de la función se reservan en la pila de usuario del programa, por lo que al entrar en la función se reserva espacio para ellos, pero al salir de la función desaparecen. Este tipo de almacenamiento será estático y ocupará la zona de memoria reservada a las variables globales; además, esa variable existirá durante todo el tiempo de ejecución del programa.

◆ Ejemplo 1.30:

La siguiente función acepta dos cantidades enteras y determina el valor mayor, que se escribe a continuación. La función no devuelve información al punto de llamada

```
maximo (int x, int y)
{
    int z ;
    z=(x>=y) ?x :y;
    printf("\n\nValor máximo = %d", z);
    return;
}
```

◆

1.8.2 Prototipos de funciones y acceso a una función

Un *prototipo de una función* es la primera línea de una definición de función, añadiendo al final un punto y coma. Los prototipos de funciones normalmente se escriben

al comienzo del programa, delante de todas las funciones definidas por el programador (incluida `main`).

Los prototipos de funciones no son obligatorios en C. Sin embargo, son aconsejables, ya que facilitan la comprobación de errores.

La forma general de un prototipo de función es:

```
tipo nombre_función (tipo1 arg1, tipo2 arg2,...,tipoN argN);
```

Por otra parte, se puede *llamar o acceder a una función* especificando su nombre, seguido de una lista de argumentos encerrados entre paréntesis y separados por comas.

Los argumentos o parámetros que aparecen en la llamada a la función se denominan *argumentos reales*, en contraste con los argumentos formales que aparecen en la primera línea de la definición de la función.

◆ Ejemplo 1.31:

El siguiente programa completo en C, calcula el factorial de una cantidad entera positiva. El programa utiliza la función `factorial` además de la función `main`.

```
/*Calcular el Factorial de una cantidad entera*/
#include <stdio.h>
long int factorial(int n); /*Prototipo de la función factorial*/

main()
{
    int n;
    /* Leer la cantidad entera*/
    printf("\n n= ");
    scanf("%d", &n);
    /*Calcular y visualizar el factorial*/
    printf("\n n!= %ld", factorial(n));
}

/*Definición de la función factorial*/
long int factorial(int n)
{
    int i;
    long int prod=1;
    if (n>1)
        for (i=2, i<=n; ++i)
```

```
        prod *=i;
    return(prod);
}
```

Programa 1.2



1.8.3 Paso de argumentos a una función

Existen dos formas de pasar los argumentos a una función: *paso por valor* y *paso por referencia*.

Cuando se le pasa un valor simple a una función mediante un argumento real, se copia el valor del argumento real a la función. Por tanto, se puede modificar el valor del argumento formal dentro de la función, pero el valor del argumento real en la rutina que efectúa la llamada no cambiará. Este procedimiento para pasar el valor de un argumento a una función se denomina *paso por valor*.

A menudo los punteros son pasados a las funciones como argumentos. Esto permite que datos de la parte del programa en la que se llama a la función sean accedidos por la función, alterados dentro de ella y luego devueltos al programa de forma alterada. Este procedimiento para pasar el valor de un argumento a una función se denomina *paso por referencia*.

Cuando se pasa un argumento por referencia la dirección del dato es pasada a la función. El contenido de esta función puede ser accedido libremente, tanto dentro de la función como dentro de la rutina de la llamada. Además cualquier cambio que se realiza al dato (al contenido de la dirección) será reconocido en ambas, la función y la rutina de llamada. Así, el uso de punteros como argumentos de funciones permite que el dato sea alterado globalmente desde dentro de la función.

El paso de arrays como parámetros de funciones siempre se realiza por referencia. Las estructuras se pueden pasar por valor o por referencia. Si se pasa una estructura por valor, las modificaciones de sus campos serán locales mientras que si se pasan por referencia las modificaciones serán permanentes.

El hecho de que los parámetros pasados por valor sólo sufran modificaciones a nivel local, se debe a que el parámetro es una copia en la pila de usuario de la variable a la que se refiere. Así las modificaciones se hacen sobre la copia de la variable que hay en la pila y no sobre la propia variable.

◆ Ejemplo 1.32:

El siguiente programa ilustra la diferencia entre argumentos ordinarios, que son pasados por valor, y argumentos puntero, que son pasados por referencia.

```
#include <stdio.h>

void func1(int u, int v);      /*Prototipo de la función func1*/
void func2(int *pu, int *pv); /*Prototipo de la función func2*/

main()
{
    int u=1;
    int v=3;

    printf("\nAntes de la llamada a func1: u=%d v=%d", u, v);
    func1(u,v);
    printf("\nDespués de la llamada a func1: u=%d v=%d", u, v);

    printf("\nAntes de la llamada a func2: u=%d v=%d", u, v);
    func2(&u,&v);
    printf("\nDespués de la llamada a func2: u=%d v=%d", u, v);
}

void func1(int u, int v)
{
    u=0;
    v=0;
    printf("\nDentro de func1: u=%d v=%d", u, v);
    return;
}

void func2(int *pu, int *pv)
{
    *pu=0;
    *pv=0;
    printf("\nDentro de func2: *pu=%d *pv=%d", *pu, *pv);
    return;
}
```

Programa 1.3

La ejecución de este programa genera la siguiente salida:

```
Antes de la llamada a func1:  u=1  v=3
Dentro de func1:  u=0  v=0
Después de la llamada a func1:  u=1  v=3
Antes de la llamada a func2:  u=1  v=3
Dentro de func2:  u=0  v=0
Después de la llamada a func2:  u=0  v=0
```

◆

1.8.4 Punteros a funciones

Las funciones aunque no son variables tienen de igual modo una posición física en memoria, a la cual se le puede asignar un puntero. La dirección de memoria de una función es la entrada a la función, por tanto se puede usar un puntero para ejecutar una función y este puntero nos permitirá también pasar funciones como argumentos a otras funciones.

Un puntero a una función puede ser pasado como argumento a otra función. Esto permite que una función sea transferida a otra, como si la primera función fuera una variable. A la primera función se la denomina *función huésped* y a la segunda función se la denomina *función anfitriona*. De este modo la función huésped es pasada a la anfitriona, donde puede ser accedida.

Cuando una función anfitriona acepta el nombre de una función huésped como argumento, la declaración de argumento formal debe identificar el argumento como un puntero a la función huésped. En su forma más sencilla, la declaración de este argumento formal se puede escribir:

```
tipo_dato(*nombre_función) ( );
```

En la declaración anterior, `tipo_dato` es el tipo de dato de la cantidad devuelta por la función huésped, y `nombre_función` es el nombre de la función huésped.

La declaración de argumento formal también se puede escribir como:

```
tipo_dato (*nombre_función)(tipo1 arg1, tipo2 arg2,..., tipoN, argN);
```

O equivalentemente como:

```
tipo_dato(*nombre_función)(tipo1, tipo2,..., tipoN);
```

En la declaraciones anteriores, `tipo1`, `tipo2`, ..., `tipoN` se refiere a los tipos de datos de los argumentos asociados con la función huésped, y `arg1`, `arg2`, ..., `argN` son los nombres de los argumentos asociados con la función huésped.

◆ **Ejemplo 1.33:**

A continuación se muestra el boceto de un programa en C. Este programa consta de cuatro funciones `main`, `procesar`, `func1` y `func2`. Notar que `procesar` es una función anfitriona para `func1` y `func2`. Cada una de las tres funciones subordinadas devuelve un valor entero.

```
int procesar (int (*) (int,int)); /*Prototipo de función (anfitriona) */
int func1(int, int); /*Prototipo de función (huesped) */
int func2(int, int); /*Prototipo de función (huesped) */

main()
{
    int i,j;
    ...

    i=procesar(func1); /*Se pasa func1 a procesar,
                       devuelve un valor para i*/

    ...

    j=procesar(func2); /*Se pasa func2 a procesar,
                       devuelve un valor para j*/
}

/*Definición de la función anfitriona */
    procesar(int (*pf)(int,int))/*El argumento
    formal es un puntero a una función */
    {
        int a,b,c;
        ...

        c=(*pf)(a,b); /*Acceso a la función pasada a la
        función anfitriona, devuelve un valor para c*/

        ...
        return(c);
    }

/*Definición de función huésped */
func1(int a, int b)
{
    int c;
    c=a+b;
    return(c);
}
```

```
    }  
    /*Definición de función huésped */  
    func2(int x, int y)  
    {  
  
int z;  
z=x/y;  
return(z);  
  
    }
```

◆

1.8.5 Argumentos de la función main()

Es posible pasar argumentos a la función `main()`, de tal modo que los use como opciones iniciales en la ejecución del programa desde la línea de comandos. En ese caso la declaración de `main` es:

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

En C existen dos argumentos ya predefinidos para realizar esta operación:

- `int argc`. Contiene el número de argumentos de la línea de comandos y es un entero. El valor mínimo que toma es uno puesto que el nombre del programa cuenta como primer argumento.
- `char *argv[]`. Es un array de punteros a caracteres, es decir, un array de cadenas de caracteres. Cada elemento del array apunta a un argumento de la línea de ordenes. El contenido de `argv[0]` es el nombre del programa.

Los nombres `argc` y `argv` pueden ser sustituidos por otros nombres. Todos los argumentos de la línea de comandos son cadenas y deben ir separados por espacios en blanco. Si alguno de los argumentos de la línea de comandos se va a usar numéricamente en el programa, es obligación del programador pasar el argumento, que se considera una cadena, al tipo de datos numérico adecuado para la aplicación. Para realizar estas operaciones C tiene una extensa librería de funciones que permiten pasar datos de un formato a otro.

Cuando un programa usa argumentos la ausencia de uno de ellos en su invocación desde la línea de comandos puede provocar la ejecución incorrecta del programa. Luego es obligación del programador comprobar las condiciones iniciales de ejecución del programa.

♦ Ejemplo 1.34:

Supóngase el siguiente programa en C:

```
#include <stdio.h>
#include <stdlib.h>
void main (int argc, char *argv[])
{
    if (argc<2)
    {
        printf ("Error, falta clave de acceso\n");
        exit(0);
    }
    else
    {
        if (!strcmp(argv[1], "Azul") )
            printf("Acceso al programa...\n");
        else
        {
            printf ("Acceso denegado\n");
            exit(0);
        }
    }
}
```

Programa 1.4

Supuesto que el ejecutable de este programa lleva por nombre `clave`, en la línea de ordenes de la consola habría que llamarlo de la siguiente forma:

```
$ clave azul
```

Entonces aparecería el siguiente mensaje

```
Acceso al programa...
```

Por el contrario, si se llamase por ejemplo de la siguiente forma:

```
$ clave rojo
```

Entonces aparecería el siguiente mensaje

```
Acceso denegado
```

Finalmente, si se llamase por ejemplo de la siguiente forma:

```
$ clave
```

Entonces aparecería el siguiente mensaje

```
Error, falta clave de acceso
```



1.9 MACROS

La directiva del preprocesador `#define` también se emplea para definir *macros*, es decir, identificadores simples que son equivalentes a expresiones, a instrucciones completas o a grupos de instrucciones. En este sentido las macros se parecen a las funciones. No obstante, son definidas y tratadas de forma diferente que las funciones durante el proceso de compilación.

Las definiciones de macros están normalmente colocadas al principio de un archivo, antes de la definición de la primera función. El ámbito de definición de una macro va desde el punto de definición hasta el final del archivo donde ha sido definida. Sin embargo una macro definida en un archivo no es reconocida dentro de otro archivo.

Pueden ser definidas macros con varias líneas colocando una barra invertida (`\`) al final de cada línea excepto en la última. Esta característica permite que una sola macro (un identificador simple) represente una instrucción compuesta.

Una definición de *macro* puede incluir argumentos, que están encerrados entre paréntesis. El paréntesis izquierdo debe aparecer inmediatamente detrás del nombre de la macro, es decir, no pueden existir espacios entre el nombre de la macro y el paréntesis izquierdo.

◆ Ejemplo 1.35:

Supóngase el siguiente programa en C:

```
#include <stdio.h>
#define bucle(n) for (lineas=1; lineas<=n; lineas++){           \
    for(cont=1; cont<=n-lineas; cont++)                          \
        putchar(' ');                                           \
    for(cont=1; cont<=2*lineas-1; cont++)                        \
        putchar(' ');                                           \
    printf("\n");                                               \
}

main()
```

```

{
    int cont, lineas, n;

    printf("número de líneas= ");
    scanf("%d", &n);
    printf("\n");

    bucle(n);
}

```

Programa 1.5

Este programa contiene una macro `bucle(n)` de varias líneas, que representa a una instrucción compuesta. La instrucción compuesta consta de varios bucles `for` anidados. Notar la barra invertida (`\`) al final de la línea, excepto en la última.

Cuando el programa es compilado, la referencia a la macro es reemplazada por las instrucciones contenidas dentro de la definición de la macro. Así, el programa mostrado anteriormente se convierte en

```

main()
{
    int cont, lineas, n;

    printf("número de líneas= ");
    scanf("%d", &n);
    printf("\n");

    for (lineas=1; lineas<=n; lineas++){
        for(cont=1; cont<=n-lineas; cont++)
            putchar(' ');
        for(cont=1; cont<=2*lineas-1; cont++)
            putchar(' ');
        printf("\n");
    }
}

```

Programa 1.6



A veces las macros son usadas en lugar de funciones dentro de un programa. El uso de una macro en lugar de una función elimina el retraso asociado con la llamada a la función. Si el programa contiene muchas llamadas a funciones repetidas, el tiempo ahorrado por el uso de macros puede ser significativo.

Por otra parte, la sustitución de la macro se realizará en todas las referencias a la macro que aparezcan dentro de un programa. Así un programa que contenga varias referencias a la misma macro puede volverse excesivamente largo. Por tanto, se debe llegar a un compromiso entre la velocidad de ejecución y el tamaño del programa objeto compilado. El uso de la macro es más ventajoso en aplicaciones donde hay relativamente pocas llamadas a funciones pero la función es llamada repetidamente (por ejemplo, una función llamada dentro de un bucle).

1.10 ASIGNACION DINAMICA DE MEMORIA

A las variables globales se les asigna memoria en tiempo de compilación y a las variables locales se les reserva espacio en la pila. Hay aplicaciones en las que se necesita guardar un espacio de memoria variable en tiempo de ejecución, es lo que se conoce como *asignación dinámica de memoria*. Los punteros son la herramienta adecuada para el uso de la memoria dinámicamente.

Las funciones `malloc` y `free` definidas en la biblioteca `stdlib.h` permiten reservar y liberar memoria, respectivamente, de una forma dinámica. Estrechamente relacionado con estas funciones se encuentra el operador `sizeof` que devuelve el tamaño en bytes de su operando.. Su sintaxis es:

`sizeof(operando)`

◆ Ejemplo 1.36:

El siguiente programa:

```
#include <stdio.h>
main()
{
    int i;
    float x;
    double d;
    char c;

    printf("Entero: %d\n", sizeof(i));
    printf("Coma flotante: %d\n", sizeof(x));
    printf("Doble precisión: %d\n", sizeof(d));
    printf("Carácter: %d\n", sizeof(c));
}
```

Programa 1.7

genera la siguiente salida:

```
Entero: 2
Coma flotante: 4
Doble precisión: 8
Carácter: 1
```

Es decir el número de bytes asignado a cada variable.



Con el operador `sizeof` se asegura la portabilidad de un programa, al no depender la aplicación del tamaño del tipo de datos de la máquina que se vaya a usar.

Una vez analizado el operador `sizeof`, es posible describir una de las sintaxis más habituales para la función de biblioteca `malloc`:

```
ptr= (tipo*) malloc(N*sizeof(tipo));
```

donde `tipo` hace referencia a un tipo de datos (`int`, `float`, `char`, ...) y `N` indica el número de elementos del tipo `tipo` para los que se va reservar espacio (en bytes) en memoria. Si la función se ejecuta con éxito entonces en `ptr` se almacena un puntero a la zona de memoria reservada. En caso contrario (cuando no pueda reservar memoria), devolverá `NULL`. Es importante resaltar que el espacio reservado por `malloc` está sin inicializar.

Por su parte la sintaxis de la función de biblioteca `free` es:

```
free(ptr);
```

donde `free ptr` es un puntero previamente inicializado con `malloc`. Si la función se ejecuta correctamente libera la zona de memoria apuntada por `ptr`.

◆ Ejemplo 1.37:

El siguiente programa en primer lugar invoca a la función `malloc` para reservar un bloque de memoria cuyo tamaño en bytes es equivalente a 10 cantidades enteras. A continuación comprueba que la asignación se ha realizado correctamente. Finalmente invoca a la función `free` para liberar el espacio de memoria reservado con anterioridad.

```
#include <stdlib.h>
#include <stdio.h>
main()
{
    int *puntero;
```

```
puntero=(int *)malloc(10*sizeof(int));  
if(puntero==NULL)  
    printf("Error");  
else  
    printf("Memoria asignada");  
  
free(puntero);  
}
```

Programa 1.8



TEMA 2

CONSIDERACIONES GENERALES DEL SISTEMA OPERATIVO UNIX

2.1 INTRODUCCION

Un *programa* es un fichero ejecutable, y un *proceso* es una instancia de un programa en ejecución. Muchos procesos pueden ser ejecutados simultáneamente en el sistema UNIX, y varias instancias de un mismo programa pueden existir simultáneamente en el sistema.

El *sistema operativo* UNIX es un programa (a menudo denominado *núcleo*) que controla el hardware. Asimismo el núcleo administra (crea, destruye y controla) a los procesos y suministra varios servicios para ellos.

El núcleo reside en memoria secundaria en un archivo denominado típicamente `/vmmunix` o `/unix` (dependiendo de la distribución de UNIX). Cuando la computadora arranca, carga el núcleo desde el disco a memoria principal usando un procedimiento especial de arranque. El núcleo inicializa el sistema y configura el entorno para la ejecución de procesos. A continuación crea unos pocos procesos iniciales, los cuales a su vez crean otros procesos. Una vez cargado, el núcleo permanece en memoria principal hasta que el sistema se apaga.

Desde un punto de vista más general, el sistema operativo UNIX no incluye solo el núcleo, sino también es el anfitrión para otros programas y utilidades (como los intérpretes de comandos (shells), editores, compiladores, etc.) que se suelen distribuir conjuntamente con el núcleo. El núcleo, sin embargo, es especial por varios motivos. En primer lugar es el único programa indispensable sin el cual ningún otro podría ejecutarse. Y en segundo lugar define el interfaz de programación del sistema. Mientras que distintos

editores e intérpretes de comandos deben ejecutarse concurrentemente, solamente un único núcleo puede ser cargado a la vez.

Por un abuso del lenguaje, en muchas ocasiones cuando los usuarios utilizan el término “sistema UNIX” están englobando tanto al núcleo como a los programas y a las aplicaciones que le acompañan. En estos apuntes se usaran de forma frecuente los términos “sistema UNIX”, “núcleo” o “sistema” para hacer referencia exclusivamente al núcleo del sistema operativo UNIX.

Entre las principales características que han contribuido al éxito y popularidad de UNIX se encuentran:

- Está escrito en C, que es un lenguaje de programación de alto nivel, lo que hace que UNIX sea fácil de leer, entender, modificar y utilizar en diferentes computadoras.
- Posee un interfaz de usuario sencillo pero con muchas funcionalidades.
- Suministra primitivas que posibilitan el escribir programas complejos a partir de otros más sencillos.
- Utiliza un sistema de ficheros jerarquizado que posibilita su fácil mantenimiento y una eficiente implementación.
- Utiliza un formato consistente para los archivos, lo que posibilita que los programas de aplicación sean relativamente fáciles de escribir.
- Suministra una interfaz simple y consistente para los dispositivos periféricos.
- Es un sistema multiusuario y multiproceso; cada usuario puede ejecutar varios procesos simultáneamente.
- Oculta la arquitectura de la máquina al usuario, lo que simplifica la escritura de programas que puedan ser ejecutados sobre distintas implementaciones de hardware, es decir, son portables.

De acuerdo con las características anteriores, se puede afirmar que el sistema UNIX sigue una filosofía de simplicidad y consistencia.

Existen diferentes distribuciones de UNIX, como por ejemplo: *System V* de AT&T (American Telephone & Telegraph), BSD (Berkeley Software Distribution) de la Universidad de California en Berkeley, OSF/1 de Open Software Foundation, *SunOS* y *Solaris* de Sun Microsystems, etc. Además dentro de cada distribución existen diferentes versiones.

En este tema en primer lugar se detalla la historia del sistema UNIX, su lectura aclarará, sin duda, el porqué de la existencia de tantas distribuciones. A continuación se describe la arquitectura de UNIX. También se enumeran los principales servicios prestados por el núcleo. Después, se analizan los dos modos de ejecución en UNIX: modo usuario y modo núcleo. Asimismo se realiza una clasificación de los tipos de procesos en función del modo de ejecución. Además se realiza una primera introducción a dos de los principales eventos que son atendidos en modo núcleo: las interrupciones y las excepciones. A continuación se realiza una descripción de la estructura del sistema operativo UNIX. El tema finaliza con una introducción al interfaz de usuario para el sistema de ficheros.

2.2 HISTORIA DEL SISTEMA OPERATIVO UNIX

2.2.1 Orígenes

A finales de los años 60, los laboratorios BTL (Bell Telephone Laboratories) propiedad de la compañía AT&T estaban involucrados en un proyecto con la compañía General Electric y el MIT (Massachusetts Institute of Technology) para desarrollar un sistema operativo multiusuario denominado *Multics*. Cuando el proyecto Multics fue cancelado en marzo de 1969, uno de sus creadores, Ken Thompson, comenzó a programar el juego *Space Travel* que corría sobre la computadora PDP-7 (construida por DEC (Digital Equipment Corporation)).

Con el objetivo de facilitar el desarrollo de *Space Travel*, Thomson junto con Dennis Ritchie, comenzaron a desarrollar un sistema operativo para la PDP-7. Su primer componente fue un sencillo sistema de ficheros el cual evolucionó hasta convertirse en la primera versión de lo que ahora se conoce como sistema de ficheros System V (sfs5). A continuación le añadieron un subsistema de procesos, un *interprete de comandos* simple (el cual evolucionó hasta convertirse en el *Bourne shell*), y un pequeño conjunto de utilidades. Bautizaron a este nuevo sistema operativo con el nombre de UNIX (nombre que se obtiene de realizar un juego de palabras con Multics).

Al año siguiente Thompson, Ritchie y Joseph Ossanna portaron UNIX a una computadora PDP-11, y le añadieron varias utilidades para el procesamiento de textos, como el editor *ed*. Por otra parte, Thompson también desarrolló un nuevo lenguaje al que llamó B y lo utilizó para escribir diversas utilidades. Posteriormente, Ritchie lo mejoró hasta convertirlo en lo que denominó como lenguaje C, el cual era compilable y soportaba diferentes tipos y estructuras de datos. En 1973, UNIX fue escrito en lenguaje C, un hecho que resultó fundamental para el éxito de este sistema operativo.

Debido a las leyes antimonopolio vigentes en los Estados Unidos, AT&T concedió licencias gratuitas de uso de UNIX con fines educativos y de investigación a las universidades. Dentro del ámbito universitario UNIX rápidamente se extendió por todo el mundo. El uso de UNIX por la comunidad universitario aportó a AT&T ideas y sugerencias para ir mejorando su sistema operativo. Este espíritu de cooperación entre propietarios y usuarios (el cual se deterioró considerablemente una vez que UNIX tuvo éxito comercial) fue un factor clave para el rápido crecimiento y aumento de la popularidad de UNIX.

Las primeras versiones de UNIX únicamente corrían sobre la computadora PDP-11 y la computadora Interdata 8/32. Pronto UNIX fue portado a otras arquitecturas. Microsoft Corporation y Santa Cruz Operation (SCO) colaboraron para portar UNIX a la arquitectura Intel 8086, lo que resultó en *XENIX*, una de las primeras variantes comerciales de UNIX. En 1978 DEC introdujo la computadora VAX-11 de 32 bits e impulsó un grupo de trabajo para portar UNIX a la arquitectura VAX, la versión resultante (la primera para una máquina de 32 bits) se denominó UNIX/32V.

2.2.2 La distribución BSD de UNIX

La Universidad de Berkeley en California obtuvo una de las primeras licencias de UNIX en diciembre de 1974. Durante los años siguientes, un grupo de estudiantes entre los que se encontraban Bill Joy y Chuck Haley desarrollaron diversas utilidades para UNIX, como el editor *ex* (al cual le siguió el editor *vi*) y un compilador de Pascal. Incluyeron estas utilidades en un paquete denominado BSD y lo comercializaron en la primavera de 1978. Las versiones iniciales de BSD consistían únicamente en aplicaciones y utilidades, y no modificaban o redistribuían el sistema operativo. Una de las primeras contribuciones de Bill Joy fue el interprete de comandos C, que suministraba servicios tales como el control de tareas y un histórico de comandos, los cuales no se encontraban incluidos en el *interprete de comandos* Bourne.

En 1978 Berkeley adquirió una computadora VAX-11/780 y el UNIX/32V. La VAX tenía una arquitectura de 32 bits, lo que permitía un espacio de direccionamiento de 4 Gigabytes, pero solo una memoria física de 2 Megabytes. Ozalp Babaoglu diseñó para VAX un sistema de memoria virtual basado en páginas y lo incorporó dentro de UNIX. El resultado fue la versión 3.0 de BSD (BSD3.0) a finales de 1979, que fue la primera versión del sistema operativo UNIX generada por Berkeley. A ésta le siguieron las versiones 4.x (BSD4.x): BSD4.0 en 1980, BSD4.1 en 1981, BSD4.2 en 1983, BSD4.3 en 1986 y BSD4.4 en 1993.

El equipo de Berkeley fue responsable de importantes contribuciones técnicas a UNIX. Además de la memoria virtual y la incorporación del TCP/IP, BSD UNIX introdujo el sistema de ficheros rápido (FFS), una implementación más fiable del mecanismo de señales, y el servicio de conectores (sockets).

Con el objetivo de comercializar BSD4.4 se creó la compañía BSDI (Berkeley Software Design, Inc). Puesto que la mayoría del código fuente de UNIX había sido sustituido con nuevo código desarrollado en Berkeley, BSDI afirmaba que el código fuente de su distribución era completamente libre de las licencias de AT&T. Así, AT&T inició una batalla judicial contra BSDI, alegando una vulneración del copyright, incumplimiento de contrato y apropiación de secretos comerciales.

2.2.3 La distribución System V de UNIX

De forma paralela al desarrollo de BSD, AT&T sacó al mercado la distribución de UNIX System III en 1982 y la distribución System V en 1983. De esta última distribución aparecieron la versión 2 (SVR2) en 1984, la versión 3 (SVR3) en 1987 y la versión 4 (SVR4) en 1989.

La distribución System V de UNIX incluía bastantes características y servicios nuevos. Su implementación de la memoria virtual, denominada *arquitectura de regiones*, era bastante diferente de la de la distribución BSD. SVR3 introdujo nuevos mecanismos de comunicación entre procesos (semáforos, memoria compartida y colas de mensajes), ficheros remotos compartidos, librerías compartidas, y los *streams* (para los drivers de dispositivos y para los protocolos de red).

2.2.4 Comercialización de UNIX

La creciente popularidad de UNIX atrajo el interés de distintas empresas fabricantes de computadoras que se apresuraron a comercializar sus propias distribuciones de UNIX,

las cuales era una adaptación para el hardware de sus computadoras de las distribuciones de AT&T o de Berkeley, mejoradas en algunos aspectos. En 1977 Interactive Systems fue el primer vendedor comercial de UNIX. Su primera distribución de UNIX se llamó IS/1 y corría en las computadoras PDP-11.

En 1982 Bill Joy dejó Berkely para fundar Sun Microsystems, la cual comercializó una variante de la versión 4.2 de la distribución BSD a la que de llamó *SunOS* (y más tarde una variante de SVR4 llamada *Solaris*). Microsoft y SCO sacaron la distribución XENIX. Posteriormente, SCO portó SVR3 a la arquitectura 386 y sacó al mercado la distribución SCO UNIX. En década de los 80 existían numerosas ofertas comerciales, incluyendo *AIX* de IBM, *HP-UX* de Hewlett-Packard Corporation, y *ULTRIX* (seguido por *DEC OSF/1*, posteriormente rebautizado como *Digital UNIX*) de DEC.

Todas estas variantes comerciales introdujeron bastantes características nuevas, algunas de las cuales fueron incorporadas sucesivamente en las nuevas versiones. SunOS introdujo el sistema de ficheros en red NFS (Network File System), el interfaz *nodo-v/sfv* para soportar múltiples tipos de sistemas de ficheros, y una nueva arquitectura de memoria virtual que fue adoptada por SVR4. Asimismo sacó ULTRIX uno de los primeras distribuciones UNIX para multiprocesador.

2.2.5 Estándares para compatibilidad en UNIX

La proliferación de variantes de UNIX condujo a varios problemas de compatibilidad. Mientras que todas las variantes “parecían como UNIX” desde lejos, diferían en bastantes aspectos importantes. En un principio, la industria estaba dividida por las diferencias entre la distribución System V de AT&T (el UNIX oficial) y la distribución BSD de Berkeley. La introducción de variantes comerciales empeoró la situación.

System V y BSD4.x difieren en muchos aspectos: sistemas de ficheros físicos, entorno de trabajo en red, arquitecturas de memoria virtual, etc. Algunas de estas diferencias se limitan al diseño e implementación del núcleo, pero otras se manifiestan en la programación a nivel del interfaz entre los programas y el sistema operativo. No es posible escribir una aplicación compleja que pueda ejecutarse sin ser modificada en sistemas System V y en sistemas BSD.

Las variantes comerciales derivaban o del System V o del BSD, y después eran mejoradas en algunos aspectos. Estas características adicionales eran a menudo inherentemente no portables. Como resultado, los programadores de aplicaciones

estaban frecuentemente confundidos y consumían mucho tiempo en asegurarse de que sus programas funcionaban en casi todas las variantes de UNIX.

Por lo tanto, se hacía necesario disponer de un conjunto de interfaces estándares. Los estándares resultantes fueron casi tan numerosas y diversas como las versiones de UNIX. Finalmente, la mayoría de los vendedores se pusieron de acuerdo en unos pocos estándares:

- SVID (*System V Interface Definition*) de AT&T. SVID es esencialmente una especificación detallada de la interfaz de programación del System V.
- POSIX (Portable Operating System based on UNIX) del IEEE (Institute of Electrical and Electronic Engineers). En 1986 el IEEE nombró un comité para publicar un estándar formal para los entornos de los sistemas operativos. Su estándar se denominó POSIX y era una amalgama de partes del núcleo de SVR3 y del UNIX 4.3BSD. Este estándar ha tenido bastante aceptación en parte porque no se alinea con una única variante de UNIX.
- La *guía de portabilidad* del consorcio internacional de fabricantes de computadores X/Open. Se formó en 1984, no para producir nuevos estándares, sino para desarrollar un entorno abierto de aplicaciones comunes basado de hecho en los estándares existentes. Su XPG es un borrador del estándar POSIX, pero va más allá al abordar muchas áreas adicionales como la internacionalización, interfaces de ventanas y administración de datos.

Cada estándar se ocupaba del interfaz entre los programadores y el sistema operativo y no de como el sistema implementaba dicha interfaz. Definía un conjunto de funciones y su semántica detallada. Los sistemas que siguen estos estándares deben cumplir estas especificaciones, pero pueden implementar las funciones o bien en el núcleo o bien en las librerías a nivel de usuario.

Los estándares tratan con un subconjunto de las funciones suministradas por la mayoría de los sistemas UNIX. Teóricamente, si los programadores se restringen a usar este subconjunto, la aplicación resultante debería ser portable a cualquier sistema que siga el estándar.

2.2.6 Las organizaciones OSF y UI

En 1987 AT&T, tuvo que hacer frente a una protesta pública contra su política de licencias, al anunciar la compra del 20% de Sun Microsystems. AT&T y Sun acordaron colaborar en el desarrollo de la versión 4 del System V. Así AT&T anunció que Sun recibiría un trato preferente, y Sun anunció que a diferencia del SunOS, el cual estaba basado en BSD4, sus próximo sistema operativo estaría basado en SVR4.

Este anuncio produjo una rápida reacción en los otros vendedores de UNIX, quienes temían que esto diera a Sun una injusta ventaja. En respuesta, un grupo de grandes compañías, como DEC, IBM y HP, anunciaron en 1988 la creación de OSF (*Open Software Foundation*) que estaba financiada por sus compañías fundadoras, y se comprometieron a desarrollar un sistema operativo libre de las licencias de AT&T.

En respuesta, AT&T y Sun, junto con otros vendedores de sistemas basados en el System V, formaron rápidamente una organización llamada UI (UNIX International). UI estaba dedicada a la comercialización del SVR4 y a definir las futuras mejoras del UNIX System V.

En 1989 OSF sacó un interfaz de usuario gráfico llamado *Motif*, que fue muy bien recibido. Poco después, sacó las primeras versiones de su sistema operativo OSF/1, que poseía muchas ventajas de las que carecía SVR4, tales como un soporte completo para multiprogramación, carga dinámica, y administración de volúmenes lógicos. El plan de los miembros fundadores era desarrollar un sistema operativo comercial basado en OSF/1.

En 1990 UI sacó el UNIX System V Road Map, el cual perfilaba las futuras mejoras del desarrollo de UNIX.

OSF y UI comenzaron como grandes rivales, pero pronto se unieron para hacer frente a una amenaza común. A principios de los 90 la ralentización de la economía y la aparición de del sistema operativo Windows de Microsoft, amenazaban el crecimiento e incluso la supervivencia de UNIX. UI se fue del negocio en 1993 y OSF abandonó muchos de sus ambiciosos planes. DEC OSF/1 fue el principal sistema basado en OSF/1. Con el tiempo, DEC eliminó muchas de las dependencias del OSF/1 de su sistema operativo, y en 1995, cambió su nombre por el de *Digital UNIX*.

2.2.7 La distribución SVR4 y más allá

AT&T y Sun desarrollaron conjuntamente SVR4, que salió al mercado en 1989. SVR4 integraba características del SVR3, BSD4, SunOS y XENIX. También incluía nuevas funcionalidades como las clases de planificación en tiempo real, el interprete de comandos *Korn*, mejoras del subsistema de *streams*, etc.

Al año siguiente, AT&T formó una compañía de software llamada USL (UNIX Systems Laboratories) para desarrollar y vender UNIX. En 1991 Novell, Inc, creador del sistema operativo *Netware*, para computadoras personales en red, compró parte de USL y creó una empresa filial llamada *Univel*. Univel se dedicó a desarrollar una versión para computadoras personales del SVR4 integrado con *Netware*. Este sistema operativo conocido como UNIXWare, salió al mercado a finales de 1992.

En 1993 AT&T vendió el resto de sus acciones a Novell. Al cabo de un año, Novell sacó la marca registrada UNIX. En 1994, Sun Microsystems compró los derechos del código del SVR4 a Novell. Al sistema Sun basado en SVR4 se le denominó *Solaris*, siendo su versión 10 la más reciente.

2.3 ARQUITECTURA DEL SISTEMA OPERATIVO UNIX

En la Figura 2.1 se representa un posible diagrama de la arquitectura del sistema operativo UNIX. En el mismo se observa la existencia de 4 niveles o capas.

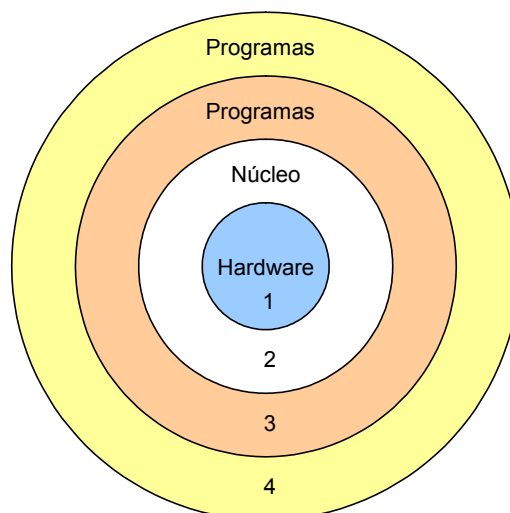


Figura 2.1: Arquitectura del sistema operativo UNIX

En el nivel más interno o primer nivel, se encuentra el *hardware* de la computadora cuyos recursos se desean gestionar.

En el segundo nivel, directamente en contacto con el hardware, se encuentra el *núcleo del sistema*, también llamado únicamente *núcleo (kernel)*. Este núcleo está escrito en lenguaje C en su mayor parte, aunque coexistiendo con lenguaje ensamblador. El núcleo suministra los servicios que utilizan todos los programas de aplicación del sistema UNIX.

En el tercer nivel, en contacto con el núcleo, se encuentran los programas estándar de cualquier sistema UNIX (intérpretes de comandos, editores, etc.) y programas ejecutables generados por el usuario.

Un programa ubicado en este nivel puede interactuar con el núcleo mediante el uso de las *llamadas al sistema*, las cuales dan instrucciones al núcleo para que realice (en el nombre del programa que las invoca) diferentes operaciones con el hardware. Además, las llamadas al sistema permiten un intercambio de datos entre el núcleo y el programa.

En definitiva, las *llamadas al sistema* son el mecanismo que los programas utilizan para solicitar al núcleo el uso de los recursos del computador (hardware). Habitualmente las llamadas al sistema se identifican como un conjunto perfectamente definido de funciones.

En el cuarto nivel, se sitúan las aplicaciones que se sirven de otros programas ya creados ubicados en el nivel inferior para llevar a cabo su función. Estas aplicaciones no se comunican directamente con el núcleo. Por ejemplo una aplicación situada en este cuarto nivel sería el compilador de C `cc` que invoca de forma secuencial a los programas `cpp`, `comp`, `as` y `ld`. situados en el tercer nivel.

La jerarquía de programas no tiene porqué verse limitada a cuatro niveles. El usuario puede crear tantos niveles como necesite. Además, puede haber también programas que se apoyen en diferentes niveles y que se comuniquen con el núcleo por un lado, y con otros programas ya existentes, por otro.

La existencia del núcleo posibilita que los programas de los niveles superiores puedan ser escritos sin realizar ninguna suposición sobre el hardware de la computadora. A su vez esto facilita su portabilidad entre diferentes tipos de computadoras (siempre que tengan instalado UNIX).

2.4 SERVICIOS REALIZADOS POR EL NUCLEO

Los principales servicios realizados por el núcleo son:

- *Control de la ejecución de los procesos* posibilitando su creación, terminación o suspensión, y comunicación.
- *Planificación de los procesos para su ejecución en la CPU.* En UNIX los procesos comparten el uso de la CPU por ello el núcleo debe velar porque la utilización de la CPU por parte de todos los procesos se realice de una forma justa.
- *Asignación de la memoria principal.* La memoria principal de una computadora es un recurso finito y muy valioso. Si el sistema posee en un cierto momento poca memoria principal libre, el núcleo liberará memoria escribiendo uno o varios procesos temporalmente en memoria secundaria (en un espacio predefinido denominado *dispositivo de intercambio*). Si el núcleo escribe un proceso entero en el dispositivo de intercambio, se dice que el sistema de gestión de memoria sigue una política de intercambio. Mientras que si escribe páginas de memoria asociadas al proceso en el dispositivo de intercambio, se dice que el sistema de gestión de memoria sigue una política de demanda de páginas.
- *Protección del espacio de direcciones de un proceso en ejecución.* El núcleo protege el espacio de direcciones de un proceso de intromisiones externas por parte de otros procesos. No obstante, bajo ciertas condiciones un proceso puede compartir porciones de su espacio de direcciones con otros procesos.
- *Asignación de memoria secundaria para almacenamiento y recuperación eficiente de los datos de usuario.* El núcleo asigna memoria secundaria para los ficheros de usuario, reclama el espacio no utilizado, estructura el sistema de ficheros de una forma entendible, y protege a los ficheros de usuario de accesos ilegales.
- *Regulación del acceso de los procesos a los dispositivos periféricos* tales como terminales, unidades de disco, dispositivos en red, etc.
- Administración de archivos y dispositivos
- Tratamiento de las interrupciones y excepciones

2.5 MODOS DE EJECUCION

2.5.1 Modo usuario y modo núcleo

El núcleo reside permanentemente en memoria principal así como el proceso actualmente en ejecución también denominado *proceso actual* (o partes del mismo, por lo menos). Cuando se compila un programa, el compilador genera un conjunto de direcciones de memoria asociadas al programa que representan las direcciones de las variables y de las estructuras de datos, o las direcciones de instrucciones como por ejemplo funciones. El compilador genera las direcciones para una máquina virtual considerando que ningún otro programa será ejecutado simultáneamente en la máquina física.

Cuando un programa se ejecuta en la máquina, el núcleo le asigna espacio en memoria principal, pero las direcciones virtuales generadas por el compilador no necesitan ser idénticas a las direcciones físicas que ocupan en la máquina. El núcleo se coordina con el hardware de la máquina para traducir las direcciones virtuales a direcciones físicas. Esta traducción depende de las capacidades del hardware de la máquina, y en consecuencia las partes del sistema UNIX que se ocupan de la misma son dependientes de la máquina.

Con el objetivo de poder implementar una protección eficiente del espacio de direcciones de memoria asociado al núcleo y de los espacios de direcciones asociados a cada proceso, la ejecución de los procesos en un sistema UNIX está dividida en dos modos de ejecución: un modo de mayor privilegio denominado *modo núcleo o supervisor* y otro modo de menor privilegio denominado *modo usuario*.

Un *proceso ejecutándose en modo usuario* sólo puede acceder a unas partes de su propio espacio de direcciones (código, datos y pila). Sin embargo, no puede acceder a otras partes de su propio espacio de direcciones, como aquellas reservadas para estructuras de datos asociadas al proceso usadas por el núcleo. Tampoco puede acceder al espacio de direcciones de otros procesos o del mismo núcleo. De esta forma se evita una posible corrupción de los mismos.

Por otra parte, un *proceso ejecutándose en modo núcleo* puede acceder a su propio espacio de direcciones al completo y al espacio de direcciones del núcleo, pero no puede acceder al espacio de direcciones de otros procesos. Debe quedar claro que cuando se dice que un proceso se está ejecutando en modo núcleo, en realidad el que se está

ejecutando es el núcleo pero en el nombre del proceso. Por ejemplo, cuando un proceso en modo usuario realiza una llamada al sistema está pidiendo al núcleo que realice en su nombre determinadas operaciones con el hardware de la máquina.

Entre los principales casos que producen que un proceso ejecutándose en modo usuario pase a ejecutarse en modo núcleo se encuentran: las llamadas al sistema, las interrupciones (*hardware* o *software*) y las excepciones.

2.5.2 Tipos de procesos

Los procesos en el sistema UNIX pueden ser de tres tipos: procesos de usuario, procesos demonio y procesos del núcleo o del sistema.

- Los *procesos de usuario* son aquellos procesos asociados a un determinado usuario. Se ejecutan en modo usuario excepto cuando realizan llamadas al sistema para acceder a los recursos del sistema, que pasan a ser ejecutados en modo núcleo.
- Los *procesos demonio* no están asociados a ningún usuario. Al igual que los procesos de usuario, son ejecutados en modo usuario excepto cuando realizan llamadas al sistema que pasan a ser ejecutados en modo núcleo. Los procesos demonio realizan tareas periódicas relacionadas con la administración del sistema, como por ejemplo: la administración y control de redes, la ejecución de actividades dependientes del tiempo, la administración de trabajos en las impresoras en línea, etc.
- Los *procesos del núcleo* no están asociados a ningún usuario. Se ejecutan exclusivamente en modo núcleo. Son similares a los procesos demonio en el sentido de que realizan tareas de administración del sistema, como por ejemplo, el intercambio de procesos (proceso intercambiador) o de páginas (proceso ladrón de páginas) a memoria secundaria. Su principal ventaja respecto a los procesos demonio es que poseen un mayor control sobre sus prioridades de planificación puesto que su código es parte del núcleo. Por ello pueden acceder directamente a los algoritmos y estructuras de datos del núcleo sin hacer uso de las llamadas al sistema, en consecuencia son extremadamente potentes. Sin embargo no son tan flexibles como los procesos demonio, ya que para modificarlos se debe de recompilar el núcleo.

2.5.3 Interrupciones y Excepciones

El sistema UNIX permite al reloj del sistema, a los periféricos de E/S o a los terminales interrumpir a la CPU mientras se está ejecutando un proceso. Estos dispositivos usan el mecanismo de interrupciones para notificar al núcleo que se ha completado una operación de E/S o que se ha producido un cambio en su estado. Así, las *interrupciones hardware* son eventos asíncronos que ocurren entre la ejecución de dos instrucciones de un proceso y pueden estar asociadas a eventos totalmente ajenos a la ejecución del proceso actualmente en ejecución.

Las *interrupciones software o traps*, se producen al ejecutar ciertas instrucciones especiales y son tratadas de forma síncrona. Son utilizadas, por ejemplo, en las llamadas al sistema, en los cambios de contexto, en tareas de baja prioridad de planificación asociadas con el reloj del sistema, etc.

Las *excepciones* hacen referencia a la aparición de eventos síncronos inesperados, típicamente errores, causados por la ejecución de un proceso, como por ejemplo, el acceso a una dirección de memoria ilegal, el rebose de la pila de usuario, el intento de ejecución de instrucciones privilegiadas, la realización de una división por cero, etc. Las excepciones se producen durante el transcurso de la ejecución de una instrucción.

Tanto las interrupciones (hardware o software) como las excepciones son tratadas en modo núcleo por determinadas rutinas del núcleo, no por procesos del núcleo.

Puesto que existen diferentes eventos que pueden causar una interrupción, puede suceder que llegue una petición de interrupción mientras otra interrupción está siendo atendida. Por lo tanto es necesaria asignar a cada tipo de interrupción un determinado *nivel de prioridad de interrupción (npi)* o *nivel de ejecución del procesador*. De tal forma que las interrupciones de mayor *npi* tenga preferencia sobre las de menor *npi*. Por ejemplo, una interrupción del reloj de la máquina debe tener preferencia sobre una interrupción de un dispositivo de red, puesto que ésta última requerirá un mayor tiempo de uso de la CPU, varios tics de reloj, para ser atendida.

El *npi* se almacena en un campo del *registro de estado del procesador*. Las computadoras típicamente poseen un conjunto de instrucciones privilegiadas para comparar y configurar el *npi* a un determinado valor. Además el núcleo también dispone de rutinas, típicamente implementadas como macros por motivos de eficiencia, para explícitamente comprobar o configurar el *npi*.

Cuando el núcleo se encuentra realizando ciertas actividades críticas para el correcto funcionamiento del sistema no debe atender ciertos tipos de interrupciones para evitar la corrupción de determinadas estructuras de datos. Para ello, fija el *npi* a un determinado valor. Así las interrupciones del mismo nivel o de niveles inferiores quedan *enmascaradas* o *bloqueadas*, por lo que sólo se atenderán las interrupciones de los niveles superiores.

El número de *niveles de prioridad de interrupción* permitidos depende de cada distribución de UNIX. Usualmente, el menor *npi* es 0¹.

♦ Ejemplo 2.1:

En la Figura 2.2 se muestra un ejemplo de un conjunto de niveles de prioridad de interrupción o niveles de ejecución del procesador. Si el núcleo configura el *npi* al valor asociado a los discos (se dice que se han enmascarado las interrupciones de los discos), entonces se estarán bloqueando todas las interrupciones excepto las interrupciones del reloj y las interrupciones asociadas a los errores de la máquina.

Por otro lado, si el núcleo configura el *npi* al valor asociado a las interrupciones software (se dice que se han enmascarado las interrupciones software), entonces todas las demás tipos de interrupciones estarán permitidas.

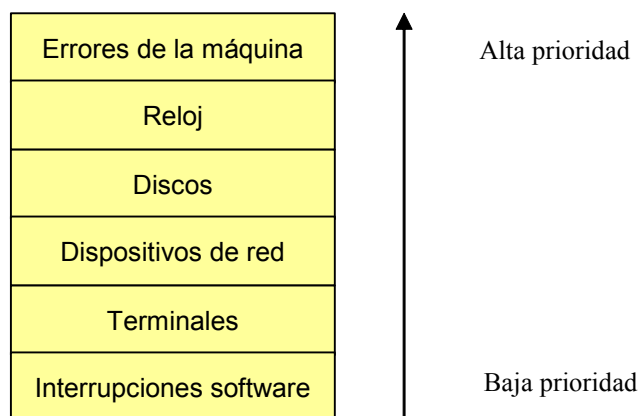


Figura 2.2: Niveles de prioridad de interrupción típicos

♦

¹ En algunas distribuciones el criterio es justamente el contrario, es decir, 0 está asociado al máximo *npi*.

2.6 ESTRUCTURA DEL SISTEMA OPERATIVO UNIX

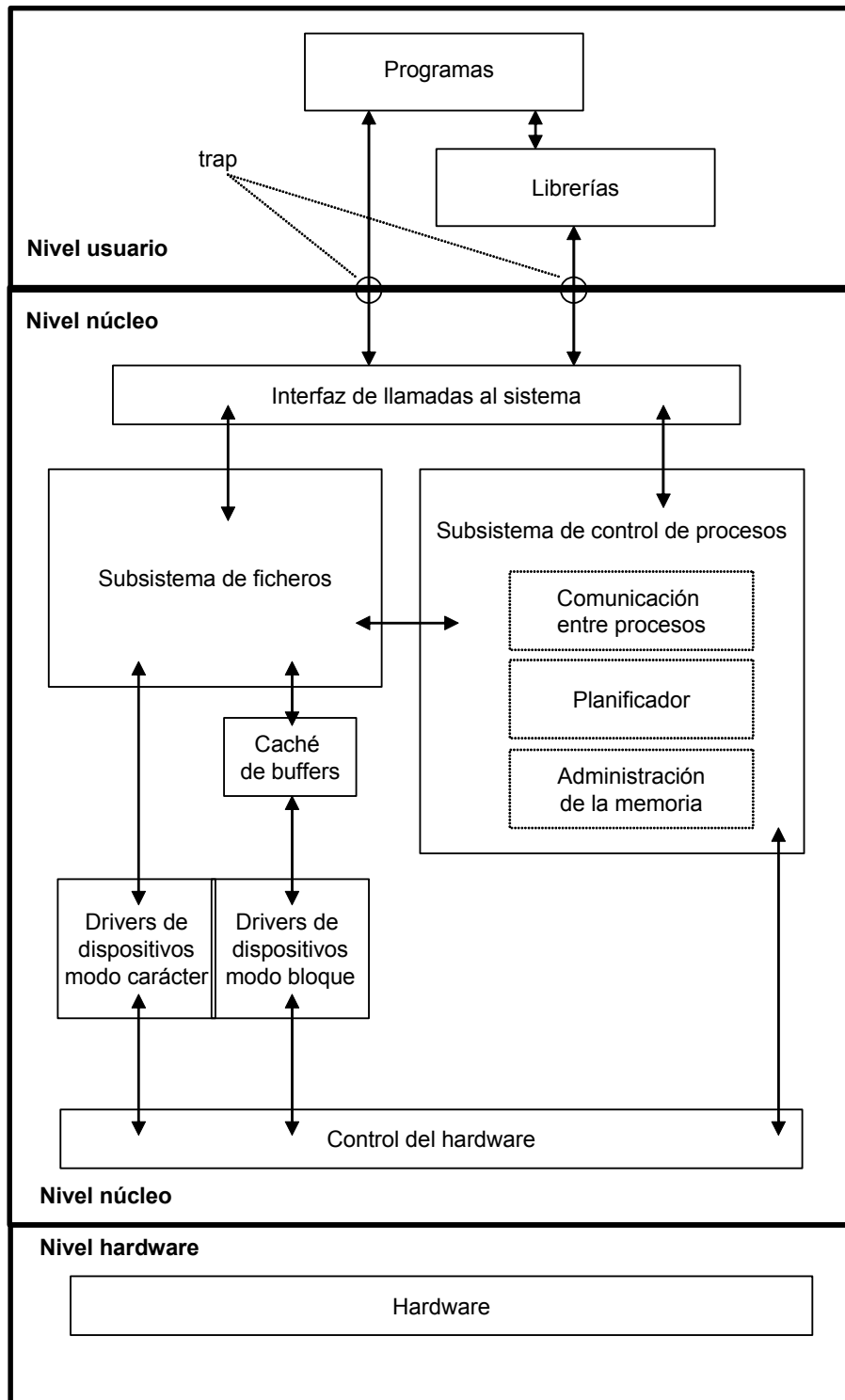


Figura 2.3: Estructura del sistema operativo UNIX

En la Figura 2.3 se muestra un posible esquema de la estructura del sistema operativo UNIX, se distinguen tres niveles: nivel de usuario, nivel del núcleo y nivel del hardware. En las siguientes secciones se describe cada uno de estos niveles.

2.6.1 Nivel de usuario

En el nivel de usuario se encuentran los programas de usuario y los programas demonio. Estos programas interaccionan con el núcleo haciendo uso de las *llamadas al sistema*. Los programas pueden invocar a las llamadas al sistema de dos formas:

- *Mediante el uso de librerías de llamadas al sistema.* Las llamadas al sistema se realizan de forma semejante a como se realizan las llamadas a cualquier función de un programa escrito en lenguaje C. Existen librerías de llamadas al sistema que trasladan estas llamadas a las funciones primitivas necesarias que permiten acceder al núcleo. Estas librerías se enlazan por defecto con el código de los programas en tiempo de compilación, formando así parte del fichero objeto asociado al programa.
- *Forma directa.* Los programas escritos en lenguaje ensamblador pueden invocar a las llamadas al sistema de forma directa sin usar una librería de llamadas al sistema

Al invocar un proceso a una llamada al sistema se ejecuta una instrucción especial que es una *interrupción software o trap* que provoca la conmutación hardware al modo supervisor.

2.6.2 Nivel del núcleo

En este nivel se encuentran el *subsistema de ficheros* y el *subsistema de control de procesos*, que son los dos módulos más importantes del núcleo. El esquema de la Figura 2.3 ofrece solamente una visión lógica útil del núcleo, en la práctica el comportamiento real del núcleo se desvía del modelo propuesto, puesto que algunos de los módulos interactúan con las operaciones internas de otros módulos.

El interfaz de llamadas al sistema representa la frontera entre los programas de usuario y el núcleo. Las llamadas al sistema pueden interactuar tanto con el *subsistema de ficheros* como con el *subsistema de control de procesos*. Asimismo el núcleo está en contacto con el hardware de la máquina a través de su *módulo de control del hardware*.

2.6.2.1 Subsistema de ficheros

El *subsistema de ficheros* se encarga de realizar todas las tareas del sistema asociadas a los ficheros: reserva espacio en memoria principal para las copias de los

ficheros, administra el espacio libre del sistema de ficheros, controla el acceso a los ficheros, regula el intercambio de datos (lectura o escritura) entre los ficheros y los usuarios, etc.

Los *procesos* interactúan con el *subsistema de ficheros* mediante un interfaz bien definido, que encapsula la visión que tiene el usuario del sistema de ficheros. Además este interfaz especifica el comportamiento y la semántica de todas las llamadas al sistema pertinentes tales como: `open` (abre un fichero para la lectura o escritura), `close` (cierra un fichero), `read` (lee en un fichero), `write` (escribe en un fichero), `stat` (devuelve los atributos de un fichero), `chown` (cambia el propietario de un fichero), `chmod` (cambia los permisos de acceso al fichero), etc. El interfaz exporta al usuario un pequeño número de abstracciones tales como: *ficheros*, *directorios*, *descriptores de ficheros* y *sistemas de ficheros*.

Asimismo dentro del subsistema de ficheros se encuentra el *interfaz nodo-v/sfv* que permite a UNIX soportar diferentes tipos de sistemas de ficheros tanto UNIX (sf5s, FFS, etc) como DOS (fat). Este interfaz será objeto de estudio en el Tema 8.

En UNIX hay diferentes tipos de ficheros: ordinarios (también denominados regulares o de datos), directorios, enlaces simbólicos, tuberías y ficheros de dispositivos (también denominados ficheros especiales), etc.

Los *ficheros ordinarios* contienen bytes de datos organizados como un array lineal. Los *directorios* son ficheros que permiten dar una estructura jerárquica a los sistemas de ficheros de UNIX. Los *enlaces simbólicos* son ficheros que contienen el nombre de otro fichero. Las *tuberías* (sin nombre y ficheros FIFO²) son un mecanismo de comunicación que permite la transmisión de un flujo de datos no estructurados de tamaño fijo. Los *ficheros de dispositivos* permiten a los procesos comunicarse con los dispositivos periféricos (discos, CD-ROM, cintas, impresoras, terminales, redes, etc.)

Existen dos tipos de ficheros de dispositivos: *dispositivos modo bloque* y *dispositivos modo carácter*, cada uno de ellos tiene asignado un tipo de fichero de dispositivo.

En los *dispositivos modo bloque*, el dispositivo contiene un array de bloques de tamaño fijo (generalmente un múltiplo de 512 bytes). La transferencia de datos entre el dispositivo y el núcleo, o viceversa, se realiza a través de un espacio en la memoria principal denominado *caché de buffers de bloques* que es gestionado por el núcleo. Esta

² FIFO es el acrónimo derivado del término inglés “First In First Out”.

caché está implementada por software y no debe confundirse con las memorias caché hardware que poseen muchas computadoras. El uso de esta caché permite regular el flujo de datos lográndose así un incremento en la velocidad de transferencia de los datos. Ejemplos típicos de dispositivos modo bloque son los discos y las unidades de cinta.

Los dispositivos modo carácter son aquellos dispositivos que no utilizan un espacio intermedio de almacenamiento en memoria principal para regular el flujo de datos con el núcleo. En consecuencia las transferencias de datos se van a realizar a menor velocidad. Ejemplos típicos de dispositivos modo carácter son los terminales serie y las impresoras en línea. En los ficheros de dispositivos modo carácter la información no se organiza según una estructura concreta y es vista por el núcleo, o por el usuario, como una secuencia lineal de bytes.

Un mismo dispositivo físico puede soportar los dos modos de acceso: bloque y carácter, y de hecho esto suele ser habitual en el caso de los discos.

Los módulos del núcleo que gestionan la comunicación con los dispositivos se denominan *manejadores o drivers de dispositivos*. Lo normal es que cada dispositivo tenga su manejador propio, aunque puede haber manejadores que controlen a toda una familia de dispositivos con características comunes (por ejemplo, el manejador que controla los terminales).

2.6.2.2 Subsistema de control de procesos

El *subsistema de control de procesos* se encarga, entre otras, de las siguientes tareas: sincronización de procesos, comunicación entre procesos, administración de la memoria principal y planificación de procesos.

El subsistema de ficheros y el subsistema de control de procesos interactúan cuando se carga un fichero en memoria principal para su ejecución. El subsistema de procesos es el encargado de cargar los ficheros ejecutables en la memoria principal antes de ejecutarlos.

Algunas de las llamadas del sistema para control de procesos son: `fork` (crea un nuevo proceso), `exec` (ejecuta un programa), `exit` (finaliza la ejecución de un proceso), `wait` (sincroniza la ejecución de un proceso con la terminación de uno de sus procesos hijos), `signal` (controla la respuesta de un proceso ante un determinado tipo de señal), etc.

El subsistema de control de procesos esta formado por tres módulos: módulo de administración de memoria, módulo de planificación y módulo de comunicación entre procesos.

El *módulo de administración o gestión de memoria* controla la asignación de memoria principal a los procesos. Si en algún momento el sistema no dispone de suficiente memoria principal, el núcleo transferirá algunos procesos de la memoria principal a la secundaria. A esta operación se le denomina *intercambio (swapping)* y con ella se intenta garantizar que todos los procesos tengan la oportunidad de ser ejecutados

El *módulo de planificación (scheduler)* asigna el uso de la CPU a los procesos. Un proceso A se ejecutará hasta que voluntariamente ceda el uso de la CPU (por ejemplo al tener que esperar por un recurso ocupado) o hasta que el núcleo lo expropie debido a que su *tiempo de utilización del procesador* o *cuanto* haya expirado. En ese momento, el planificador seleccionará para ejecutar al proceso de mayor prioridad de planificación que se encuentre listo para ser ejecutado. El proceso A volverá a ser ejecutado cuando sea el proceso de mayor prioridad de planificación listo para ejecución.

Existen diferentes formas de comunicación entre proceso, desde los mecanismos asíncronos de señalización de eventos (señales) hasta la transmisión síncrona de mensajes entre procesos.

2.6.2.3 Módulo de control del hardware

Finalmente, el módulo de *control del hardware* es el responsable del manejo de las interrupciones y de la comunicación con el hardware de la máquina.

2.7 EL INTERFAZ DE USUARIO PARA EL SISTEMA DE FICHEROS

2.7.1 Ficheros y directorios

Un *fichero* es un contenedor permanente de datos. Un fichero permite tanto el acceso secuencial como el acceso aleatorio a sus datos. El núcleo suministra al usuario varias operaciones de control para nombrar, organizar y controlar el acceso a los ficheros. El núcleo no interpreta el contenido o la estructura de los ficheros, simplemente considera que un fichero es una colección de bytes. Además posibilita el acceso a los contenidos del fichero mediante flujos de bytes.

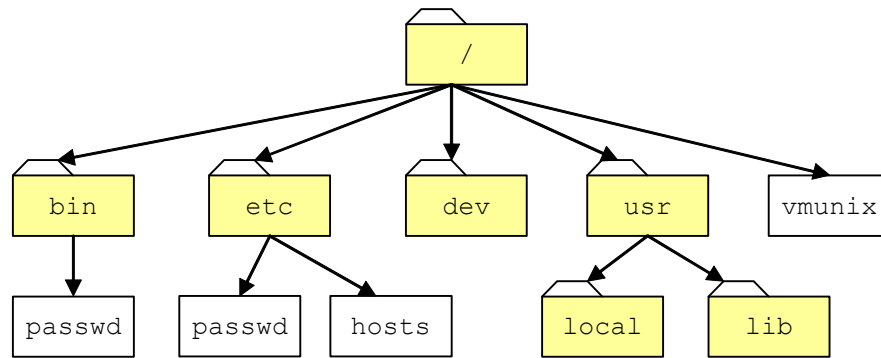


Figura 2.4. Un ejemplo de árbol de directorio

Un *directorio* contiene información sobre el nombre de los ficheros y directorios que residen en él. Desde el punto de vista del usuario, UNIX organiza los ficheros en un *árbol de directorios* (ver Figura 2.4) constituido por directorios y por ficheros. Este árbol comienza en el directorio raíz que se denota con '/'. Por debajo del directorio raíz se encuentran otros directorios importantes tales como:

- `/bin`. Contiene la mayoría de los programas ejecutables esenciales del sistema.
- `/etc`. Contiene diferentes ficheros de configuración del sistema.
- `/dev`. Aloja en diferentes subdirectorios los ficheros de dispositivos que permiten a los procesos comunicarse con los dispositivos periféricos.
- `/usr`. Aloja en una serie de subdirectorios diferentes programas y ficheros de configuración del sistema.

El nombre de un fichero o directorio puede contener cualquier carácter ASCII, excepto los caracteres '/' y '\0'. La longitud máxima de los nombres de ficheros y directorios está limitada por el sistema de ficheros. Los nombres de ficheros solo necesitan ser distintos dentro de un directorio. Por ejemplo, en la Figura 2.4 los directorios `bin` y `etc` contienen un fichero llamado `passwd`.

Para localizar a un fichero en el árbol de directorios, es necesario especificar su *ruta de acceso*, que puede ser de dos tipos: *absoluta* y *relativa*. La *ruta de acceso absoluta* está compuesta de todos los componentes en el camino desde el directorio raíz hasta el fichero, separados mediante caracteres '/'. Por lo tanto, en la Figura 2.4 los dos ficheros `passwd` tienen el mismo nombre, pero diferentes rutas, `/bin/passwd` y `/etc/passwd`,

respectivamente. El carácter ‘/’ en UNIX se utiliza tanto para el nombre del directorio raíz como para separar las componentes de la ruta.

Por otra parte, se denomina *directorio de trabajo actual* al directorio desde donde un usuario está ejecutando comando o programas. Esto permite a los usuarios referirse a los ficheros por su *ruta relativa*, que son interpretadas en relación al directorio actual. Existen dos componentes de ruta especiales: el primero es “.”, que se refiere al propio directorio ; el segundo es “..” que se refiere al directorio padre. El directorio raíz no tiene directorio padre, y su componente “..” se refiere al propio directorio raíz. Por ejemplo, en la Figura 2.4, un usuario cuyo directorio actual es `/usr/local/` puede referirse al directorio `lib` por su ruta absoluta: `/usr/lib` o por su ruta relativa: `../lib`.

Un proceso puede cambiar su directorio actual usando la llamada al sistema `chdir`. Asimismo, también un proceso puede designar otro directorio como su directorio raíz usando la llamada al sistema `chroot`. El cambio de directorio raíz puede resultar bastante útil cuando se están desarrollando aplicaciones que actúan sobre los ficheros de configuración del sistema.

La entrada de un fichero en un directorio constituye un *enlace duro* (o simplemente un *enlace*) para el fichero. Cualquier fichero puede tener uno o más enlaces, en el mismo o en diferentes directorios. Así un fichero no tiene porque estar limitado a estar en un único directorio y a tener un único nombre. Los enlaces del fichero son iguales en todas sus formas y son simplemente nombres diferentes para el mismo fichero. El fichero puede ser accedido a través de cualquiera de sus enlaces, y no hay forma de distinguir cual es el enlace original.

Los sistemas de ficheros UNIX modernos también suministran otro tipo de enlace denominado *enlace simbólico*, que son simplemente ficheros que contienen el nombre de un fichero.

2.7.2 Atributos de un fichero

Aparte del nombre de un fichero, el sistema de ficheros mantiene un conjunto de atributos para cada fichero. Estos atributos no están almacenados en la entrada del directorio, sino en una estructura del disco denominada *nodo índice (nodo-i)*. El formato y los contenidos de un nodo-i dependen del sistema de ficheros que se considere. Entre los atributos comúnmente soportados se encuentran:

- *Tipo de fichero.*
- Permisos e indicadores de modo (se explican en la próxima sección)
- *Número de enlaces duros* al fichero.
- *Tamaño del fichero* en bytes.
- *Identificador de dispositivo.* Es un número entero que identifica el dispositivo en el que se encuentra alojado el fichero. El identificador de dispositivo es una propiedad del sistema de ficheros, en consecuencia todos los ficheros de un mismo sistema de ficheros tienen el mismo identificador de dispositivo.
- *Número de nodo-i.* Es un número entero que identifica a cada nodo-i en un sistema de ficheros. Existe un único nodo-i asociado con cada fichero o directorio independientemente de cuantos enlaces duros tenga. De esta forma un fichero queda identificado de forma única mediante su identificador de dispositivo y su número de nodo-i. Cada entrada de un directorio almacena el número de nodo-i y el nombre de un fichero o de otro directorio.
- *Identificadores de usuario real (uid) y del grupo real (gid) del propietario del fichero.* Estos identificadores serán descritos en la sección 4.3.
- *Información asociada al tiempo:* la fecha y hora en que el fichero fue accedido por última vez, la fecha y hora en que el fichero fue modificado por última vez, y la fecha y la hora en que los atributos del fichero fueron cambiados por última vez (excluyendo la fecha y la hora en que el fichero fue accedido y/o modificado por última vez).

UNIX suministra varias llamadas al sistema para conocer y manipular los atributos de un fichero. Por ejemplo:

- `stat` y `fstat` permiten conocer los atributos de un fichero independientemente del formato que use un cierto sistema de ficheros.
- `link` y `unlink` crean o borran enlaces duros, respectivamente. El núcleo borra el fichero solo si todos sus enlaces duros han sido eliminados

- `utimes` cambia la fecha y la hora del último acceso o modificación de un fichero.
- `chown` cambia el *uid* y el *gid* del propietario del fichero.
- `chmod` cambia los permisos e indicadores de modo del fichero.

2.7.3 Modo de un fichero

Cada fichero en UNIX tiene asociada una máscara de 16 bits conocida como *máscara de modo* del fichero (ver Figura 2.5).

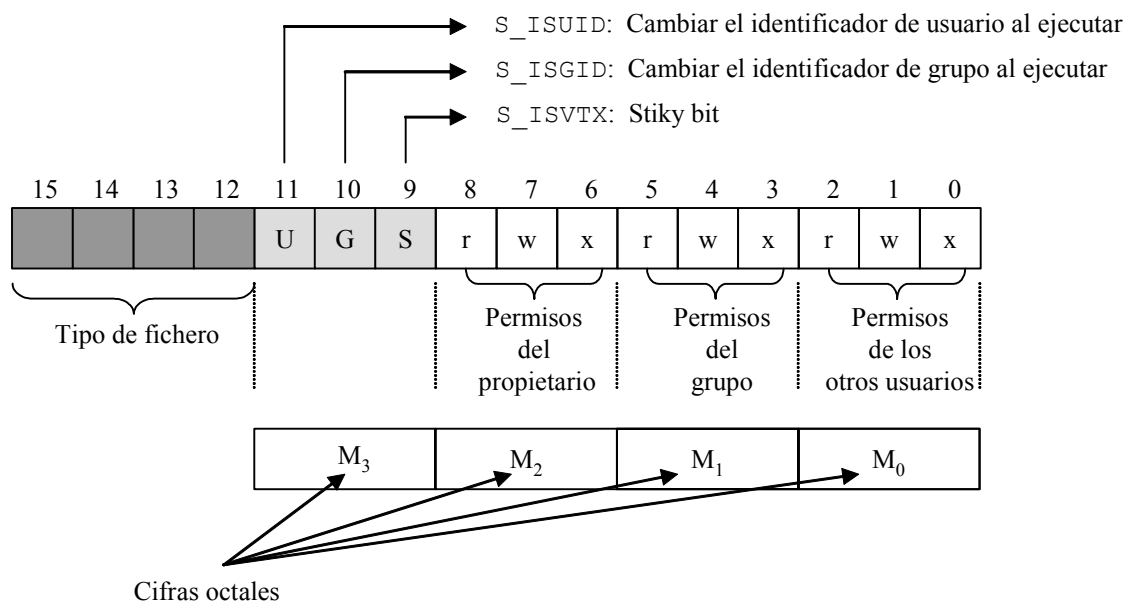


Figura 2.5: Máscara de modo de un fichero

El significado de los bits de la *máscara de modo* de un fichero es el siguiente:

- Bits para indicar el *Tipo de fichero* (bits 15-12).
- Bit `S_ISUID` (bit 11), la activación de este bit le indica al núcleo que cuando un proceso ejecute este fichero cambie el identificador de usuario efectivo *euid* del proceso y le ponga el valor del *uid* del propietario del fichero. Esto se explicará en detalle en la sección 4.3.

- Bit `S_ISGID` (bit 10), la activación de este bit tiene un significado parecido al de `S_ISUID`, pero referido al grupo de usuarios al que pertenece el propietario del fichero.
- Bit `S_ISVTX` (bit 9), este bit se denomina *sticky bit* y cuando se activa se le está indicando al núcleo que este fichero es un programa con capacidad para que varios procesos compartan su segmento de código y que este segmento se debe mantener en memoria, aún cuando alguno de los procesos que lo utiliza deje de ejecutarse o pase al área de intercambio. Esta técnica de compartición de código permite el ahorro de memoria en el caso de programas muy utilizados.
- *Permisos de acceso* al fichero (bits 8-0). Indican el tipo de permiso de acceso (lectura (r), escritura (w) y ejecución (x)) para el propietario del fichero, los usuarios pertenecientes al mismo grupo que el propietario y para otros usuarios.
 - El permiso de *lectura* permite a un usuario leer el contenido del fichero o en el caso de un directorio, listar el contenido del mismo (usando `ls`).
 - El permiso de *escritura* permite a un usuario escribir y modificar el fichero. Para directorios, el permiso de escritura permite crear nuevos ficheros o borrar ficheros ya existentes en dicho directorio.
 - El permiso de *ejecución* permite a un usuario ejecutar el fichero si es un programa o script del intérprete de comandos. Para directorios, el permiso de ejecución permite al usuario cambiar al directorio en cuestión con `cd`.

De los 16 bits de la máscara de modo de un fichero, el propietario del fichero o el superusuario pueden modificar únicamente los valores de los bits nº 11 a nº 0, ya que los cuatro bits más significativos asociados al tipo de fichero son configurados por el núcleo al crear dicho fichero. Por lo tanto la máscara de modo de un fichero se reduce desde el punto de vista de la posible manipulación del usuario a una máscara de 12 bits, que se agrupa en cuatro dígitos octales:

$$M_3M_2M_1M_0$$

- El dígito octal M_3 permite configurar el valor de los bits nº 11, 10 y 9, es decir, S_ISUID , S_ISGID y S_ISVTX .
- El dígito octal M_2 permite configurar el valor de los bits nº 8, 7 y 6, es decir, los permisos de lectura, escritura y ejecución del propietario del fichero.
- El dígito octal M_1 permite configurar el valor de los bits nº 5, 4 y 3, es decir, los permisos de lectura, escritura y ejecución de los usuarios pertenecientes al mismo grupo que el propietario del fichero.
- El dígito octal M_0 permite configurar el valor de los bits nº 2, 1 y 0, es decir, los permisos de lectura, escritura y ejecución de los otros usuarios.

En la Tabla 2.1 se representan el valor de los bits $i+2$, $i+1$ e i , con $i=3 \times j$ $j=0,1,2,3$ de la máscara de modo de un fichero en función del valor de la cifra octal M_j .

Cifra octal M_j	Bit $i+2$	Bit $i+1$	Bit i
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Tabla 2.1: Valor de los bits $i+2$, $i+1$ e i , con $i=3 \times j$ $j=0,1,2,3$ de la máscara de modo de un fichero en función del valor de la cifra octal M_j

♦ **Ejemplo 2.2:**

A continuación a modo de ejemplo, para varias máscaras de modo expresadas en octal se van especificar su máscara en binario y su significado.

- a) 0755. Su máscara binaria es 000 111 101 101. Los bits S_ISUID , S_ISGID y S_ISVTX están desactivados. El propietario del fichero puede leer, escribir y ejecutar el fichero. Los usuarios pertenecientes al grupo del fichero, y el resto de usuarios pueden leer y ejecutar el fichero.

- b) 0600. Su máscara binaria es 000 110 000 000. Los bits `S_ISUID`, `S_ISGID` y `S_ISVTX` están desactivados. El propietario del fichero puede leer y escribir. Nadie más puede acceder al fichero.
- c) 0777. Su máscara binaria es 000 111 111 111. Los bits `S_ISUID`, `S_ISGID` y `S_ISVTX` están desactivados. Todos los usuarios pueden leer, escribir y ejecutar el fichero.
- d) 7777. Su máscara binaria es 111 111 111 111. Los bits `S_ISUID`, `S_ISGID` y `S_ISVTX` están activados. Todos los usuarios pueden leer, escribir y ejecutar el fichero.
- e) 7666. Su máscara binaria es 111 110 110 110. Los bits `S_ISUID`, `S_ISGID` y `S_ISVTX` están activados. Todos los usuarios pueden leer y escribir el fichero, pero no pueden ejecutarlo.
- f) 7700. Su máscara binaria es 111 111 000 000. El bit `S_ISUID` está activado y los bits `S_ISGID` y `S_ISVTX` están activados. Solamente el propietario del fichero pueden leer, escribir y ejecutar el fichero.



2.7.4 Descriptores de ficheros

Cuando un usuario invoca a la llamada al sistema `open` para abrir un fichero el núcleo crea en memoria principal una estructura de datos asociada al fichero abierto que de forma general se denomina *objeto de fichero abierto*. En la distribución SVR3 (y anteriores) cada objeto de fichero abierto es almacenado en una entrada de una estructura global del núcleo denominada *tabla de ficheros*.

El núcleo también asigna un *descriptor de fichero*, que es un número entero positivo que actúa como identificador del objeto de fichero abierto. El *descriptor de fichero* es un identificador local a cada proceso, es decir, el mismo descriptor de fichero en dos procesos diferentes puede, y usualmente así lo hace, referirse a ficheros diferentes. Todos los descriptores de fichero asociados a un determinado proceso se suelen almacenar en una tabla denominada *tabla de descriptores de ficheros*. En conclusión, cada proceso posee su propia *tabla de descriptores de ficheros*.

Cuando se arranca un proceso en UNIX, el sistema abre para él, por defecto, tres ficheros que van a ocupar las tres primeras entradas de la tabla de descriptores. Estos ficheros se conocen como:

- Fichero estándar de entrada (*stdin*), que tiene asociado el descriptor número 0 y que por lo general es el teclado de un terminal.
- Fichero estándar de salida (*stdout*), que tiene asociado el descriptor número 1 y que por lo general es la pantalla de un terminal.
- Fichero estándar de salida de mensajes de error (*stderr*) que tiene asociado el descriptor número 2 y que por lo general también es la pantalla de un terminal.

El proceso pasa el descriptor de fichero a las llamadas al sistema asociadas con operaciones de E/S tales como `read` o `write`. El núcleo usa el descriptor para localizar rápidamente el objeto de fichero abierto. De esta forma, apoyándose en estos dos elementos el núcleo solamente necesita realizar una vez (durante la ejecución de `open`), y no en cada operación de E/S con el fichero, tareas tales como la búsqueda de la ruta de acceso o el control de acceso al fichero. Esto supone una mejora en el rendimiento del sistema.

Cada descriptor de fichero representa una sesión de trabajo independiente con el fichero. El objeto de fichero abierto asociado mantiene la información necesaria para poder continuar cada sesión. Esto incluye entre otros datos el modo de apertura del fichero y el *puntero de lectura/escritura* que es un desplazamiento en bytes desde el inicio del fichero. Este puntero marca la posición del fichero donde la próxima operación de lectura o de escritura debe comenzar.

En UNIX, los ficheros son accedidos secuencialmente por defecto. Cuando un usuario abre el fichero, el núcleo inicializa el puntero de lectura/escritura a cero. Cada vez que el proceso lee o escribe datos, el núcleo avanza el puntero en la cantidad de bytes transferidos.

El mantener un puntero de lectura/escritura en el objeto de fichero abierto permite al núcleo aislar unas de otras las diferentes sesiones de trabajo sobre un mismo fichero (ver Figura 2.6). Si dos procesos abren el mismo fichero, o si un proceso abre el mismo fichero dos veces, el núcleo genera un nuevo objeto de fichero abierto y un nuevo descriptor de fichero en cada invocación a `open`. De esta forma una operación de lectura o de escritura por un proceso producirá el avance de su propio puntero de lectura/escritura y no afectará al del otro. Esto permite que múltiples procesos compartan de forma transparente el mismo fichero.

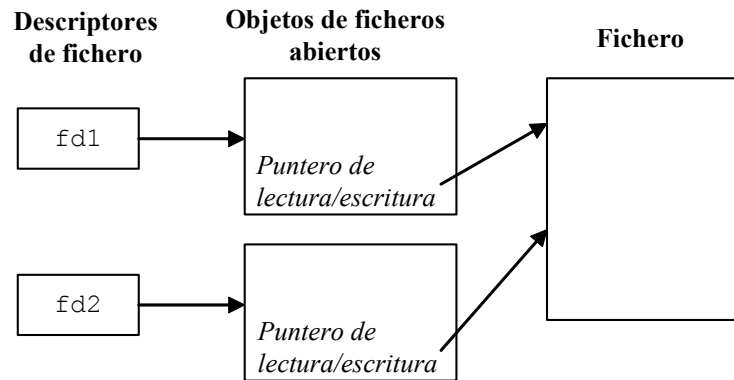


Figura 2.6: Un fichero es abierto dos veces

Por otra parte un proceso puede duplicar un descriptor usando las llamadas al sistema `dup` o `dup2`. Estas llamadas al sistema crean un nuevo descriptor que referencia al mismo objeto de fichero abierto y por tanto comparten la misma sesión de trabajo (ver Figura 2.7). Puesto que dos descriptors comparten la misma sesión para el fichero, ambos ven el mismo fichero y usan el mismo puntero de lectura/escritura.

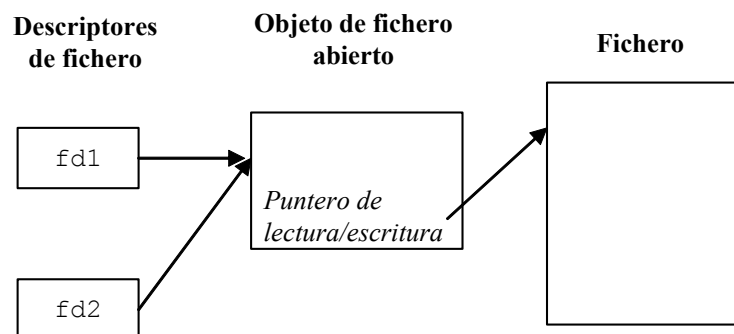


Figura 2.7: Descriptor clonado mediante las llamadas `dup`, `dup2` o `fork`.

De forma similar, la llamada al sistema `fork` que permite a un proceso (padre) crear otro proceso (hijo) duplica todos los descriptors del proceso padre y se los pasa al hijo. Después de retornar de `fork`, el padre y el hijo comparten el mismo conjunto de ficheros abiertos. Las versiones más modernas de UNIX como SVR4 y BSD4.3 permiten pasar un descriptor de fichero a otro proceso no relacionado genealógicamente con él, lo que puede resultar útil para aplicaciones en red.

2.7.5 Operaciones de E/S con un fichero

2.7.5.1 Apertura y cierre de un fichero

La llamada al sistema `open` permite abrir un fichero ya existente o crear uno nuevo. Su sintaxis es:

```
fd=open(path, flags, modo);
```

donde `path` es la ruta del fichero, `flags` es una máscara de bits que le indican al núcleo el modo en que se desea abrir el fichero y `modo` es la máscara de modo octal $M_3M_2M_1M_0$ que permite especificar los permisos que deben ser asociados al fichero si éste debe ser creado. Si la llamada al sistema se ejecuta con éxito en la variable entera `fd` se almacena un descriptor de fichero. En caso contrario en `fd` se almacena el valor -1.

Los bits de la máscara `flags` permiten especificar el modo de apertura del fichero mediante una combinación (usando el operador OR a nivel de bit ('|')), de entre otras las siguientes constantes:

- `O_RDONLY`. Abrir en modo sólo lectura.
- `O_WRONLY`. Abrir en modo sólo escritura.
- `O_RDWR`. Abrir para leer y escribir.
- `O_APPEND`. Situar el *puntero de lectura/escritura* al final del fichero para añadir datos.
- `O_TRUNC`. Si el fichero existe, trunca su longitud a cero bytes, incluso si el fichero se abre para leer.
- `O_CREAT`. Si el fichero no existe, crearlo con la máscara de permisos especificada en `modo`.

De las constantes `O_RDONLY`, `O_WRONLY` u `O_RDWR` sola una de ellas puede estar presente al componer la máscara `flags`, de lo contrario, el modo de apertura quedaría indefinido. Por otro lado, el argumento `modo` de `open` sólo se considera cuando está activo el indicador `O_CREAT`.

♦ Ejemplo 2.3:

La llamada al sistema `fd=open("texto.txt",O_RDONLY)` abre el fichero `texto.txt` en modo sólo lectura.

La llamada al sistema `fd=open("texto.txt",O_RDWR|O_APPEND)` abre el fichero `texto.txt` en modo lectura - escritura y fuerza a que el puntero de lectura/escritura del fichero se situé al final del fichero.

La llamada al sistema `fd=open("texto.txt",O_WRONLY|O_CREAT,0600)` abre el fichero `texto.txt` en modo sólo - escritura. Si el fichero no existe lo crea con permisos de lectura y escritura para el propietario del fichero y ningún permiso para el grupo y el resto de usuarios.

La llamada al sistema `fd=open("texto.txt", 0666)` abre el fichero `texto.txt` con permisos de lectura y escritura para todos los usuarios.

♦

Otra llamada al sistema que permite crear un fichero es `creat`. Su sintaxis es:

```
fd=creat(ruta, modo);
```

donde `path` es la ruta del archivo y `modo` es la máscara de modo octal $M_3M_2M_1M_0$ del fichero. Si el fichero ya existe, `creat` trunca su longitud a 0 bytes. Si la llamada al sistema se ejecuta con éxito en la variable entera `fd` se almacena un descriptor de fichero. En caso contrario en `fd` se almacena el valor -1.

Los ficheros abiertos son cerrados automáticamente cuando un proceso termina, si bien es posible cerrarlos de forma explícita usando la llamada al sistema `close`, cuya sintaxis es:

```
resultado=close(fd);
```

donde `fd` es el descriptor del fichero que se desea cerrar. Si la llamada al sistema se ejecuta con éxito en `resultado` se almacena el valor 0 en caso contrario se almacena el valor -1.

2.7.5.2 Lectura y escritura en un fichero

La llamada al sistema `read` permite leer en un fichero. Su sintaxis es:

```
nread=read(fd,buffer,nbytes);
```

donde `fd` es el descriptor de fichero, `buffer` es el array de caracteres donde se almacenarán los datos que se lean en el fichero, y `nbytes` es el número de bytes que se desea leer. Si la llamada al sistema se ejecuta con éxito en `nread` se almacenan el número de bytes transferidos. En caso de error en `nread` se almacena el valor -1.

El núcleo lee datos desde un fichero asociado con `fd`, comenzando en la posición indicada por el puntero de lectura/escritura almacenado en el objeto de fichero abierto. Puede leer menos bytes que `nbytes` si alcanza el final del fichero o, en el caso de los ficheros FIFO o de los ficheros de dispositivos, si no hay suficientes datos disponibles. Bajo ninguna circunstancia el núcleo transmitirá más que `nbytes` bytes. Es responsabilidad del usuario asegurarse que `buffer` es suficientemente grande para almacenar los `nbytes` bytes de datos. La llamada `read` también avanza el puntero de lectura/escritura en `nread` bytes para que la siguiente operación de lectura o de escritura comience donde la última operación ha terminado.

La llamada al sistema `write` permite escribir en un fichero. Su sintaxis es muy similar a la de `read`:

```
nwrite=write(fd,buffer,nbytes);
```

donde `fd` es el descriptor de fichero, `buffer` es el array de caracteres donde se encuentran almacenados los datos que se van a escribir en el fichero, y `nbytes` es el número de bytes que se desea escribir. Si la llamada al sistema se ejecuta con éxito en `nwrite` se almacenan el número de bytes escritos. En caso de error en `nwrite` se almacena el valor -1.

♦ Ejemplo 2.4:

```
#include <fcntl.h>
main()
{
    int ident;
    char buffer(100);
[1]  if((ident=open("escribir.txt",O_RDONLY))== -1)
    {
[2]      printf("ERROR DE APERTURA");
[3]      exit(1);
    }
[4]  gets(buffer);
[5]  if(write(ident,buffer,100)!=100)
```

```
[6]      printf("ERROR DE ESCRITURA");  
[7]      close(ident);  
}
```

Programa 2.1

Supóngase que el ejecutable que resulta de compilar este programa se llama `exwrite` y que es invocado desde la línea de ordenes del terminal:

```
$ exwrite
```

Al ejecutarse el programa en primer lugar [1] se invoca a la llamada al sistema `open` para abrir el fichero `escribir.txt` en modo sólo lectura. Si se la llamada no se ejecuta con éxito, en `ident` se almacena el valor -1. El programa comprueba esta circunstancia y en dicho caso se escribe [2] en la pantalla el mensaje

```
ERROR DE APERTURA
```

e invoca [3] a la llamada al sistema `exit` para finalizar la ejecución del programa.

Si la llamada `open` se ejecuta con éxito en la variable `ident` se almacena el descriptor del fichero. A continuación [4] se solicita al usuario que introduzca por el teclado un array de caracteres que se almacenará en la variable `buffer`.

Acto seguido [5] se invoca a la llamada al sistema `write` para escribir el contenido de `buffer` en el fichero cuyo descriptor es `ident`. Además se comprueba que la llamada `write` se ha ejecutado correctamente, es decir, se han escrito 100 caracteres (todo el contenido de `buffer`) en el fichero. En caso contrario se escribe en la pantalla [6] el mensaje

```
ERROR DE ESCRITURA
```

Finalmente [7] se invoca a la llamada al sistema `close` para cerrar el fichero cuyo descriptor es `ident` y el programa finaliza.

◆

2.7.5.3 Acceso aleatorio a un fichero

UNIX permite realizar tanto accesos secuenciales como accesos aleatorios a un fichero. El patrón de acceso por defecto es secuencial. El núcleo mantiene un puntero de lectura/escritura al fichero, que es inicializado a cero cuando un proceso abre por primera vez un fichero. La llamada al sistema `lseek` permite realizar accesos aleatorios mediante la configuración del puntero de lectura/escritura a un valor específico. Su sintaxis es:

```
resultado=lseek(fd, offset, origen);
```

donde `fd` es el descriptor del fichero, `offset` es el número de bytes que se va desplazar el puntero, y `origen` es la posición desde donde se va desplazar el puntero, que puede tomar los siguientes valores predefinidos:

- `SEEK_SET`. El puntero avanza `offset` bytes con respecto al inicio del fichero.
- `SEEK_CUR`. El puntero avanza `offset` bytes con respecto a su posición actual.
- `SEEK_END`. El puntero avanza `offset` bytes con respecto al final del fichero.

Si `offset` es un número positivo, los avances deben entenderse en su sentido natural; es decir, desde el inicio del fichero hacia el final del mismo. Sin embargo, también se puede conseguir que el puntero retroceda pasándole a `lseek` un desplazamiento negativo.

Si la llamada se ejecuta con éxito en `resultado` se almacena la posición que ha tomado el puntero de lectura/escritura, medida en bytes, con respecto al inicio del fichero. En caso de error en `resultado` se almacena el valor -1.

♦ Ejemplo 2.5:

```
#include <stdio.h>
#include <io.h>
#include <stdlib.h>
#include <fcntl.h>
#define TAM 128
void main(int argc, char* argv[])
{
    char buff[TAM+1], cad[10];
    int fd, sector;
[1]  if (argc!=2)
    {
[2]      printf("Falta el nombre del fichero");
[3]      exit(0);
    }
[4]  if ((fd=open(argv[1], O_RDONLY))==-1)
    {
[5]      printf("Imposible abrir el fichero\n");
```

```

[6]         exit(0);
        }
[7]     do
        {
[8]         printf("\n buffer: ");
[9]         gets(cad);
[10]        sector=atoi(cad);
[11]        if(lseek(fd, (long) sector*TAM, 0)==-1)
[12]            printf("\ERROR DE POSICIONAMIENTO\n");
[13]        if(read(fd, buff, TAM)==0)
[14]            printf("Sector fuera de rango\n")
        else
[15]            printf("%s", buff);
        } while (cad[0] != '*');
[16] close(fd);
    }

```

Programa 2.2

Supóngase que el ejecutable que resulta de compilar el Programa 2.2 se llama `exlseek` y que es invocado desde la línea de órdenes (\$) del terminal de la siguiente forma:

```
$ exlseek texto.txt
```

Puesto que la definición hecha de `main` permite el pasar parámetros al ejecutable, en primer lugar [1] se comprueba que al invocar a `exlseek` se le ha pasado como parámetro el nombre del fichero (`texto.txt`). En caso negativo se escribe en la pantalla [2] el mensaje

```
Falta el nombre del fichero
```

y se invoca [3] a la llamada al sistema `exit` para terminar el programa.

A continuación [4] se invoca a la llamada al sistema `open` para abrir en modo sólo lectura el fichero cuyo nombre (`texto.txt`) ha sido pasado como parámetro del ejecutable. Se comprueba si se ha producido un error durante la ejecución de `open`. En caso afirmativo entonces se escribe en la pantalla [5] el mensaje

```
Imposible abrir el fichero
```

y se invoca [6] a la llamada al sistema `exit` para terminar el programa.

Si `open` se ha ejecutado con éxito entonces en `fd` se habrá almacenado un descriptor de fichero para poder operar sobre el fichero abierto `texto.txt`.

En **[7]** el programa entra en un bucle del tipo `do-while`, en el que se realizan las siguientes acciones: **[8]**-**[15]**. En **[8]** se escribe en pantalla el mensaje

```
buffer:
```

En **[9]** se ejecuta la función de librería `gets` lo que permite al usuario poder introducir por teclado el número del sector del fichero que desea leer. En **[10]** se ejecuta la función de librería `atoi` para convertir dicho número de tipo carácter a tipo entero. En **[11]** se invoca a la llamada al sistema `lseek` para posicionar el puntero de lectura/escritura al principio de dicho sector del fichero. Asimismo se comprueba si durante la ejecución de `lseek` se ha producido algún error. En dicho caso se escribe en la pantalla **[12]** el mensaje:

```
ERROR DE POSICIONAMIENTO
```

En **[13]** se invoca a la llamada al sistema `read` para leer dicho sector y copiarlo en la variable `buff`. Asimismo se comprueba si durante la operación de lectura se ha sobrepasado el final del fichero (`read` devuelve el valor 0), entonces se escribe en la pantalla **[14]** el mensaje

```
Sector fuera de rango
```

En caso contrario, muestra en pantalla **[15]** el contenido de la variable `buff`.

```
[buff]
```

Sólo se saldrá del bucle `do-while`, cuando el usuario introduzca el carácter `^*`. Cuando se sale del bucle, se invoca **[16]** a la llamada al sistema `close` para cerrar el fichero y el programa finaliza.

Una posible traza de ejecución de este programa podría ser:

```
$ exlseek texto.txt
buffer: 0
Las tareas para este cuatrimestre son:
buffer: 1
En primer lugar leerse detenidamente los apuntes.
buffer: *
```



TEMA 3

ADMINISTRACION BASICA DEL SISTEMA UNIX

3.1 INTRODUCCION

Este tema pretende dar una pequeña introducción a las tareas básicas de administración de un sistema UNIX, tales como: la gestión de usuarios, la configuración de los permisos de acceso a los ficheros y el control de tareas.

Puesto que las distribuciones de UNIX requieren del pago de una licencia para poder ser instalado en un PC, para poder practicar con los conceptos que se explican en este tema y en los sucesivos temas se recomienda instalar cualquier distribución del sistema operativo *Linux*. Éste es un sistema operativo de distribución libre que puede considerarse como un clon de UNIX conforme a las especificaciones POSIX, aunque también posee ciertas extensiones propias del UNIX System V y BSD. El código de Linux es completamente original y es distribuido libremente bajo licencia GPL¹, que es la licencia pública del Proyecto GNU²

La historia de Linux comienza en Finlandia en 1991 cuando *Linus B. Torvalds*, por entonces un estudiante de la universidad de Helsinki, compró un PC equipado con un procesador 386 para estudiar su funcionamiento. Puesto que el sistema operativo MS/DOS no explotaba completamente las propiedades de los 386, Linus usó el sistema operativo Minix (creado por Andrew Tanenbaum) que se puede considerar como un sistema UNIX reducido. Motivado por las limitaciones de este sistema, Linus comenzó a reescribir ciertas partes del software para añadirles una mayor funcionalidad. Después, mediante el uso de Internet distribuyó su trabajo libremente con el nombre de *Linux*, que

¹ GPL es el acrónimo derivado del término inglés “General Public License ”

² GNU es el acrónimo recursivo para “GNU No es Unix”. Para una información detallada sobre el proyecto GNU se recomienda visitar www.gnu.org.

es una contracción de las palabras Linus y UNIX. Su primera versión oficial fue la versión 0.02 y fue hecha pública en octubre de 1991. Esta versión únicamente podía ejecutarse bajo Minix, además sólo permitía ejecutar unos pocos programas GNU, tales como `bash`, `gcc`, etc. Sin embargo, el hecho de que el código fuente fuera ampliamente diseminado por Internet ayudó a que el sistema se desarrollara rápidamente, ya que miles de personas en todo el mundo colaboraron desinteresadamente con Linus para mejorar el sistema.

Las primeras versiones de Linux eran relativamente inestables. La primera versión que afirmaba ser estable fue la versión 1.0 y fue hecha pública en marzo de 1994. El número de versión está asociada al ciclo de desarrollo del sistema, de hecho la evolución de Linux se ha realizado mediante una sucesión de dos fases: una fase de desarrollo y una fase de estabilización.

En una *fase de desarrollo* se pretende añadir mayor funcionalidad al núcleo probando nuevas ideas. En esta fase es cuando se realiza la mayor cantidad de trabajo sobre el núcleo. Obviamente debido a las manipulaciones a las que está siendo sometido, en esta fase el núcleo no es muy estable. Estas fases se distinguen porque las versiones se denotan con números impares, por ejemplo: 1.1, 1.3, etc.

Por otra parte en una *fase de estabilización* se pretende obtener un núcleo tan estable como sea posible, por lo que sólo se realizan ajustes y modificaciones menores. Estas fases se distinguen porque las versiones se denotan con números pares, por ejemplo: 1.0, 1.2, etc.

En la actualidad³, Linux es por completo un sistema basado en UNIX. Es estable, pero continua evolucionando. No solamente es capaz de controlar los últimos dispositivos periféricos del mercado sino que su comportamiento es comparable a ciertos sistemas UNIX comerciales, y se puede considerar superior en algunos puntos.

Finalmente, Linux está comenzando a salir del ámbito universitario para ser adoptado como sistema operativo por ciertas empresas. De hecho su potencia y flexibilidad, y el hecho de que es libre, está comenzando a interesar a un número creciente de compañías.

³ Para estar al tanto de la última versión del núcleo de Linux se recomienda visitar www.kernel.org

3.2 PRIMEROS PASOS EN UNIX

3.2.1 Entrar al sistema

UNIX es un sistema operativo multiusuario, por lo tanto, los usuarios deben identificarse para poder acceder al sistema. Este proceso de identificación consta de dos pasos:

- 1) *Introducción del nombre de usuario (login)*, que es el nombre con que el usuario será identificado por el sistema.
- 2) *Introducción de la contraseña (password) de acceso*, que es una clave personal secreta que posee cada usuario para poder entrar en el sistema.

♦ Ejemplo 3.1:

En el momento de entrar en el sistema, se verá la siguiente línea de comandos en la pantalla:

```
PULGAS login:
```

donde PULGAS es el nombre del ordenador (*hostname*).

A continuación se tecleará el nombre de usuario por ejemplo ALUMNO, con lo que en pantalla aparece:

```
PULGAS login: ALUMNO
```

```
Password:
```

Ahora se introduce la contraseña. Ésta no será mostrada en la pantalla conforme se va tecleando, como medida de seguridad, por lo que se debe teclear cuidadosamente. Si se introduce una contraseña incorrecta, se mostrará el siguiente mensaje:

```
Login incorrect
```

En ese caso, deberá intentarse de nuevo. Una vez que se ha introducido correctamente el nombre de usuario y la contraseña, se podrá tener acceso al sistema.

♦

3.2.2 Consolas virtuales

Por *consola del sistema* se entiende el monitor y teclado conectado directamente al sistema. UNIX, proporciona acceso a consolas virtuales, lo que permitirá tener más de

una sesión de trabajo activa. Para acceder a la segunda consola, se debe pulsar `Alt + F2`. En pantalla aparecerá el siguiente mensaje:

```
login:
```

Si es así esta será la segunda consola virtual, para volver a la primera, se pulsa `ALT+F1`. Las demás consolas virtuales se activan pulsando `ALT + F[n]`, siendo `[n]` el número de la consola.

3.2.3 Intérpretes de comandos

Un intérprete de comandos es simplemente un programa que toma las ordenes que teclea el usuario y las traduce a instrucciones. Esto puede ser comparado con el `command.com` de MS-DOS, el cual efectúa esencialmente la misma tarea. El intérprete de comandos es sólo uno de los interfaces con UNIX. Hay muchos interfaces posibles como el sistema X Windows, el cual permite ejecutar comandos usando el ratón y el teclado.

Tan pronto como se entra en el sistema, el sistema arranca un intérprete de comandos que permite dar órdenes al sistema.

♦ Ejemplo 3.2:

Un usuario entra en el sistema introduciendo su nombre de usuario y su *contraseña*

```
PULGAS login: ALUMNO
```

```
Password:
```

A continuación en la pantalla aparece el marcador (prompt) del intérprete de comandos, indicando que esta listo para recibir ordenes.

```
/home/ALUMNO$
```

Un ejemplo de orden sería:

```
/home/ALUMNO$ cp fich1 fich2
```

Aquí, el nombre de la orden es `cp` copiar, y los argumentos son `fich1` y `fich2`.

♦

Cuando se teclea una orden, el intérprete de comandos en primer lugar busca el nombre de la orden y comprueba si es una orden interna, es decir, una orden que el intérprete de comandos sabe ejecutar por si mismo. En segundo lugar comprueba si la

orden es un *alias*, es decir, un nombre sustitutorio de otra orden. Si no se cumple ninguno de estos casos, el intérprete de comandos busca el programa y lo ejecuta pasándole los argumentos especificados en la línea de comandos. En el caso de que no se pueda encontrar el programa con el nombre dado en la pantalla se mostrará un mensaje de error.

3.2.4 Comandos básicos

3.2.4.1 Cambiar el directorio de trabajo

El *directorio de trabajo inicial* es el directorio desde donde inicialmente empezará a trabajar el usuario cuando acceda al sistema. Cada usuario tiene su propio directorio de trabajo inicial, usualmente es un subdirectorio del directorio `/home`. En cualquier momento, las órdenes que se tecleen en el intérprete de comandos toman como referencia el directorio de trabajo.

Para cambiar el directorio de trabajo y moverse en la estructura de directorios se utiliza la orden `cd`, abreviatura de *cambio de directorio*. La sintaxis de este comando es:

```
cd <ruta>
```

donde `<ruta>` hace referencia a la ruta de acceso absoluta o relativa del nombre del directorio al que se quiere ir.

♦ Ejemplo 3.3:

Supóngase que el directorio de trabajo del usuario ALUMNO es `/home/ALUMNO`. Si ALUMNO quiere ir al subdirectorio `VARIOS`, se puede teclear la orden:

```
/home/ALUMNO$ cd VARIOS
```

o la orden:

```
/home/ALUMNO$ cd /home/ALUMNO/VARIOS
```

En el primer caso se está indicando la ruta relativa mientras que en el segundo se está indicando la ruta absoluta. En ambos casos como respuesta aparece el marcador

```
/home/ALUMNO/VARIOS$
```

Se observa que la línea de comandos de ALUMNO cambia para mostrar su directorio de trabajo actual. Para volver al directorio anterior, se puede teclear la orden:

```
/home/ALUMNO/VARIOS$ cd ..
```

o la orden

```
/home/ALUMNO/VARIOS$ cd /home/ALUMNO
```

Asimismo, como el directorio anterior es el directorio de trabajo inicial, también se puede teclear la orden

```
/home/ALUMNO/VARIOS$ cd
```

En los tres casos anteriores como respuesta aparece el marcador

```
/home/ALUMNO$
```



3.2.4.2 Obtener información de un fichero o de un directorio

Para obtener información sobre un fichero o un directorio se usa el comando `ls`. Su sintaxis más habitual es:

```
ls -<opciones>
```

donde el parámetro `<opciones>` permite detallar la información que se desea visualizar por pantalla. Los valores más frecuentes para `<opciones>` son:

- `F` se muestra información sobre el tipo de fichero.
- `l` se genera un listado largo incluyendo tamaño, propietario, permisos, etc.
- `i` se muestra en la primera columna el número de nodo-i de cada fichero.
- `r` se invierte el orden de clasificación.

Es posible especificar varias opciones simultáneamente, por ejemplo, `ls -lF`.

◆ Ejemplo 3.4:

Supóngase que se teclea la orden

```
home/ALUMNO$ ls
```

En pantalla aparecería la siguiente respuesta:

```
Correo
```

Cartas

VARIOS

Se observa que el usuario ALUMNO tiene tres entradas `Correo`, `Cartas` y `VARIOS` en su directorio de trabajo inicial `/home/ALUMNO`.

Esta información es claramente insuficiente, ya que no indica si se las entradas son ficheros o directorios. Para obtener parte esta información se puede teclear la siguiente orden:

```
/home/ALUMNO$ ls -F
```

En pantalla aparecería la siguiente respuesta:

`Correo/`

`Cartas/`

`VARIOS/`

Por el carácter `'/'` añadido a cada nombre se sabe que las tres entradas son subdirectorios. Si se hubiese añadido al final el carácter `'*'` estaría indicando que es un fichero ejecutable. Si no añade nada, entonces es un fichero ordinario.

◆

3.2.4.3 Crear directorios nuevos

Para crear un nuevo directorio se usa la orden `mkdir`. Cuya sintaxis es:

```
mkdir <dir1> <dir2> ...<dirN>
```

donde `<dir1> <dir2> ...<dirN>` son los nombres de los directorios que se desean crear.

◆ Ejemplo 3.5:

La orden

```
/home/ALUMNO$ mkdir PRUEBA
```

crearía dentro del directorio `ALUMNO` el subdirectorio `PRUEBA`. Esto se puede comprobar escribiendo la siguiente orden

```
/home/ALUMNO$ ls -F
```

En pantalla aparecería la siguiente respuesta:

`Correo/`

PRUEBA/

Cartas/

VARIOS/

3.2.4.4 Copiar ficheros

Para copiar ficheros se usa la orden `cp`. Su sintaxis es

```
cp <fichero1> <fichero2> ... <ficheroN> <destino>
```

donde `<fichero1>` `<fichero2>` ... `<ficheroN>` son las rutas de acceso de los ficheros a copiar, y `<destino>` es la ruta del directorio donde se van a copiar.

◆ Ejemplo 3.6:

La orden

```
/home/ALUMNO/PRUEBA$ cp /etc/host.conf .
```

copia en el directorio de trabajo actual `.'` el fichero `host.conf` (que está dentro del directorio `/etc/`). Esto se puede comprobar mediante la siguiente orden:

```
/home/ALUMNO/PRUEBA$ ls -F
```

En pantalla aparecería la siguiente respuesta:

```
host.conf
```

◆

3.2.4.5 Mover ficheros

Para mover ficheros de un directorio a otro se puede usar la orden `mv` cuya sintaxis es:

```
mv <fichero1> <fichero2> ... <ficheroN> <destino>
```

donde `<fichero1>` `<fichero2>` ... `<ficheroN>` son las rutas de acceso de los ficheros que se desean mover y `<destino>` es el directorio destino.

Asimismo `mv` también se puede usar para renombrar un fichero.

♦ Ejemplo 3.7:

Supóngase que en el directorio de trabajo actual se encuentran los ficheros `prueba.txt` y `host.conf`. En ese caso la orden

```
/home/ALUMNO/PRUEBA$ mv host.conf ..
```

mueve el fichero `host.conf` al directorio `/home/ALUMNO`

Asimismo, para cambiar el nombre del fichero `prueba.txt` por el nombre `prueba2.txt` habría que teclear la siguiente orden:

```
/home/ALUMNO/PRUEBA$ mv prueba.txt prueba2.txt
```

♦

3.2.4.6 Borrar ficheros y directorios

Para borrar un fichero, se usa la orden `rm` su sintaxis es

```
rm <fichero1> <fichero2> ... <ficheroN>
```

donde `<fichero1> <fichero2> ... <ficheroN>` son las rutas de acceso de los ficheros que se desean borrar. Por defecto esta orden no pregunta antes de borrar los ficheros. Si se desea que se pida una confirmación antes de borrar cada fichero hay que colocar al final de la orden la opción `-i`.

Una orden relacionada con `rm` es `rmdir`. Esta orden borra un directorio, pero sólo si está vacío. Si el directorio contiene ficheros o subdirectorios, `rmdir` generará un mensaje de aviso por pantalla.

♦ Ejemplo 3.8:

La siguiente orden

```
/home/ALUMNO/PRUEBA$ rm prueba2.txt
```

borraría el fichero `prueba2.txt` del directorio de trabajo actual.

♦

3.2.4.7 Acceder al contenido de los ficheros

Para ver el contenido de un fichero se pueden usar las ordenes `more` y `cat`. El comando `more` muestra el fichero pantalla a pantalla mientras que `cat` lo muestra entero de una vez. Sus sintaxis son

```
more <fichero1> <fichero2> ... <ficheroN>
```

```
cat <fichero1> <fichero2> ... <ficheroN>
```

donde `<fichero1> <fichero2> ... <ficheroN>` son las rutas de acceso de los ficheros cuyo contenido se desea mostrar por pantalla

Cuando se ejecuta `more` se debe pulsar `<espacio>` para avanzar a la página siguiente y `b` para volver a la página anterior. Si se pulsa la tecla `q` finalizará la ejecución de `more`.

♦ Ejemplo 3.9:

La orden

```
/home/ALUMNO/PRUEBA$ more /etc/host.conf
```

mostraría pantalla a pantalla el contenido del fichero `host.conf` situado dentro del directorio `/etc`.

Si se tecleara la siguiente orden:

```
/home/ALUMNO/PRUEBA$ cat /etc/host.conf
```

el contenido del fichero se mostraría por pantalla demasiado rápido como para poder leerlo.

♦

Por otra parte la orden `cat` puede ser usada para concatenar el contenido de varios ficheros y guardar el resultado en otro fichero. En este caso su sintaxis es

```
cat <fichero1> <fichero2> ... <ficheroN> > <destino>
```

donde `<destino>` es la ruta del fichero destino. Obsérvese que antes de `<destino>` se ha colocado el símbolo `'>'` que indica al intérprete de comandos que debe cambiar la salida estándar del sistema donde se presentará el resultado de la orden. Ahora la salida estándar, en vez de ser la pantalla, será el fichero `<destino>`.

El cambio de la salida estándar mediante el símbolo ‘>’ puede hacerse para todos los comandos, no solo para `cat`. De forma análoga el símbolo ‘<’ permite cambiar la entrada estándar para que en vez de ser el teclado, sea un fichero.

◆ **Ejemplo 3.10:**

La orden

```
/home/ALUMNO$ cat capitulo1 capitulo2 capitulo3 > libro
```

concatenará el contenido de los ficheros `capitulo1`, `capitulo2` y `capitulo3` y guardará el resultado en el fichero `libro`.

◆

3.2.4.8 Acceder al manual de ayuda

UNIX dispone en su línea de comandos de un manual de ayuda para los comandos y los recursos del sistema (como las funciones de librería). La orden usada para acceder a la ayuda del sistema es `man`, su sintaxis es:

```
man <nombre>
```

donde `<nombre>` es el nombre del comando o recurso del sistema cuya página del manual de ayuda se desea visualizar por pantalla.

Es importante recordar que no todos los comandos disponen de página de ayuda.

◆ **Ejemplo 3.11:**

Si se escribe la orden

```
/home/ALUMNO$ man ls
```

se mostrará por pantalla la página del manual de ayuda asociada a `ls`.

◆

3.2.4.9 Edición de ficheros

Para editar un fichero se puede invocar desde la línea de comandos, por ejemplo, el editor `vi`. Para conocer las opciones disponibles en este editor se recomienda consultar su página del manual de ayuda.

3.2.4.10 Salir del sistema

Para salir del sistema desde la línea de comandos se pueden usar los siguientes comandos `exit`, `halt` o `shutdown`.

3.3 CONSIDERACIONES GENERALES DE LOS INTÉRPRETES DE COMANDOS

3.3.1 Tipos de intérpretes de comandos

Existen diferentes intérpretes de comandos UNIX, siendo los más importantes *Bourne* y *C*.

El intérprete *Bourne*, usa la sintaxis de comandos de los primeros sistemas UNIX, como el UNIX System III. El nombre del intérprete *Bourne* en la mayoría de las distribuciones de UNIX es `sh`⁴. Normalmente su ruta de acceso es `/bin/sh`.

Por su parte el intérprete *C* usa una sintaxis diferente, semejante a la del lenguaje de programación C. El nombre del intérprete *C* en la mayoría de las distribuciones de UNIX es `csh`. Normalmente su ruta de acceso es `/bin/csh`.

En Linux dos de los intérpretes más utilizados son *Bash* (*Bourne Again Shell*) y *Tcsh*⁵. *Bash* es equivalente al intérprete *Bourne* pero incluye muchas características del intérprete *C*. Normalmente su ruta de acceso es `/bin/bash`.

Por su parte *Tcsh*, es una versión extendida del intérprete *C*. Normalmente su ruta de acceso es `/bin/tcsh`.

El usar un intérprete de comandos u otros es cuestión de gustos. Algunas personas prefieren la sintaxis del intérprete *Bourne* con las características avanzadas que proporciona el intérprete *Bash*, y otros prefieren el más estructurado intérprete *C*. En lo que respecta a los comandos usuales como `cp`, `ls`, etc., es indiferente el tipo de intérprete de comandos usado, la sintaxis es la misma. Sólo cuando se usan características avanzadas de los intérpretes es cuando se pueden apreciar las diferencias entre ellos.

⁴ *sh* son las dos primeras letras de la palabra inglesa *shell*, que es el término que se utiliza para denotar a un intérprete de comandos.

⁵ La letra *t* al principio de *tcsh* viene de la T de TENEX que es el sistema operativo en el que se inspiró Ken Greer para escribir este intérprete de comandos.

3.3.2 Variables del intérprete de comandos y el entorno

Un usuario, desde el intérprete de comandos, puede definir variables que pueden ser accedidas o consultadas por las posteriores órdenes que se realicen desde el intérprete. La forma de definir una variable es:

NOMBRE_VARIABLE=VALOR_VARIABLE

♦ Ejemplo 3.12:

La orden

```
/home/ALUMNO$ PRUEBA="Hola a todos"
```

Crea una nueva variable denominada PRUEBA y le asigna el valor "Hola a todos".

Por su parte la orden

```
/home/ALUMNO$ echo $PRUEBA
```

Muestra por pantalla el mensaje

```
Hola a todos
```

Es decir, permite visualizar por pantalla el valor de la variable PRUEBA.

♦

Estas variables son internas al intérprete. Esto significa que solo este podrá acceder a las variables. Usando la orden `set` se mostrará una lista de todas las variables definidas en el intérprete de comandos. De cualquier modo, el intérprete de comandos permite exportar variables al entorno. El *entorno* se define como el conjunto de variables a las cuales tienen acceso todas las órdenes que se ejecuten. Una vez que se define una variable en el intérprete, exportarla hace que se convierta también en parte del entorno. La orden `export` es usada para exportar variables al entorno. El entorno es muy importante en un sistema Unix. Le permite configurar ciertas órdenes simplemente inicializando variables con las órdenes ya conocidas.

♦ Ejemplo 3.13:

La variable de entorno `PAGER` es usada por la orden `man`, permite especificar el comando que se utilizará para mostrar las páginas del manual, por defecto usa el comando `PAGER="more"`.

La orden

```
/home/ALUMNO$ PAGER="cat "
```

asigna a la variable `PAGER` el valor `cat`, esto hará que la salida de `man` sea mostrada de una vez, sin pausas entre páginas.

Sin embargo, para que el cambio en la variable `PAGER` tenga efecto es necesario exportarla al entorno, para ello se debe teclear la orden

```
/home/ALUMNO$ export PAGER
```



3.3.3 La variable de entorno `PATH`

Una de las variables de entorno más importantes es `PATH`, que es utilizada por el intérprete de comandos para localizar los ficheros ejecutables u los comandos que se teclean. Sin embargo, `PATH` no interviene en la localización de los ficheros ordinarios.

◆ Ejemplo 3.14:

La orden

```
/home/ALUMNO$ PATH=/bin:/usr/bin:/usr/local/bin:.
```

Configura la variable `PATH` con la lista de rutas de accesos de los directorios (cada ruta viene separada por ':')

```
/bin:/usr/bin:/usr/local/bin:.
```

donde el intérprete debe buscar las ordenes y ficheros ejecutables que se ejecuten. Así cuando se teclea un comando, el intérprete lo buscará primero en `/bin`, luego en `/usr/bin`, luego en `/usr/local/bin` y finalmente en el directorio de trabajo actual ('.').

Por otra parte, por ejemplo si se usa la orden

```
/home/ALUMNO$ cp PRUEBA DESTINO
```

El intérprete no usará `PATH` para localizar a los ficheros `PRUEBA` y `DESTINO`, esos nombres se suponen que son las rutas de acceso. Sólo se usará `PATH` para localizar al comando `cp`.



3.3.4 Caracteres comodines

Una característica importante de la mayoría de los intérpretes de comandos en UNIX es la capacidad para referirse a más de un fichero usando caracteres especiales denominados *comodines*, tales como '*', '?' o "[]".

El comodín "*" hace referencia a cualquier carácter o cadena de caracteres en el fichero. Por ejemplo, cuando se usa el carácter "*" en el nombre de un fichero, el intérprete de comandos lo sustituye por todas las combinaciones posibles provenientes de los ficheros en el directorio de trabajo.

El proceso de la sustitución de "*" en nombres de ficheros es llamado *expansión de comodines* y es efectuado por el intérprete de comandos.

◆ Ejemplo 3.15:

Supóngase que el usuario ALUMNO tiene los ficheros `metas` y `telefonos` en el directorio de trabajo actual. Para ver un listado de los ficheros que poseen alguna letra 'o' en su nombre, se puede usar la orden:

```
/home/ALUMNO$ ls *o*
```

En pantalla aparecería la siguiente respuesta:

```
telefonos
```

Como se puede ver, el comodín '*' ha sido sustituido con todas las combinaciones posibles que coincidían con la estructura propuesta de entre los ficheros del directorio de trabajo actual.

Por otra parte la orden

```
/home/ALUMNO$ ls *
```

mostraría la siguiente respuesta en la pantalla

```
metas telefonos
```

Es decir, se han listado todos los ficheros existentes en el directorio de trabajo actual, puesto que todos los caracteres coinciden con el comodín.

◆

Otro carácter comodín es '?' que expande un único carácter. Luego `ls ?` mostrará todos los nombres de ficheros con un carácter de longitud, y `ls ejemplo?` mostrará ejemplos pero no ejemplos.txt.

También, otro comodín es "[]", que sustituye a los caracteres incluidos dentro del paréntesis. Así por ejemplo la orden `ls *[ae]` permite listar los ficheros que terminen con las letras 'a' o 'e'.

En definitiva, los caracteres comodines permiten referirse a más de un fichero a la vez.

3.3.5 Scripts del intérprete de Comandos

El intérprete de comandos no es sólo un intérprete interactivo de los comandos que se teclean, es también un lenguaje de programación, el cual permite escribir *scripts*, es decir, ficheros de texto que contienen varias órdenes. Los scripts son análogos a los ficheros batch en el sistema MS-DOS. El intérprete de comandos lee cada línea del script y ejecuta la línea como si hubiese sido tecleada en la línea de comandos.

La aplicación de los scripts es evidente ya que si un usuario a menudo teclea consecutivamente los mismos comandos, en lugar de teclear todos esos comandos uno a uno, podría agruparlos en un script del intérprete de comandos y ejecutarlo.

Un usuario no podrá ejecutar un script si no tiene los permisos de ejecución oportunos.

◆ Ejemplo 3.16:

La orden

```
/home/ALUMNO$ cat capitulo1 capitulo2 capitulo3 > libro
```

concatenará el contenido de los ficheros capítulo1, capítulo2 y capítulo3 y lo guardará en el fichero libro. Si ahora se teclea la orden

```
/home/ALUMNO$ wc -l libro
```

mostrará el recuento del número de líneas del fichero libro. Finalmente si se teclea la orden

```
/home/ALUMNO$ lp libro
```

se imprimirá el contenido del fichero.

Todas estas órdenes se podrían agrupar en el siguiente script llamado, por ejemplo, `makelibro`:

```
#!/bin/sh

# script para crear e imprimir libro

cat capitulo1 capitulo2 capitulo3 > libro

wc -l libro

lp libro
```

La primera línea `#!/bin/sh`, identifica el fichero como un script e indica que tipo de intérprete debe ejecutarlo, en este caso se usará el intérprete `sh` para su ejecución, cuya ruta de acceso es `/bin/sh`. La segunda línea es un comentario, ya que comienza con el carácter `#`. Cada línea de comentario debe de comenzar por este carácter. Los comentarios son ignorados por el intérprete de comandos. El resto de las líneas del script son simplemente órdenes como las que se pueden teclear directamente.

La orden:

```
/home/ALUMNO$ chmod u+x makelibro
```

otorgaría al usuario permiso de ejecución sobre el script `makelibro`. Finalmente la invocación de este script se realizaría con la orden:

```
/home/ALUMNO$ makelibro
```

◆

3.4 GESTION DE USUARIOS

3.4.1 Cuentas de usuario

UNIX es un sistema operativo multiusuario, cada usuario tiene su propia cuenta que le permite acceder al sistema. Además, existe una cuenta especial *root* definida por el sistema. El usuario *root* o *superusuario* puede leer, modificar o borrar cualquier fichero en el sistema. Además puede cambiar permisos y ejecutar programas especiales. Todos estos privilegios están vetados para un usuario normal.

Puesto que es fácil cometer errores que tengan consecuencias catastróficas para el sistema, la cuenta *root* debe ser usada exclusivamente por el administrador del sistema para la realización de aquellas tareas, que por falta de privilegios, no pueden ser ejecutadas desde una cuenta normal.

Comenzada una sesión de trabajo un usuario puede saber si se encuentra en la cuenta *root* observando el marcador de la línea de comandos. Usualmente se suele utilizar el carácter "\$" como marcador para los usuarios normales, y el carácter "#" como marcador para el superusuario.

El sistema almacena información sobre las cuentas de usuarios en el fichero `/etc/passwd`. Cada línea de este fichero contiene la siguiente información acerca de un único usuario:

- *Nombre de usuario (login)*. Es un identificador único dado a cada usuario del sistema. Este identificador pueden contener los siguientes caracteres: letras, dígitos, "_" y ".". Además su longitud está limitada normalmente a 8 caracteres de longitud.
- *Identificador de usuario real (uid)*. Es un número único dado a cada usuario del sistema.
- *Identificador de grupo real (gid)*. Es un número único dado a cada grupo de usuarios del sistema.
- *Contraseña (password)*. Es una clave personal secreta que posee cada usuario para poder entrar en el sistema. Como medida de seguridad el sistema almacena encriptada esta clave.
- *Nombre real o completo del usuario*.
- *Directorio de trabajo inicial*. Es el directorio desde donde inicialmente empezará a trabajar el usuario cuando acceda al sistema. Cada usuario debe tener su propio directorio inicial, normalmente como un subdirectorio del directorio `/home`.
- *Intérprete de comandos inicial*. Es el intérprete de comandos con el que comenzará a trabajar el usuario cuando acceda al sistema.

◆ **Ejemplo 3.17:**

Supóngase que el fichero `/etc/passwd` posee entre otras la siguiente línea:

```
C3PO:Xv8Q981g71oKK:102:100:Juan Gomez:/home/C3PO:/bin/bash
```

El significado de los elementos de esta línea es el siguiente:

- `C3PO` es el nombre de usuario.
- `Xv8Q981g71oKK` es la clave encriptada.
- `102` es el *uid*.
- `100` es el *gid*.
- `Juan Gomez` es el nombre completo del usuario.
- `/home/C3PO` es el directorio de trabajo inicial.
- `/bin/bash` es el intérprete de comandos inicial.

◆

3.4.2 Creación y eliminación de una cuenta de usuario

El superusuario es el único que puede crear o eliminar una cuenta de usuario. La manera más simple de realizar estas acciones es utilizando un programa interactivo que vaya preguntando por la información necesaria y actualice todos los ficheros del sistema automáticamente.

Así si se desea crear una cuenta de usuario se debe usar el programa `useradd` o `adduser`. Por el contrario si se desea eliminar una cuenta de usuario se debe utilizar el programa `userdel` o `deluser`, dependiendo de que software fuera instalado en el sistema.

Por otra parte, si se desea deshabilitar temporalmente la cuenta de un usuario sin borrarla, basta con colocar el carácter "*" delante de la clave encriptada de la línea correspondiente del fichero `/etc/passwd`.

◆ Ejemplo 3.18:

Cambiando la línea de `/etc/passwd` correspondiente a `C3PO` a:

```
C3PO:*Xv8Q981g71oKK:102:100:Juan Gómez:/home/C3PO:/bin/bash
```

se evitará que `C3PO` pueda acceder a su cuenta.

◆

3.4.3 Modificación de la información asociada a una cuenta de usuario

El superusuario puede modificar la información asociada a una cuenta de usuario. La forma más simple de hacer esto es cambiar los valores directamente en la línea apropiada del fichero `/etc/passwd`.

Por otra parte si un usuario desea modificar la contraseña de acceso a su cuenta puede utilizar el comando `passwd`, que solicita la contraseña vieja y la contraseña nueva. Esta última la solicita dos veces para validarla.

Si un usuario olvida su contraseña deberá pedirle al superusuario que le asigne una nueva contraseña.

3.4.4 Grupos de usuarios

Cada usuario puede pertenecer a uno o más grupos, lo que implica el tener unos determinados permisos de acceso a un fichero. Cada fichero tiene un grupo propietario y un conjunto de permisos de grupo que definen de que forma pueden acceder al fichero los usuarios del grupo.

El fichero `/etc/group` contiene información acerca de los grupos de usuarios existentes en el sistema. Cada línea de este fichero contiene la siguiente información acerca de un único grupo: nombre del grupo, clave encriptada de acceso a un grupo (rara vez se utiliza), *gid*, y otros miembros del grupo.

◆ Ejemplo 3.19:

Supóngase que el fichero `/etc/group` contiene entre otras las siguientes líneas:

```
root:*:0:

users:*:100:PROFESOR,ALUMNO

invitados:*:200:

otros:*:250:C3PO
```

La primera línea contiene la siguiente información: El nombre del grupo es `root`, que es un grupo especial del sistema reservado para la cuenta `root`. No tiene especificada contraseña de entrada. Su *gid* es 0. Este grupo consta de un único miembro, el superusuario.

La segunda línea contiene la siguiente información: El nombre del grupo es `users`. No tiene especificada contraseña de entrada. Su `gid` es 100. A este grupo tienen acceso los usuarios que tengan asignados un `gid=100` (recuérdese que en `/etc/passwd` cada usuario tiene un `gid` por defecto) y los usuarios `PROFESOR` y `ALUMNO`.

La tercera línea contiene la siguiente información: El nombre del grupo es `invitados`. No tiene especificada contraseña de entrada. Su `gid` es 200. A este grupo únicamente tienen acceso los usuarios que tengan asignados un `gid=200`.

Finalmente la cuarta línea contiene la siguiente información: El nombre del grupo es `otros`. No tiene especificada contraseña de entrada. Su `gid` es 250. A este grupo tienen acceso los usuarios que tengan asignados un `gid=250` y el usuario `C3PO`.

◆

El superusuario es el único que puede crear o eliminar un grupo de usuarios. Si se desea añadir un nuevo grupo o algún usuario a un grupo ya definido la manera más simple de realizar estas acciones es añadir una nueva línea o modificar una línea ya existente del archivo `/etc/group`. También se pueden utilizar los comandos `addgroup` o `groupadd` para añadir grupos a su sistema. Para borrar un grupo, solo hay que borrar su entrada de `/etc/group`.

3.5 CONFIGURACION DE LOS PERMISOS DE ACCESO A UN FICHERO

3.5.1 Máscara de modo simbólica

El comando `ls -l` muestra información detallada sobre los ficheros contenidos en el directorio de trabajo actual. En concreto al principio de la línea asociada a cada fichero muestra una cadena de caracteres con información sobre la máscara de modo del fichero. A dicha cadena se le denominará *máscara de modo simbólica*. Su estructura es:

$S_9 S_8 S_7 S_6 S_5 S_4 S_3 S_2 S_1 S_0$

- El carácter s_9 , indica el tipo de fichero: ordinario ($-$), directorio (d), especial modo carácter (c), especial modo bloque (b), FIFO o tubería (p) y conector (socket) (s).
- El carácter s_8 , indica si se encuentra habilitado el permiso de lectura (r) para el propietario del fichero, en caso negativo tomará el valor ($-$).

- El carácter s_7 , indica si se encuentra habilitado el permiso de escritura (w) para el propietario del fichero, en caso negativo tomará el valor $(-)$.
- El carácter s_6 , puede indicar varias cosas:
 - * Si vale (s) indica que el bit S_ISUID está activado y que se encuentra habilitado el permiso de ejecución para el propietario del fichero.
 - * Si vale (S) indica que el bit S_ISUID está activado y que se encuentra deshabilitado el permiso de ejecución para el propietario del fichero.
 - * Si vale (x) indica que el bit S_ISUID no está activado y se encuentra habilitado el permiso de ejecución para el propietario del fichero.
 - * Si vale $(-)$ indica que el bit S_ISUID no está activado y se encuentra deshabilitado el permiso de ejecución para el propietario del fichero.
- El carácter s_5 , indica si se encuentra habilitado el permiso de lectura (r) para los miembros del grupo al que pertenece el propietario del fichero, en caso negativo tomará el valor $(-)$.
- El carácter s_4 , indica si se encuentra habilitado el permiso de escritura (w) para los miembros del grupo al que pertenece el propietario del fichero, en caso negativo tomará el valor $(-)$.
- El carácter s_3 , puede indicar varias cosas:
 - * Si vale (s) indica que el bit S_ISGID está activado y que se encuentra habilitado el permiso de ejecución para los miembros del grupo al que pertenece el propietario del fichero.
 - * Si vale (S) indica que el bit S_ISGID está activado y que se encuentra deshabilitado el permiso de ejecución para los miembros del grupo al que pertenece el propietario del fichero.
 - * Si vale (x) indica que el bit S_ISGID no está activado y se encuentra habilitado el permiso de ejecución para los miembros del grupo al que pertenece el propietario del fichero.

- * Si vale (-) indica que el bit `S_ISUID` no está activado y se encuentra deshabilitado el permiso de ejecución para los miembros del grupo al que pertenece el propietario del fichero.
- El carácter `s2`, indica si se encuentra habilitado el permiso de lectura (`r`) para el resto de usuarios, en caso negativo tomará el valor (-).
- El carácter `s1`, indica si se encuentra habilitado el permiso de escritura (`w`) para el resto de usuarios, en caso negativo tomará el valor (-).
- El carácter `s0`, puede indicar varias cosas:
 - * Si vale (`t`) indica que el bit `S_ISVTX` está activado y que se encuentra habilitado el permiso de ejecución para el resto de usuarios.
 - * Si vale (`T`) indica que el bit `S_ISVTX` está activado y que se encuentra deshabilitado el permiso de ejecución para el resto de usuarios.
 - * Si vale (`x`) indica que el bit `S_ISVTX` no está activado y se encuentra habilitado el permiso de ejecución para el resto de usuarios.
 - * Si vale (-) indica que el bit `S_ISVTX` no está activado y se encuentra deshabilitado el permiso de ejecución para el resto de usuarios.

◆ Ejemplo 3.20:

Supóngase que en el directorio de trabajo actual existe únicamente el fichero `notas`. La orden:

```
/home/ALUMNO/PRUEBA$ ls -l
```

mostraría en pantalla la siguiente línea:

```
-rw-r--r--  1 ALUMNO   users 515 Mar 12 17:05 notas
```

El significado de los elementos de esta línea es el siguiente: `-rw-r--r--` es la máscara de modo simbólica, `1` es el número de enlaces duros, `ALUMNO` es el propietario del fichero, `users` es el grupo al cual pertenece el propietario, `515` es el tamaño del fichero en bytes, `Mar 12 17:05` es la fecha de la última modificación del fichero y `notas` es el nombre del fichero.

La máscara de modo simbólica `-rw-r--r--` (ver Figura 3.1) informa, por orden, de los permisos para el propietario, los miembros del grupo al que pertenece el propietario del fichero y otros usuarios.

s_9	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0
-	r	w	-	r	-	-	r	-	-
Tipo de fichero	Propietario			Grupo			Otros usuarios		

Figura 3.1: Máscara de modo simbólica del fichero `notas`

El primer carácter de la cadena de permisos $s_9 = -$ representa el tipo de fichero. El `-` significa que es un fichero regular.

Las siguientes tres letras ($s_8s_7s_6 = rw-$) representan los permisos para el propietario del fichero, ALUMNO. Luego ALUMNO tiene permisos de lectura y escritura para el fichero `notas`. Además como $s_6 = -$, significa que ALUMNO no tiene permiso para ejecutar ese fichero y el bit `S_ISUID` no está activado.

Los siguientes tres caracteres, ($s_5s_4s_3 = r--$) representan los permisos para los miembros del grupo `users`. Como sólo aparece una `r` cualquier usuario que pertenezca al grupo `users` puede leer este fichero, pero no escribir en él o ejecutarlo y además el bit `S_ISGID` no está activado.

Los últimos tres caracteres, también ($s_2s_1s_0 = r--$), representan los permisos para cualquier otro usuario del sistema (diferentes del propietario o de los pertenecientes al grupo `users`). En este caso los demás usuarios pueden leer el fichero, pero no escribir en él o ejecutarlo y además el bit `S_ISVTX` no está activado.

◆

◆ Ejemplo 3.21:

A continuación, para varias máscaras simbólicas se van especificar su máscara octal y su significado.

- `- rwx r-x r-x`. Su máscara octal es `0755`. Se trata de un fichero regular. El propietario del fichero puede leer, escribir y ejecutar el fichero. Los usuarios pertenecientes al grupo del fichero, y todos los demás usuarios pueden leer y ejecutar el fichero. Además los bits `S_ISUID`, `S_ISGID` y `S_ISVTX` están desactivados.
- `- rw- --- ---`. Su máscara octal es `0600`. Se trata de un fichero regular. El propietario del fichero puede leer y escribir. Nadie más puede acceder al fichero. Además los bits `S_ISUID`, `S_ISGID` y `S_ISVTX` están desactivados.
- `- rwx rwx rwx`. Su máscara octal es `0777`. Se trata de un fichero regular. Todos los usuarios pueden leer, escribir y ejecutar el fichero. Además los bits `S_ISUID`, `S_ISGID` y `S_ISVTX` están desactivados.

- d) - `rws rws rwt`. Su máscara octal es `7777`. Se trata de un fichero regular. Todos los usuarios pueden leer, escribir y ejecutar el fichero. Además los bits `S_ISUID`, `S_ISGID` y `S_ISVTX` están activados.
- e) - `rwS rwS rwT`. Su máscara octal es `7666`. Se trata de un fichero regular. Todos los usuarios pueden leer y escribir el fichero, pero no pueden ejecutarlo. Además los bits `S_ISUID`, `S_ISGID` y `S_ISVTX` están activados.
- f) - `rws --S --T`. Su máscara octal es `7700`. Se trata de un fichero regular. El propietario del fichero pueden leer, escribir y ejecutar el fichero. El grupo y el resto de usuarios no pueden leer, escribir o ejecutar el fichero. Además el bit `S_ISUID` está activado y los bits `S_ISGID` y `S_ISVTX` están activados.



3.5.2 Configuración de la máscara de modo de un fichero

Para configurar la máscara de modo de un fichero, su propietario o el superusuario, pueden utilizar el comando `chmod`. Su sintaxis es:

```
chmod {u,g,o,a}{+,-}{r,w,x,s,t} <ficheros>
```

En el comando se indica a que usuarios afecta: usuario propietario (`u`), usuarios pertenecientes al mismo grupo que el usuario (`g`), otros usuarios (`o`), todos los usuarios (`a`). A continuación se especifica si se están añadiendo permisos (+) o quitándolos (-). Finalmente se especifica que tipo de permiso se hace referencia lectura (`r`), escritura (`w`) o ejecución (`x`). También se puede activar (+) o desactivar (-) los bits `S_ISUID`, `S_ISGID` y `S_ISVTX`, a los cuales se hace referencia mediante (`s`) para `S_ISUID` y `S_ISGID` y mediante (`t`) para `S_ISVTX`.

Otra posible sintaxis para el comando `chmod` es:

```
chmod [M3M2M1M0] <ficheros>
```

Donde `[M3M2M1M0]` es el modo del fichero expresado en octal

◆ Ejemplo 3.22:

Supóngase que en el directorio de trabajo actual existe el fichero `hola`. A continuación, se muestran como se pueden modificar la máscara de modo de este fichero mediante el uso del comando `chmod`:

<code>chmod a+r hola</code>	Da a todos los usuarios acceso al fichero <code>hola</code> .
<code>chmod +r hola</code>	Como en el ejemplo anterior. Si no se indica <code>a</code> , <code>u</code> , <code>g</code> ó bien <code>o</code> por defecto se toma <code>a</code> .
<code>chmod og-x hola</code>	Quita permisos de ejecución a todos los usuarios excepto al propietario.
<code>chmod ug+s hola</code>	Activa los bits <code>S_ISUID</code> y <code>S_ISGID</code> para el fichero <code>hola</code> .
<code>chmod g-s hola</code>	Desactiva el bit <code>S_ISGID</code> para el fichero <code>hola</code> .
<code>chmod o+s hola</code>	No hace nada.
<code>chmod 0777 hola</code>	Todos los usuarios tienen permiso de lectura, escritura y ejecución. Los bits <code>S_ISUID</code> , <code>S_ISGID</code> y <code>S_ISVTX</code> están desactivados.
<code>chmod 7777 hola</code>	Todos los usuarios tienen permiso de lectura, escritura y ejecución. Los bits <code>S_ISUID</code> , <code>S_ISGID</code> y <code>S_ISVTX</code> están activados.

◆

3.5.3 Consideraciones adicionales

El acceso a un fichero no depende únicamente de la máscara de modo de dicho fichero, sino que también depende de los permisos de acceso a los componentes de la ruta de acceso del fichero. Por ejemplo, aunque un fichero tenga la siguiente máscara de modo simbólica `-rwxrwxrwx`, los usuarios no podrán acceder a él a menos que también tengan permiso de lectura y ejecución para el directorio en el cual se encuentra el fichero. Por tanto, si un usuario desea prohibir al resto de usuarios el acceso a todos sus ficheros ubicados en su directorio de trabajo actual, no necesita preocuparse de los permisos individuales de cada uno de sus ficheros, le bastaría con configurar la máscara de modo simbólica de su directorio de trabajo a `-rwx-----`.

3.6 CONTROL DE TAREAS

Cada proceso que es ejecutado por un usuario supone una tarea para el sistema. El control de tareas es una utilidad incluida en muchos intérpretes de comandos que permite controlar el estado de las diferentes tareas que se están ejecutando en el sistema.

En muchos casos, los usuarios sólo ejecutan una tarea cada vez, que es el último comando tecleado. Sin embargo, usando el control de tareas, se pueden ejecutar diferentes tareas al mismo tiempo, cambiando entre cada una de ellas conforme se necesite.

3.6.1 Visualización de los procesos en ejecución

El comando `ps` muestra por pantalla la lista de procesos que el usuario está ejecutando actualmente. Si se invoca con la opción `-aux`, es decir, `ps -aux`, se mostrará además la utilización del procesador y de la memoria.

♦ Ejemplo 3.23:

Supóngase que la ejecución de la orden

```
/home/ALUMNO$ ps
```

genera la siguiente respuesta en la pantalla:

PID	TT	STAT	TIME	COMMAND
124	3	S	10:03	(bash)
61	3	R	10:00	ps

La principal columna (`PID`) indica el identificador del proceso (*pid*), que es un número entero positivo único que el sistema asigna a cada proceso existente en el sistema. La segunda columna (`TT`) indica el número del terminal. La tercera columna (`STAT`) indica el estado del proceso. La cuarta columna (`TIME`) indica la hora en que el proceso entró en dicho estado. Finalmente la quinta columna (`COMMAND`) indica el nombre del proceso.

Se observa que el usuario ALUMNO está ejecutando dos procesos: el intérprete de comandos `bash` cuyo *pid* es 124 que se encuentra en el estado dormido (Sleeping) y el propio comando `ps` cuyo *pid* es 61 que se encuentra en el estado preparado para ejecución (Runnable).

♦

Por otra parte el comando `top` muestra la ocupación en tiempo real que hacen los procesos del sistema, se actualiza cada cinco segundos. Finalmente otro comando útil es `jobs` que permite chequear el estado de un proceso.

3.6.2 Primer plano y segundo plano

Un proceso puede estar en *primer plano* o en *segundo plano*. Solo puede haber un proceso en primer plano al mismo tiempo. El proceso que está en primer plano, es el que interactúa con el usuario, recibe entradas de teclado y envía las salidas al monitor. El proceso en segundo plano, no recibe ninguna señal desde el teclado por lo general, se ejecutan en silencio sin necesidad de interacción. Para ejecutar una tarea en segundo plano basta con añadir el carácter ‘&’ al final de la orden.

Por ejemplo compilar programas y comprimir un fichero grande son tareas que se pueden enviar al segundo plano, para dejar el ordenador en condiciones de ejecutar otro programa.

Los procesos también pueden ser suspendidos. Un proceso suspendido es aquel que no se está ejecutando actualmente, sino que está temporalmente parado. Después de suspender una tarea, se puede indicar a la misma que continúe, en primer plano o en segundo, según se necesite. Retomar una tarea suspendida no cambia en nada el estado de la misma, continuará ejecutándose justo donde se dejó.

Hay que tener en cuenta que suspender un trabajo no es lo mismo que interrumpirlo. Cuando se interrumpe un proceso, generalmente pulsando `ctrl+c`, el proceso muere, y deja de estar en memoria y de utilizar recursos del ordenador. Una vez eliminado, el proceso no puede continuar ejecutándose, y deberá ser lanzado otra vez para volver a realizar sus tareas.

Hay una gran diferencia entre una tarea que se encuentra en segundo plano, y una que se encuentra detenida. Una tarea detenida es una tarea que no se está ejecutando, es decir, que no usa tiempo de CPU, y que no está haciendo ningún trabajo (la tarea aún ocupa un lugar en memoria, aunque puede ser intercambiada a disco). Una tarea en segundo plano, se está ejecutando, y usando memoria, a la vez que completando alguna acción mientras el usuario hace otro trabajo.

♦ Ejemplo 3.24:

La orden

```
/home/ALUMNO$ yes
```

es un comando aparentemente inútil que envía una serie interminable de `yes` a la salida estándar. La serie de `yes` continuará hasta el infinito, a no ser que se interrumpa pulsando `ctrl+c`.

También se puede redirigir la salida estándar de `y` hacia `/dev/null`, que es una especie de agujero negro para los datos, todo lo que se envía allí, desaparece. Para ello hay que escribir la orden:

```
/home/ALUMNO$ yes > /dev/null
```

De esta manera, la pantalla ya no se llenará de `y`, pero el marcador del intérprete sigue sin estar disponible para poder introducir otra orden. Esto es así, porque `yes` sigue ejecutándose en primer plano y enviando esos inútiles `y` a `/dev/null`. Para interrumpir su ejecución se debe pulsar `ctrl+c`.

Si se desea enviar la salida estándar de `yes` hacia `/dev/null` y además tener la línea de ordenes disponible para ejecutar nuevas ordenes, entonces se debe ejecutar la siguiente orden:

```
/home/ALUMNO$ yes > /dev/null &
```

Nótese que es el carácter `'&'` al final de la orden lo que indica al intérprete que debe ejecutarla en segundo plano. Se generaría (por ejemplo) la siguiente respuesta en la pantalla:

```
[1] 324
```

```
/home/ALUMNO$
```

En la primera línea, `[1]` representa el número de tarea del proceso `yes`. El intérprete asigna un número a cada tarea que está ejecutando, como `yes` es el único comando que se está ejecutando, se le asigna el número de tarea 1. Por su parte 324 es el *pid* del proceso

Ahora se tiene el proceso `yes` ejecutándose en segundo plano. Para chequear el estado del proceso, se puede escribir la siguiente orden:

```
/home/ALUMNO$ jobs
```

que mostraría la siguiente respuesta en la pantalla

```
[1]+  Running                  yes >/dev/null    &
```

Esta línea está indicando que la tarea 1 asociada a la ejecución del comando `yes >/dev/null` se encuentra actualmente ejecutándose (`Running`) en segundo plano (`&`).

◆

Existe otra forma de poner una tarea en segundo plano: se puede lanzar la tarea en primer plano, pararla, y después relanzarla en segundo plano.

Para relanzar una tarea en primer plano, se usa el comando `fg`. Mientras que para relanzar en segundo plano se usa el comando `bg`.

♦ **Ejemplo 3.25:**

La orden

```
/home/ALUMNO$ yes > /dev/null
```

lanza la salida estándar de `yes` hacia `/dev/null`. Dado que `yes` corre en primer plano, no aparece disponible el marcador del intérprete de comandos para escribir nuevas órdenes.

Para suspender la tarea se pulsa la combinación de teclas `ctrl+z`. En pantalla aparece el mensaje

```
[1]+  Stopped      yes >/dev/null
```

Esta línea está indicando que la tarea 1 asociada a la ejecución del comando `yes >/dev/null` ha sido parada o suspendida (`Stopped`).

Si ahora se escribe la orden

```
/home/ALUMNO$ bg
```

en pantalla aparece el mensaje

```
[1]+  yes >/dev/null &
```

Esta línea está indicando que la tarea 1 asociada a la ejecución del comando `yes >/dev/null` ha sido relanzada en segundo plano.

♦

3.6.3 Eliminación de procesos

Para eliminar un proceso, se utiliza el comando `kill`. Este comando toma como argumento el *pid* del proceso o el número de tarea que dicho proceso tiene asignado.

♦ **Ejemplo 3.26:**

Supóngase que un usuario está ejecutando en segundo plano el comando `yes >/dev/null`, cuyo número de tarea es 1, y cuyo *pid* es 324.

La orden

```
/home/ALUMNO$ kill %1
```

finalizaría la tarea 1 del ejemplo anterior. Esto se puede comprobar ejecutando la orden

```
/home/ALUMNO$ jobs
```

En pantalla aparecería la siguiente respuesta:

```
[1]+  Terminated          yes >/dev/null
```

Esta línea está indicando que la tarea 1 asociada a la ejecución del comando `yes >/dev/null` ha finalizado (`Terminated`). De hecho una nueva invocación del comando `jobs` no mostraría ninguna respuesta por pantalla.

Por otra parte, también se podría eliminar la tarea usando el *pid* del proceso al que está asociada. En este ejemplo el *pid* de proceso es 324, así que la orden

```
/home/ALUMNO$ kill 324
```

es equivalente a

```
/home/ALUMNO$ kill %1
```

Obsérvese que no es necesario usar el carácter `'%'` cuando se usa el comando `kill` con el *pid* de un proceso.



TEMA 4

ESTRUCTURACION DE LOS PROCESOS EN UNIX

4.1 INTRODUCCION

Cuando se compila un programa, el compilador suponiendo que dicho programa va a ser el único que se va a ejecutar en el sistema genera un espacio o conjunto de direcciones de memoria virtual asociadas a dicho programa. Este espacio es traducido por la máquina a un conjunto de direcciones de memoria principal. De esta forma, varias copias de un mismo programa pueden coexistir en memoria principal, todas ellas utilizarán las mismas direcciones virtuales, sin embargo tendrán asignadas diferentes direcciones físicas.

Una *región* es un subconjunto o área de direcciones contiguas de memoria virtual. En cualquier programa se pueden distinguir al menos tres regiones: la región de código o texto, la región de datos y la región de pila.

Un *proceso* es una instancia de un programa en ejecución. Consiste en un conjunto de bytes que la CPU interpreta como código (instrucciones máquina), datos o elementos de una pila. En un sistema UNIX los procesos parecen ejecutarse de forma simultánea, aunque en un determinado instante de tiempo, realmente sólo uno de ellos estará ejecutándose en la CPU. Asimismo pueden existir simultáneamente en el sistema varias instancias de un mismo programa.

Desde un punto de vista práctico, un *proceso* es una entidad que se crea con la llamada al sistema `fork`, el proceso que invoca a esta llamada se denomina *proceso padre* y el proceso que se crea como resultado de la llamada `fork` se denomina *proceso hijo*.

Los procesos son unidades *funcionalmente independientes* ya que se debe tener en cuenta que un proceso no puede ejecutar instrucciones de otro proceso. Un proceso puede leer y escribir en sus regiones de datos y de pila, pero no puede leer ni escribir en las regiones de datos y de pila de otros procesos. Evidentemente ante esta situación se hace necesario implementar mecanismos de comunicación entre los procesos, materia que será objeto de estudio en el Tema 7.

Puesto que UNIX es un sistema multitarea y multiusuario, el núcleo asigna a cada proceso varios identificadores numéricos con el fin de llevar un control de los procesos que se están ejecutando en el sistema y saber a que usuarios pertenecen. Asimismo, el núcleo mantiene diferentes estructuras de datos asociadas a los procesos, las cuales son fundamentales para la ejecución de los mismos.

De manera poco formal, pero muy gráfica, se puede afirmar que el *contexto de un proceso A* es una “fotografía” de los valores de ciertas posiciones de memoria asociados al proceso A y de los registros de la CPU. En determinadas circunstancias, el núcleo debe realizar un *cambio de contexto*, es decir, aplazar o finalizar la ejecución del proceso (A) y comenzar o continuar con la ejecución de otro proceso B. Asimismo, cuando se produce una interrupción, una llamada al sistema o un cambio de contexto el núcleo debe *salvar el contexto del proceso* (“tomar una fotografía”).

En este tema, se describe el espacio de direcciones virtuales, los identificadores y las estructuras de datos del núcleo asociadas a un proceso. Asimismo, se analizan los diferentes elementos que constituyen el contexto de un proceso. Además se estudia cómo se salva el contexto de un proceso y cómo se realiza un cambio de contexto. También se describe el tratamiento de las interrupciones por parte del núcleo y el interfaz de las llamadas al sistema. La parte final del tema se dedica a enumerar y describir los posibles estados de un proceso, haciéndose especial hincapié en el estado dormido.

En la explicación de este tema se va a tomar como referencia principalmente el núcleo de una distribución clásica como SVR3. Las variantes modernas de UNIX tales como SVR4, OSF/1, BSD4.4 y Solaris 2.x (y superiores) presentan ciertas diferencias con respecto a este modelo clásico.

4.2 ESPACIO DE DIRECCIONES DE MEMORIA VIRTUAL ASOCIADO A UN PROCESO

4.2.1 Formato lógico de un archivo ejecutable

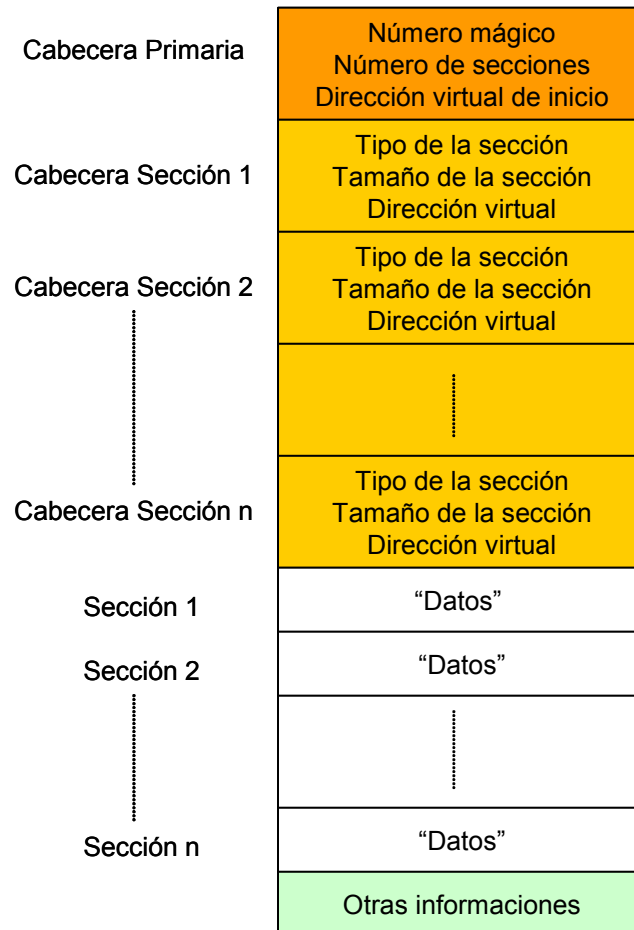


Figura 4.1: Estructura de un archivo ejecutable

Al compilar el código fuente de un programa se crea un archivo ejecutable que consta básicamente de cuatro partes (ver Figura 4.1):

1) *Cabecera primaria*. Contiene la siguiente información:

- El *número mágico*. Es un entero pequeño que permite identificar al núcleo el tipo de archivo ejecutable.
- El *número de secciones* que hay en el archivo,
- La dirección virtual de inicio, imprescindible para comenzar con la ejecución del proceso.

- 2) *Cabeceras de las secciones*. Describen cada una de las secciones del archivo. Especifican el tipo y el tamaño de la sección, además de la dirección virtual que se le debe asignar a la región cuando el proceso se ejecute en el sistema, y otras informaciones.
- 3) *Secciones*. Contienen los “datos”, que son cargados inicialmente en el espacio de direcciones del proceso, típicamente, el código (también denominado *texto*), los datos inicializados (variables estáticas y externas del programa conocidas en el momento de la compilación) y los datos no inicializados (también denominado *bss*¹).
- 4) *Otras informaciones*. Tales como la tabla de símbolos y otros “datos”. La tabla de símbolos es una tabla que se utiliza para almacenar los nombres definidos por el usuario en el programa fuente: variables, nombres de funciones, nombres de tipos, constantes, etc. Normalmente un compilador, debe comprobar, por ejemplo, que no se utiliza una variable sin haberla declarado previamente, o que no se declara una variable dos veces. Para ello, el compilador tiene que almacenar el nombre de la variable (y posiblemente su tipo y algún otro dato) en la tabla de símbolos y, cuando se utiliza esta variable en una expresión, el compilador la busca en la tabla para comprobar que existe y además para obtener información acerca de ella: tipo, dirección de memoria, etc. La información que se guarda en esta tabla depende del tipo de símbolo de que se trate. Lo habitual (excepto en compiladores muy sencillos) es implementar la tabla de símbolos utilizando una tabla de dispersión (hash) para optimizar el tiempo de búsqueda.

4.2.2 Regiones de un proceso

El núcleo carga un fichero ejecutable en memoria principal durante la realización, por ejemplo, de una llamada al sistema `exec`. El proceso cargado tiene asignado por el compilador un *espacio de direcciones de memoria virtual*, también denominado *espacio de direcciones de usuario*. Este espacio se divide en varias regiones, cada una de las cuales delimita un área de direcciones contiguas de memoria virtual. El espacio de direcciones de memoria virtual de un proceso consiste al menos de tres regiones: *la región de código* (o *texto*), *la región de datos* y *la región de pila*. Adicionalmente, puede

¹ *Bss* es el acrónimo del término inglés *block started by symbol* cuya traducción al castellano es *bloque inicializado con símbolos*.

contener *regiones de memoria compartida*, que posibilitan la comunicación de un proceso con otros procesos.

Las *regiones de código y de datos* se corresponden con las secciones de código y datos del fichero ejecutable. La *región de datos inicializados* o *zona estática de la región de datos* es de tamaño fijo. Por el contrario el tamaño de la *región de datos no inicializados* o *zona dinámica de la región de datos* puede variar durante la ejecución de un proceso.

La *región de pila* o *pila de usuario* se crea automáticamente y su tamaño es ajustado dinámicamente en tiempo de ejecución por el núcleo. La ejecución del código del programa irá marcando el crecimiento o decrecimiento de la pila, el núcleo asignará espacio para la pila conforme se vaya necesitando. La pila está constituida por *marcos de pila lógicos*. Un marco se añade a la pila cuando se llama a una función y se extrae cuando se vuelve de la misma. Existe un registro especial de la máquina denominado, *puntero de la pila* donde se almacena la dirección, dependiendo de la arquitectura de la máquina, de la próxima entrada libre o a la última utilizada. Análogamente, la máquina indica la dirección de crecimiento de la pila, hacia las direcciones altas o bajas. Un marco de pila contiene usualmente la siguiente información: los parámetros de la función, sus variables locales, y las direcciones almacenadas en el instante de la llamada a la función en diferentes registros especiales de la máquina, como por ejemplo, el contador del programa y el puntero de la pila. Salvar el contenido del contador del programa permite conocer la dirección de retorno donde debe continuar la ejecución una vez que se ha ejecutado la función. Mientras que salvar el contenido del registro de pila permite conocer la ubicación del marco de pila anterior o del siguiente libre.

♦ Ejemplo 4.1:

En la Figura 4.2 se representa a modo de ejemplo un diagrama del espacio de direcciones de memoria virtual de un proceso. Se observa que el proceso posee tres regiones: código, datos y pila. La dirección virtual de inicio de la región (DIR_{V0}) de código es $DIR_{V0}=0$ K. Por su parte la región de datos comienza en $DIR_{V0}=64$ K para su zona estática y $DIR_{V0}=128$ K para su zona dinámica. Finalmente, la región de pila o pila de usuario comienza en $DIR_{V0}=256$ K.

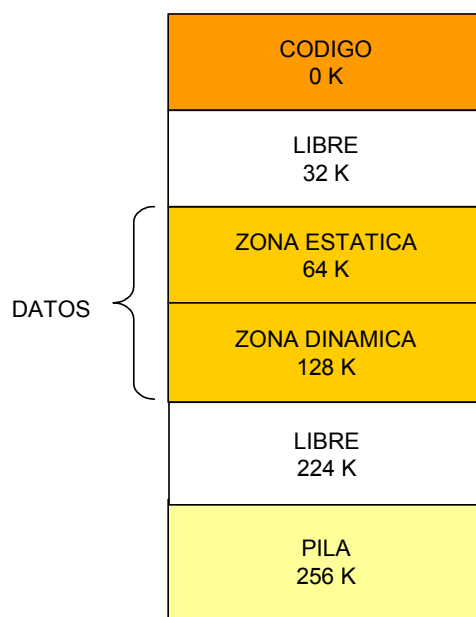


Figura 4.2: Diagrama del espacio de direcciones de memoria virtual de un proceso

Además se observa la existencia de dos regiones de direcciones de memoria virtual libre (no asignada) la primera comienza en $DIR_{V0}=32$ K y la segunda en $DIR_{V0}=224$ K.

♦

Además de las regiones descritas, existe otra parte del espacio de direcciones virtuales de un proceso denominada *espacio del núcleo* que se utiliza para ubicar estructuras de datos relativas a dicho proceso que son utilizadas por el núcleo. Existen básicamente dos estructuras locales a un proceso que deben ser manipuladas por el núcleo y que se suelen implementar en el espacio de direcciones del proceso: el *área de usuario o área U* y la *pila del núcleo*. Conceptualmente ambas estructuras aunque locales a un proceso son propiedad del núcleo. Obviamente, estas estructuras sólo pueden ser accedidas en modo núcleo o supervisor.

4.2.3 Operaciones con regiones implementadas por el núcleo

El núcleo dispone de una estructura local asociada a cada proceso denominada *tabla de regiones por proceso*, que mantiene información relevante sobre las regiones de código, datos, pila de usuario y memoria compartida (si existe) de un cierto proceso. Cada entrada de esta tabla contiene un puntero que apunta a una entrada de la *tabla de regiones*. Ésta es una estructura global del núcleo que contiene información sobre cada región. Las entradas de la tabla de regiones se organizan en dos listas: una *lista enlazada de regiones libres* y una *lista enlazada de regiones activas*. Simultáneamente una entrada sólo puede pertenecer a una de las dos listas.

Existen varias llamadas al sistema (`fork`, `exec`, ...) que tienen que manipular durante su ejecución el espacio de direcciones virtuales de un proceso. El núcleo dispone de algoritmos bien definidos para la realización de diferentes operaciones con las regiones. Las principales operaciones con regiones implementadas por el núcleo son:

- *Bloquear y desbloquear una región.* El núcleo debe bloquear una región para prevenir los posibles accesos de otros procesos mientras se encuentra trabajando con ella. Cuando termina de usarla la desbloquea. Estas operaciones son independientes de las operaciones de asignar y liberar una región.
- *Asignar una región.* Consiste en eliminar la primera entrada disponible de la lista enlazada de regiones libres y situarla en la lista enlazada de regiones activas. El núcleo implementa esta operación con el algoritmo `allocreg()`². Las llamadas al sistema que usan esta operación son: `fork`, `exec` y `shmget`.
- *Ligar una región al espacio de direcciones virtuales de un proceso.* Consiste en asociar a una región (que previamente ha tenido que ser asignada) una entrada de la tabla de regiones por proceso. El núcleo implementa esta operación con el algoritmo `attachreg()`. Las llamadas al sistema que usan esta operación son: `fork`, `exec` y `shmat`.
- *Cambiar el tamaño de una región.* Las únicas regiones cuyo tamaño pueden ser modificados son las regiones de dato y de pila. Las regiones de código y las regiones de memoria compartida no pueden crecer después de ser inicializadas. El núcleo implementa esta operación con el algoritmo `growreg()`. Existen dos llamadas al sistema `brk` y `sbrk` que usan esta operación, ambas trabajan con la región de datos. Además el núcleo también utiliza esta operación para implementar el crecimiento de la pila de usuario.
- *Cargar una región con el contenido de un fichero.* El núcleo implementa esta operación con el algoritmo `loadreg()` que es usada por la llamada al sistema `exec`.
- *Desligar una región del espacio de direcciones de un proceso.* El núcleo implementa esta operación con el algoritmo `detachreg()`. Las llamadas al sistema que usan esta operación son: `exec`, `exit` y `shmdt`.

² Los nombres de los algoritmos del núcleo que aparecen en este tema son los de la distribución SVR3.

- *Liberar una región.* Cuando una región ya no está unida a ningún proceso, el núcleo puede liberar la región y devolverla a la lista enlazada de regiones libres. El núcleo implementa esta operación con el algoritmo `freereg()`.
- *Duplicar una región.* El núcleo implementa esta operación con el algoritmo `dupreg()`, que es usado por la llamada al sistema `fork`.

4.3 IDENTIFICADORES NUMERICOS ASOCIADOS A UN PROCESO

4.3.1 Identificador del proceso

Puesto que UNIX es un sistema multitarea, necesita identificar de forma precisa a cada proceso que se está ejecutando en el sistema. La forma de identificación utilizada es asignar a cada proceso un número entero positivo distinto denominado *identificador del proceso* o *pid*. Luego los posibles valores de un pid son

$$pid = \{0, 1, 2, 3, \dots, pid_{\max}\}$$

donde pid_{\max} es el valor más grande que puede asignar el núcleo al *pid* de un proceso.

Cuando el sistema operativo arranca crea un proceso especial denominado *proceso 0* al que asigna un $pid=0$. Poco después el proceso 0 genera un proceso hijo, denominado *proceso inicial* cuyo $pid=1$, y se convierte en el *proceso intercambiador* (*swapper*). El proceso inicial es el responsable de arrancar al resto de procesos del sistema, y en consecuencia es el proceso padre de todos ellos. Si el núcleo necesita asignar un *pid* a un nuevo proceso y ya ha asignado el valor pid_{\max} entonces realiza una búsqueda de *pid* libres comenzando desde 0. Muchos procesos tienen un tiempo de ejecución muy corto, así que probablemente habrá muchos números *pid* sin utilizar.

La llamada al sistema `getpid` devuelve el *pid* del proceso que realiza esta llamada. Su sintaxis es:

```
salida=getpid();
```

Se observa que no requiere de ningún parámetro de entrada. Si la llamada al sistema se ejecuta con éxito `salida` tendrá asignado el valor del *pid* del proceso. En caso contrario contendrá el valor -1.

Asimismo la llamada al sistema `getppid` devuelve el *pid* del proceso padre del proceso que realiza esta llamada. Su sintaxis es análoga a la de `getpid`.

4.3.2 Identificadores de usuario y de grupo

Por otra parte, puesto que UNIX es un sistema multiusuario, el núcleo asocia a cada proceso dos identificadores enteros positivos de usuario y dos identificadores enteros positivos de grupo. Los identificadores de usuario son el *identificador de usuario real* (*uid*) y el *identificador de usuario efectivo* (*euid*). Mientras que para el grupo, se tiene el *identificador del grupo real* (*gid*) y el *identificador del grupo efectivo* (*egid*).

El *uid* identifica al usuario que es responsable de la ejecución del proceso y el *gid* identifica al grupo al cual pertenece dicho usuario. El *euid* se utiliza, principalmente, para determinar el propietario de los ficheros recién creados, para permitir el acceso a los ficheros de otros usuarios y para comprobar los permisos para enviar señales a otros procesos. El uso del *egid* es similar al del *euid* pero desde el punto de vista del grupo.

El núcleo reconoce un usuario privilegiado denominado *superusuario*, normalmente a este usuario privilegiado se le identifica con el nombre de *root*. El superusuario tiene asignados los valores *uid*=0 y *gid*=1.

Usualmente, el *uid* y el *euid* van a coincidir, pero si un usuario U1 ejecuta un programa P que pertenece a otro usuario U2 y que tiene activo el bit *S_ISUID* entonces el proceso asociado a la ejecución de P por parte de U1 va a cambiar su *euid* y va a tomar el valor del *uid* del usuario U2. Es decir, a efectos de comprobación de permisos sobre P, U1 va a tener los mismos permisos que tiene el usuario U2. Para el identificador de grupo efectivo *egid* se aplica la misma norma.

Las llamadas al sistema *getuid*, *geteuid*, *getgid* y *getegid* permiten determinar qué valores toman los identificadores *uid*, *euid*, *gid* y *egid*, respectivamente. Su sintaxis es similar a la de la llamada al sistema *getpid*.

Para cambiar los valores que toman estos identificadores, es posible utilizar las llamadas al sistema *setuid* y *setgid*. La sintaxis de la llamada al sistema *setuid* es:

```
salida = setuid (par);
```

La llamada al sistema *setuid* permite asignar el valor *par* al *euid* y al *uid* del proceso que invoca a la llamada. Se distinguen los siguientes casos:

- a) El identificador de usuario efectivo del proceso que efectúa la llamada es el del superusuario. En este caso *uid*=*par* y *euid*=*par*.

b) El identificador del usuario efectivo del proceso que efectúa la llamada `no` es el del superusuario. En este caso `euid=par` si se cumple alguna de las siguientes condiciones:

- El valor del parámetro `par` coincide con el valor del `uid` del proceso.
- Esta llamada se está invocando dentro de la ejecución de un programa que tiene su bit `S_ISUID` activado y el valor del parámetro `par` coincide con el valor del `uid` del propietario del programa.

Si la llamada se ejecuta con éxito entonces `salida` vale 0. Si se produce un error `salida` vale -1

La explicación del funcionamiento de la llamada al sistema `setguid` es análogo al explicado para `setuid` pero referido a los identificadores `gid` y `egid`.

♦ Ejemplo 4.2:

Un ejemplo típico de programa que usa las llamadas al sistema que se han estudiado en esta sección es el programa `login` para iniciar una sesión en el sistema. Este programa se ejecuta con el `euid` del superusuario (`root`). Después de preguntar el nombre de usuario y la contraseña, consulta en el fichero `/etc/passwd/` los valores de `uid` y `gid`, para hacer sendas llamadas a `setuid` y `setgid` y que los identificadores `uid`, `euid`, `gid`, `egid` pasen a ser los del usuario que quiere iniciar la sesión de trabajo. Luego llama a `exec` para ejecutar un intérprete de órdenes para que dé servicio al usuario. Este intérprete se va ejecutar con los identificadores de usuario y grupo, tanto reales como efectivos, de acuerdo con el usuario que ha iniciado su sesión.

Otro ejemplo es el programa ejecutable `passwd`, que permite a un usuario cambiar su contraseña de acceso al sistema. Este programa debe acceder al fichero `/etc/passwd/` que contiene entre otras informaciones las contraseñas de todos los usuarios del sistema. Para evitar que un usuario pueda cambiar las contraseñas de los demás usuarios, sólo está permitido el acceso a este fichero al superusuario. El ejecutable `passwd` es propiedad del superusuario, pero al tener su bit `S_ISUID` activado el usuario que lo ejecuta cambia su `euid` y se hace igual al del superusuario por lo que tendrá acceso al fichero `/etc/passwd/` al que debe acceder.

♦

♦ Ejemplo 4.3:

Supóngase que en un cierto directorio, se tienen los siguientes archivos:

- El programa ejecutable `ejemident`, cuyo código es el indicado en el Programa 4.1, que pertenece al usuario `USUARIO1`, este fichero tiene la siguiente máscara simbólica de permisos - `rws rwx rwx`, es decir, todos los usuarios pueden leer, escribir y ejecutar este archivo, además su bit `S_ISUID` se encuentra activado.
- El fichero de texto `fichero1.txt` que pertenece al usuario `USUARIO1`, este fichero tiene los siguientes permisos - `rw- --- ---`, es decir, únicamente el propietario del fichero puede leer y escribir en dicho fichero.
- El fichero de texto `fichero2.txt` que pertenece al usuario `USUARIO2`, este fichero tiene los siguientes permisos - `rw- --- ---`, es decir, únicamente el propietario del fichero puede leer y escribir en dicho fichero.

```
#include <fcntl.h>
main()
{
    int x, y;
    int fd1, fd2;
    x=getuid();
    y=geteuid();
    [1] printf("\nUID= %d, EUID= %d \n", x, y);
        fd1=open("fichero1.txt", O_RDONLY);
        fd2=open("fichero2.txt", O_RDONLY);
    [2] printf("fd1= %d, fd2= %d \n", fd1, fd2);
        setuid(x);
    [3] printf("Después del setuid(%d): UID= %d, EUID= %d
            \n", x, getuid(), geteuid());
        fd1=open("fichero1.txt", O_RDONLY);
        fd2=open("fichero2.txt", O_RDONLY);
    [4] printf("fd1= %d, fd2= %d \n", fd1, fd2);
        setuid(y);
    [5] printf("Después del setuid(%d): UID= %d, EUID= %d \n", y,
            getuid(), geteuid());
}
```

Programa 4.1

Supóngase además que `USUARIO1` tiene `uid=501` y que `USUARIO2` tiene `uid=503`. Se van a considerar tres casos:

* **CASO 1:** El USUARIO1 ejecuta `ejemident`, se obtiene la siguiente traza en pantalla:

```
[1] UID= 501, EUID= 501
[2] fd1= 3, fd2= -1
[3] Después del setuid(501): UID= 501, EUID= 501
[4] fd1= 4, fd2= -1
[5] Después del setuid(501): UID= 501, EUID= 501
```

Se observa que puesto que su *eid* se mantiene siempre con el valor 501, puede abrir `fichero1.txt` ($fd1 \neq -1$) ya que posee permiso de lectura y su *eid* coincide con el *uid* del propietario, que es él, o sea $eid=uid=501$. Sin embargo, no puede abrir `fichero2.txt` ($fd2 = -1$) ya que éste pertenece a USUARIO2 cuyo $uid=503$, es decir $eid \neq uid$.

* **CASO 2:** El USUARIO2 ejecuta el archivo `ejemident`, se obtiene la siguiente traza en pantalla:

```
[1] UID= 503, EUID= 501
[2] fd1= 3, fd2= -1
[3] Después del setuid(503): UID= 503, EUID= 503
[4] fd1= -1, fd2= 4
[5] Después del setuid(501): UID= 503, EUID= 501
```

Puesto que `programa2_5` tiene su bit `S_ISUID` activado, al ejecutarlo el USUARIO2 su *eid* se hace igual al *uid* del propietario de este fichero que recordé es USUARIO1. Luego en [1] $uid=503$ y $eid=501$. Al cambiar su *eid* ahora USUARIO2 puede abrir `fichero1.txt` pero no puede abrir `fichero2.txt`, pese al ser el propietario y tener permiso de lectura, ya que a efectos de permiso de apertura de fichero se trabaja con el *eid*, como se pone de manifiesto en [2]. Tras ejecutar la sentencia de `setuid(503)`, el USUARIO2 pasa a tener [3] su $eid=503$, con lo que ahora puede abrir `fichero2.txt` pero no `fichero1.txt`, como se pone de manifiesto en [4]. Tras ejecutar `setuid(501)`, el USUARIO2 pasa a tener [5] de nuevo su $eid=501$.

* **CASO 3:** El USUARIO2 ejecuta el archivo `programa2_5`, se supone ahora que su bit `S_ISUID` no está activado, es decir su máscara simbólica es `- rwx rwx rwx`, se obtiene la siguiente traza en pantalla:

```
[1] UID= 503, EUID= 503
[2] fd1= -1, fd2= 3
[3] Después del setuid(503): UID= 503, EUID= 503
[4] fd1= -1, fd2= 4
[5] Después del setuid(503): UID= 503, EUID= 503
```

Se observa que puesto que su *eid* se mantiene siempre con el valor 503, puede abrir `fichero2.txt` ($fd2 \neq -1$) ya que posee permiso de lectura y su *eid* coincide con el *uid* del propietario, que es él, o sea $eid=uid=503$. Sin embargo, no puede abrir `fichero1.txt` ($fd1 = -1$) ya que éste pertenece a USUARIO1 cuyo $uid=501$, es decir $eid \neq uid$.



4.4 ESTRUCTURAS DE DATOS DEL NÚCLEO ASOCIADAS A LOS PROCESOS

El núcleo mantiene diferentes estructuras de datos asociadas a los procesos, las cuales son imprescindibles para la ejecución de los mismos. Algunas de estas estructuras como la *pila del núcleo*, el *área U* y la *tabla de regiones por proceso* son locales a cada proceso, es decir, cada proceso tiene asignada su propia estructura privada. Otras estructuras, como la *tabla de procesos* y la *tabla de regiones* son globales para todos los procesos, es decir, sólo existe en el núcleo una estructura para todos los procesos.

Por otra parte, algunas de estas estructuras como la *pila del núcleo* y el *área U* se implementan en el espacio de direcciones virtuales de cada proceso. Mientras que otras como la *tabla de procesos*, la *tabla de regiones por proceso* y la *tabla de regiones* se implementan en el espacio de direcciones virtuales del núcleo. Todas estas estructuras tienen en común que son propiedad del núcleo y por tanto solo pueden ser accedidas en modo núcleo.

Otras estructuras de datos del núcleo asociadas a los procesos son las *tablas de páginas* y la *tabla de descriptores de ficheros*. Las tablas de páginas permiten traducir las direcciones de memoria virtual a direcciones de memoria física, (se estudiarán en el Tema 9). Por su parte, la *tabla de descriptores de ficheros* identifica a los ficheros abiertos por un proceso.

4.4.1 Pila del núcleo

La existencia en UNIX de dos modos distintos de ejecución, modo usuario y modo núcleo, hace necesario la existencia de una pila independiente para cada modo y para cada proceso: la *pila de usuario* y la *pila del núcleo*. La *pila de usuario* contiene los marcos de pila de las funciones que se ejecutan en modo usuario. De forma análoga, la *pila del núcleo* contiene los marcos de pila de las funciones que se ejecutan en modo núcleo. Por tanto, estas dos pilas crecerán de forma autónoma. De hecho la pila del núcleo está vacía cuando el proceso se ejecuta en modo usuario.

Por otra parte, puesto que pueden estar ejecutándose en paralelo varias instancias de una misma rutina del núcleo asociada cada una de ellas a un proceso distinto, se hace necesario la existencia de una pila del núcleo distinta para cada proceso. Por tanto, la *pila del núcleo* se puede definir como una estructura local a cada proceso que contiene los marcos de pila de las funciones o rutinas del núcleo invocadas durante la ejecución

del proceso en modo núcleo. Frecuentemente la pila del núcleo se implementa dentro del área U del proceso, pero también puede implementarse en un área de memoria independiente.

♦ **Ejemplo 4.4:**

```
[1] char buffer[2048];
[2] main(int argc, char *argv[])
{
[3]   int aviejo, anuevo;
[4]   if (argc!=3)
      {
[5]       printf("Error: El programa debe ser invocado con dos parametros
\n");
[6]       exit(1);
      }
[7]   aviejo=open(argv[1],0444);
[8]   if (aviejo == -1)
      {
[9]       printf("Error: No se puede abrir el archivo %s\n", argv[1]);
[10]      exit(1);
      }
[11]   anuevo=creat(argv[2],0666);
[12]   if (anuevo == -1)
      {
[13]       printf("Error: No se puede crear el archivo %s\n", argv[2]);
[14]       exit(1);
      }
[15]   copiar(aviejo,anuevo);
[16]   exit(0);
}

[17] copiar(int viejo, int nuevo)
{
[18]   int contador;
[19]   while ((contador = read(viejo,buffer,sizeof(buffer)))>0)
[20]       write(nuevo,buffer,contador);
}
```

Programa 4.2

Considérese el Programa 4.2 escrito en lenguaje C, el cual crea una copia del contenido de un fichero en otro fichero nuevo. Supóngase que el nombre del fichero ejecutable que se crea después de compilar este programa es `copfile`. Un usuario invocaría a este programa desde un terminal (\$) escribiendo:

```
$ copfile file_V file_N
```

donde `file_V` es el nombre de un fichero ya existente en el sistema que se desea copiar y `file_N` es el nombre del nuevo fichero.

El sistema invoca **[2]** a la función principal `main` suministrándole el número de parámetros `argc` en la lista `argv`, e inicializando a cada miembro del array `argv` para que apunte a un parámetro suministrado por el usuario. En la forma de invocación de este programa el número de parámetros es 3, `argv[0]` apunta al string de caracteres `copfile` (el nombre del programa es usualmente usado como parámetro 0), `argv[1]` apunta al array de caracteres `file_V` y `argv[2]` apunta al array de caracteres `file_N`.

En primer lugar **[4]** comprueba si ha sido invocado con un número erróneo de parámetros, es decir, si `argc` es distinto de 3. En caso afirmativo, imprime **[5]** en pantalla el mensaje

```
Error: El programa debe ser invocado con dos parámetros
```

e invoca **[6]** a la llamada al sistema `exit` para terminar la ejecución del programa. En caso negativo, invoca **[7]** a la llamada al sistema `open` para que abra con permisos de sólo lectura para todos los usuarios (máscara de modo octal 0444) el fichero `file_V`. Esta llamada devuelve un descriptor del fichero que es almacenado en la variable `aviejo`.

Si la llamada al sistema `open` falla **[8]**, es decir, `aviejo=-1`, entonces imprime **[9]** en pantalla el mensaje

```
Error: No se puede abrir el archivo file_V
```

e invoca **[10]** a la llamada al sistema `exit` para terminar la ejecución del programa. En caso contrario, invoca **[11]** a la llamada al sistema `creat` para crear el fichero `file_N`. con permisos de lectura y escritura para todos los usuarios (máscara de modo octal 0666) el fichero `file_N`. Esta llamada devuelve un *descriptor del fichero* y lo asocia a la variable `anuevo`.

Si la llamada al sistema falla **[12]**, es decir, `anuevo=-1`, entonces imprime **[13]** en pantalla el mensaje

```
Error: No se puede crear el archivo file_N
```

e invoca **[14]** a la llamada al sistema `exit` para terminar la ejecución del programa. En caso contrario el programa llama **[15]** a la función `copiar`, la cual entra en un lazo **[19]**, donde se invoca a la llamada al sistema `read` que lee un total de 2048 bytes procedentes del fichero `file_V` y los almacena en la zona de memoria asignada **[1]** al array de caracteres `buffer`.

A continuación invoca **[20]** a la llamada al sistema `write` para escribir los datos en el nuevo fichero. La llamada al sistema `read` devuelve el número de bytes leídos y los almacena en la

variable `contador`, devolviendo 0 cuando llega al final del fichero. El programa finaliza el lazo cuando encuentra el final del fichero o cuando hay algún error durante la llamada al sistema, es decir, `read` devuelve el valor -1 (obsérvese que el programa no comprueba la aparición de errores durante la llamada al sistema `write`). Entonces vuelve de la función `copiar` e invoca [16] a la llamada al sistema `exit` para terminar la ejecución del programa.

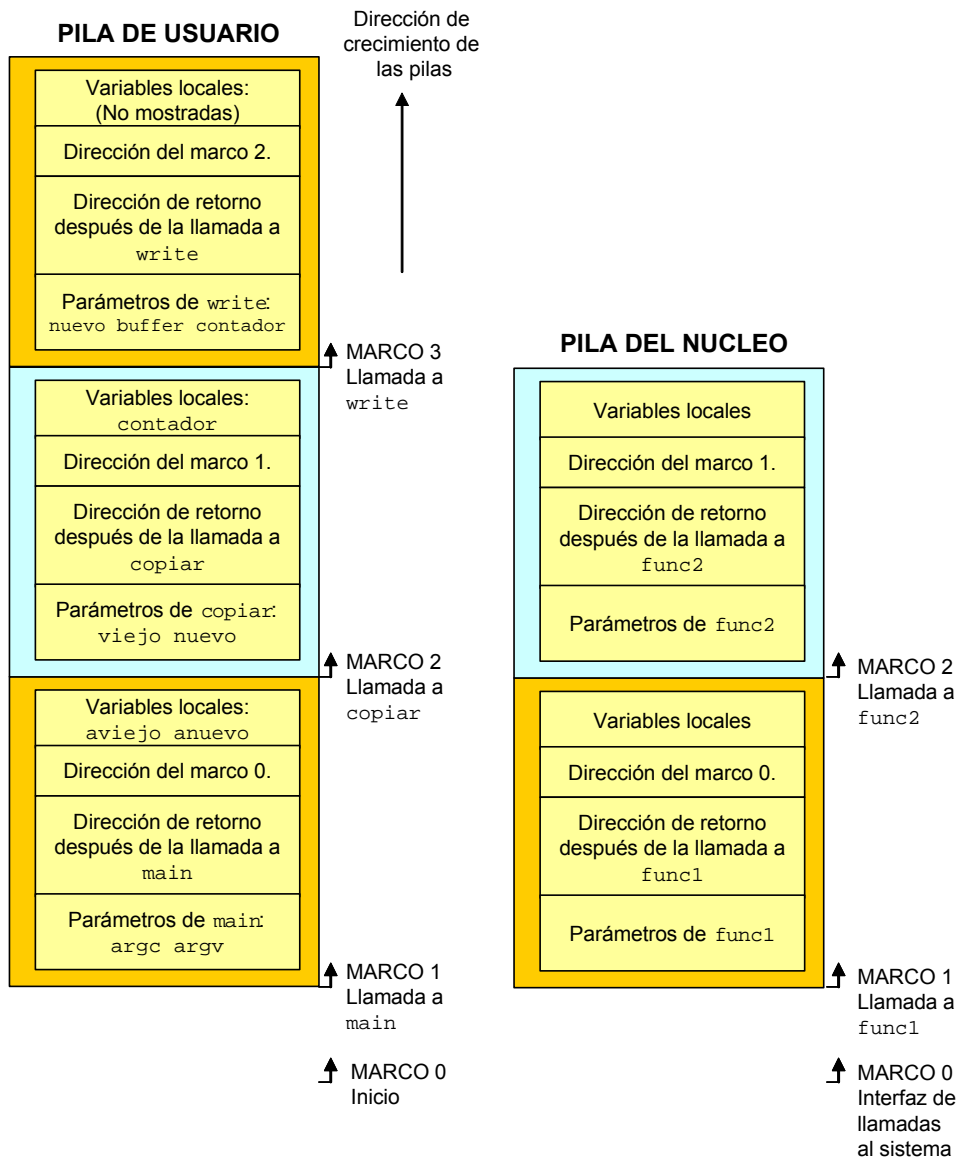


Figura 4.3: Pila del usuario y pila del núcleo en un cierto instante de la ejecución de la llamada al sistema `write`

En la Figura 4.3 se muestra un posible esquema de la *pila de usuario* y de la *pila del núcleo* en un cierto instante de la ejecución en un sistema UNIX del proceso asociado al archivo ejecutable `copfile`. En concreto, las pilas representadas corresponden al instante en que se está ejecutando la llamada al sistema `write` (sentencia [20]). Se observa que la *pila de usuario*

contiene tres marcos de pila: el primero asociado a la función `main`, el segundo a la función `copiar` (invocada dentro de la función `main`) y el tercero a la llamada al sistema `write` (invocada dentro de la función `copiar`). El marco 0 es un marco mudo que el núcleo utiliza para inicializar la pila.

Es importante resaltar que un proceso invoca a una llamada al sistema como si se tratase de una función cualquiera, lo que supone que se creará en su pila de usuario un marco de pila para dicha función. Si bien en una pila de usuario nunca podrán existir simultáneamente dos marcos de pila asociados a dos llamadas al sistema ya que hasta que no se invoque y se vuelva de la primera llamada no se podrá invocar a la segunda. De hecho, una pila de usuario en la que exista un marco de pila para una llamada al sistema no podrá crecer hasta que no se vuelva de dicha llamada al sistema.

Un proceso que invoca a una llamada al sistema en realidad está invocando a la función de librería asociada a dicha llamada al sistema, que entre sus diferentes instrucciones posee una interrupción software o trap que produce la conmutación hardware al modo núcleo. Se comienza a ejecutar código del núcleo, por lo que se utilizará la pila del núcleo asociada a dicho proceso.

Por ejemplo, supóngase que en un cierto instante de la ejecución de la llamada al sistema `write` se requiere ejecutar una cierta función del núcleo `func1`, y que ésta a su vez invoca a otra cierta función del núcleo `func2`. En consecuencia la *pila del núcleo* contendrá dos marcos de pila: el primero asociado a la función del núcleo `func1` y el segundo asociado a la función del núcleo `func2`.

Por otra parte se observa que cada marco de página asociado a una función, tanto en la pila de usuario como en la pila del núcleo contiene la siguiente información: parámetros de la función, variables locales de la función, dirección de retorno después de la ejecución de la función (es una copia del contenido del registro contador del programa en el momento de creación del marco) y dirección de comienzo del marco de pila anterior (es una copia del contenido del registro puntero de la pila en el momento de creación del marco).

◆

4.4.2 Tabla de procesos

La *tabla de procesos* es una estructura global del núcleo donde se almacena información de control relevante sobre cada proceso existente en el sistema. Cada entrada de la tabla de procesos contiene distintos campos con información sobre un determinado proceso, como por ejemplo:

- *El identificador del proceso (pid).*

- *Los identificadores de usuario (uid, euid) y de grupo (gid, egid) del proceso.*
- *Punteros que permiten al núcleo localizar la tabla de regiones por proceso y el área U del proceso.*
- *El estado del proceso.* Un proceso durante su tiempo de vida puede pasar por diferentes estados (se estudian en la sección 4.8), cada uno de los cuales posee ciertas características que determinan el comportamiento del proceso.
- *Punteros para enlazar a los procesos en diferentes listas que permiten al núcleo controlar a los procesos, como por ejemplo, la lista de procesos planificados para ejecución, la lista de procesos dormidos...*
- *Canal o dirección de dormir asociada al evento por el que el proceso ha entrado en el estado dormido.*
- *Información asociada a la prioridad de planificación del proceso que es consultada por el algoritmo de planificación del núcleo para determinar que proceso debe pasar a utilizar el procesador.*
- *Información asociada al tratamiento de las señales como por ejemplo, las máscaras de las señales que son ignoradas, bloqueadas, notificadas y tratadas.*
- *Información genealógica, que describe la relación de este proceso con otros procesos, como por ejemplo, el pid de su proceso padre, un puntero al primer hijo creado, un puntero al último hijo creado...*
- *El tiempo de ejecución del proceso y el tiempo de utilización de los recursos de la máquina.* Estas informaciones son usadas por el núcleo, entre otras cosas, para el cálculo del valor de la prioridad de planificación del proceso.

4.4.3 Area U

El *área de usuario* o *área U* es una estructura local asociada a cada proceso que contiene información de control relevante sobre el mismo que el núcleo necesita consultar únicamente cuando ejecuta dicho proceso. Entre la información contenida en los campos del área U se encuentra:

- Un *puntero* a la entrada de la tabla de procesos asociada a dicho proceso.

- Los *identificadores de usuario* (*uid*, *euid*) y *de grupo* (*gid*, *egid*) del proceso. No debe extrañar la aparición de esta información tanto en el *área U* como en la entrada de la *tabla de procesos* asociada al proceso. Sin entrar en detalles más precisos, comentar únicamente que a los *identificadores de usuario y de grupo* almacenados en el *área U* pueden en determinadas circunstancias diferir de los almacenados en la tabla de procesos.
- Los argumentos de entrada, los valores de retorno y el identificador del error (si se produjese) de la llamada al sistema en ejecución.
- Las direcciones de los manipuladores de las señales y otras informaciones relacionadas.
- Información acerca de las regiones de código, datos y pila obtenida de la cabecera del programa.
- La *tabla de descriptores de ficheros* que mantiene información sobre los ficheros abiertos por el proceso.
- El *directorio de trabajo actual* y el *directorio raíz actual*.
- La *terminal de acceso* asociada con el proceso, si existe alguna.
- Estadísticas de uso la CPU.

El *área U* de un proceso se puede considerar en ciertos aspectos como una extensión de la entrada asignada a dicho proceso en la *tabla de procesos*. Sin embargo, mientras que la información contenida en la *tabla de procesos* debe ser estar siempre accesible para el núcleo, el *área U* contiene información que necesita únicamente estar accesible para el núcleo cuando se está ejecutando el proceso.

Como se justificará en la sección 9.2.7, el núcleo puede acceder directamente a los campos del *área U* del proceso que se está ejecutando pero no al *área U* de otros procesos. Internamente, el núcleo referencia a una variable denominada *u* para acceder al *área U* del proceso (A) que actualmente se está ejecutando. Cuando se ejecuta otro proceso (B), el núcleo reorganiza su espacio de direcciones virtuales de forma que la variable *U* referencie al *área U* del nuevo proceso B. Esta implementación permite al núcleo identificar fácilmente al proceso actual siguiendo el campo puntero del *área U* que apunta a la correspondiente entrada de la tabla de procesos.

4.4.4 Tabla de regiones por proceso

La *tabla de regiones por proceso* es una estructura local a cada proceso que contiene una entrada por cada región (código, datos y pila de usuario). Si existen regiones de memoria compartida cada una de ellas también tendrá asignada una entrada.

Cada entrada de la *tabla de regiones por proceso* apunta a una entrada en la *tabla de regiones* y contiene la dirección virtual de comienzo de una región. Este nivel extra de direccionamiento (desde la *tabla de regiones por proceso* a la *tabla de regiones*) permite que procesos independientes puedan compartir regiones. Además cada entrada contiene el tipo de acceso permitido al proceso sobre dicha región: sólo lectura, lectura y escritura, o lectura y ejecución.

La *tabla de regiones por proceso* de un proceso puede implementarse dentro de la entrada de la tabla de procesos asociado a dicho proceso, o puede implementarse dentro del área U de dicho proceso. También puede implementarse en una área de memoria independiente.

4.4.5 Tabla de regiones

La *tabla de regiones* es una estructura global del núcleo que contiene una entrada por cada región asignada por el núcleo. Cada entrada de esta tabla contiene la información necesaria para describir una región, como por ejemplo:

- Un puntero al nodo-i del fichero cuyo contenido fue cargado dentro de la región.
- El tipo de región (código, datos, pila de usuario o memoria compartida).
- El tamaño de la región.
- La localización de la región en memoria principal, típicamente un puntero a una *tabla de páginas*.
- El estado de la región, que puede ser una combinación de: bloqueada, bajo demanda, en proceso de ser cargada en memoria y válida (cargada en memoria).
- El contador de referencia, que indica el número de procesos que están referenciando a una región.

Las entradas de la tabla de regiones se organizan en dos listas: una *lista enlazada de regiones libres* y una *lista enlazada de regiones activas*. Simultáneamente una entrada sólo puede pertenecer a una de las dos listas

♦ **Ejemplo 4.5:**

En la Figura 4.4 se representa un posible diagrama que relaciona a las estructuras de datos del núcleo asociadas a los procesos: *pila del núcleo*, *área U*, *tabla de procesos*, *tabla de regiones por proceso* y *tabla de regiones*. En concreto se han considerado dos procesos: el proceso A y el proceso B.

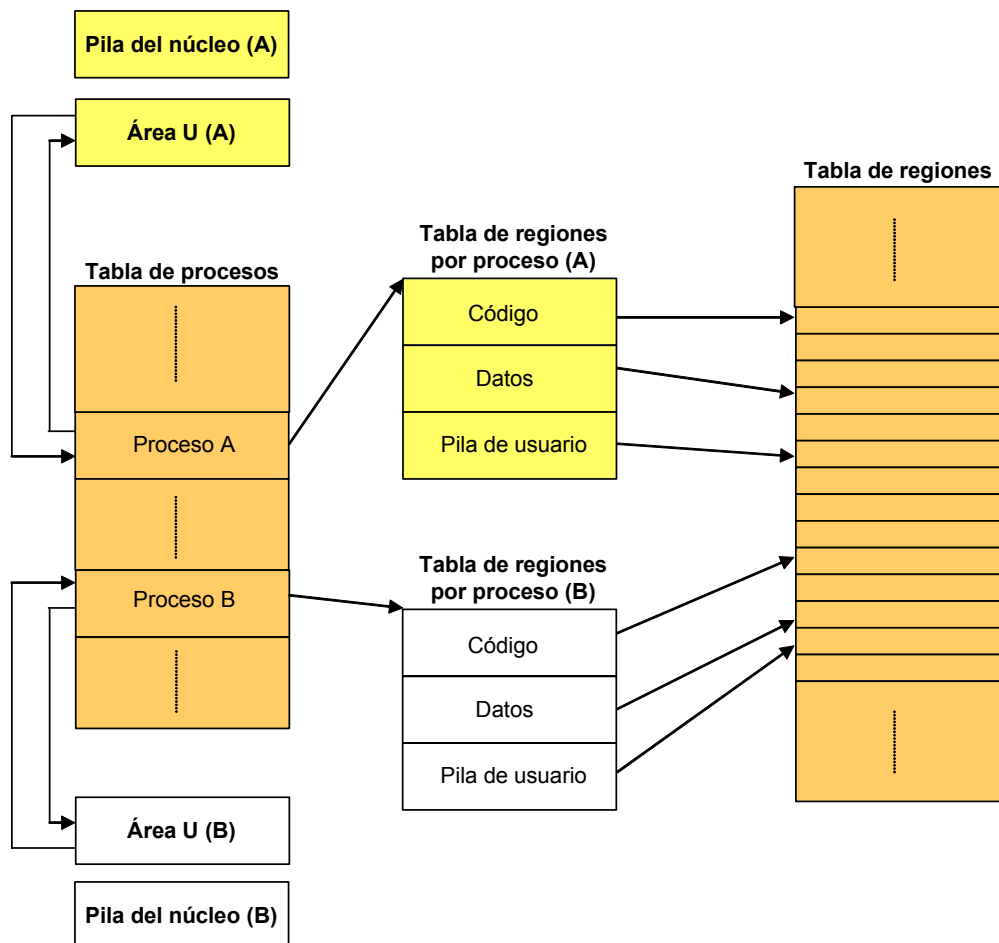


Figura 4.4. Estructura de datos del núcleo asociadas a los procesos A y B

En este diagrama se observa claramente el carácter local y global de las diferentes estructuras. Así el proceso A y el proceso B poseen cada uno de ellos su propia *pila del núcleo*, *área U* y *tabla de regiones por proceso*. Mientras que la *tabla de procesos* y la *tabla de regiones* son comunes para todos los procesos.

Asimismo, en este diagrama se observa claramente la relación existente entre estas estructuras. Por ejemplo, el *área U* del proceso A apunta a la entrada asociada a dicho proceso en la *tabla de*

procesos. A su vez dicha entrada posee un puntero tanto al *área U* como a la *tabla de regiones por proceso* asociadas al proceso A. Esta tabla posee tres entradas asociadas cada una de ellas a las regiones de código, datos y pila de usuario del proceso A. Cada una de las entradas de la *tabla de regiones por proceso* contiene un puntero a una entrada de la *tabla de regiones*. Relaciones análogas se aprecian para el proceso B.

Además, se observa en la tabla de regiones que el proceso A y el proceso B no tienen ninguna región común. Por lo tanto, el contador de referencias de estas regiones estará en 1, ya que sólo un proceso, el A o el B, está referenciando a través de una entrada de su tabla de regiones por proceso a cada región.

◆

4.5 CONTEXTO DE UN PROCESO

4.5.1 Definición

De forma general, el *contexto de un proceso* en un cierto instante de tiempo se puede definir como la información relativa al proceso que el núcleo debe conocer para poder iniciar o continuar su ejecución. Cuando se ejecuta un proceso se dice que el sistema se ejecuta en el contexto de dicho proceso. Por lo que cuando el núcleo decide pasar a ejecutar otro proceso debe de *cambiar de contexto*, de forma que el sistema pasará a ejecutarse en el contexto del nuevo proceso.

El *contexto de un proceso* en un cierto instante de tiempo está formado por su espacio de direcciones virtuales, los contenidos de los registros hardware de la máquina y las estructuras de datos del núcleo asociadas a dicho proceso. Formalmente, el *contexto de un proceso* se puede considerar como la unión del *contexto a nivel de usuario*, *contexto de registros* y *contexto a nivel del sistema*.

El *contexto a nivel de usuario* de un proceso está formado por su código, datos, pila de usuario y memoria compartida que ocupan el espacio de direcciones virtuales del proceso.

El *contexto de registros* de un proceso está formado por el contenido de los siguientes registros de la máquina:

- El *contador del programa* que indica la dirección de la siguiente instrucción que debe ejecutar la CPU. Esta dirección es una dirección virtual del espacio de memoria del núcleo o del usuario.

- El *registro de estado del procesador* que indica el estado del hardware de la máquina en relación al proceso en ejecución. Contiene diferentes campos para almacenar la siguiente información: el modo de ejecución, el nivel de prioridad de interrupción, el indicador de rebose, el indicador de arrastre, etc.
- El *puntero de la pila* donde se almacena la dirección virtual, dependiendo de la arquitectura de la máquina, de la próxima entrada libre o de la última utilizada en la pila de usuario (ejecución en modo usuario) o en la pila del núcleo (ejecución en modo núcleo). Análogamente, la máquina indica la dirección de crecimiento de la pila, hacia las direcciones altas o bajas.
- Los *registros de propósito general*, que contienen datos generados por el proceso durante su ejecución. Para simplificar la discusión, se van a considerar sólo dos registros, el registro 0 y el registro 1.

El *contexto a nivel del sistema* de un proceso está formado por: la entrada de la *tabla de procesos* asociada a dicho proceso, su *área U*, su *pila del núcleo*, su *tabla de regiones por proceso*, las entradas de la *tabla de regiones* apuntadas por las entradas de su tabla de regiones por proceso y las *tablas de páginas* asociadas a dichas entradas de la tabla de regiones.

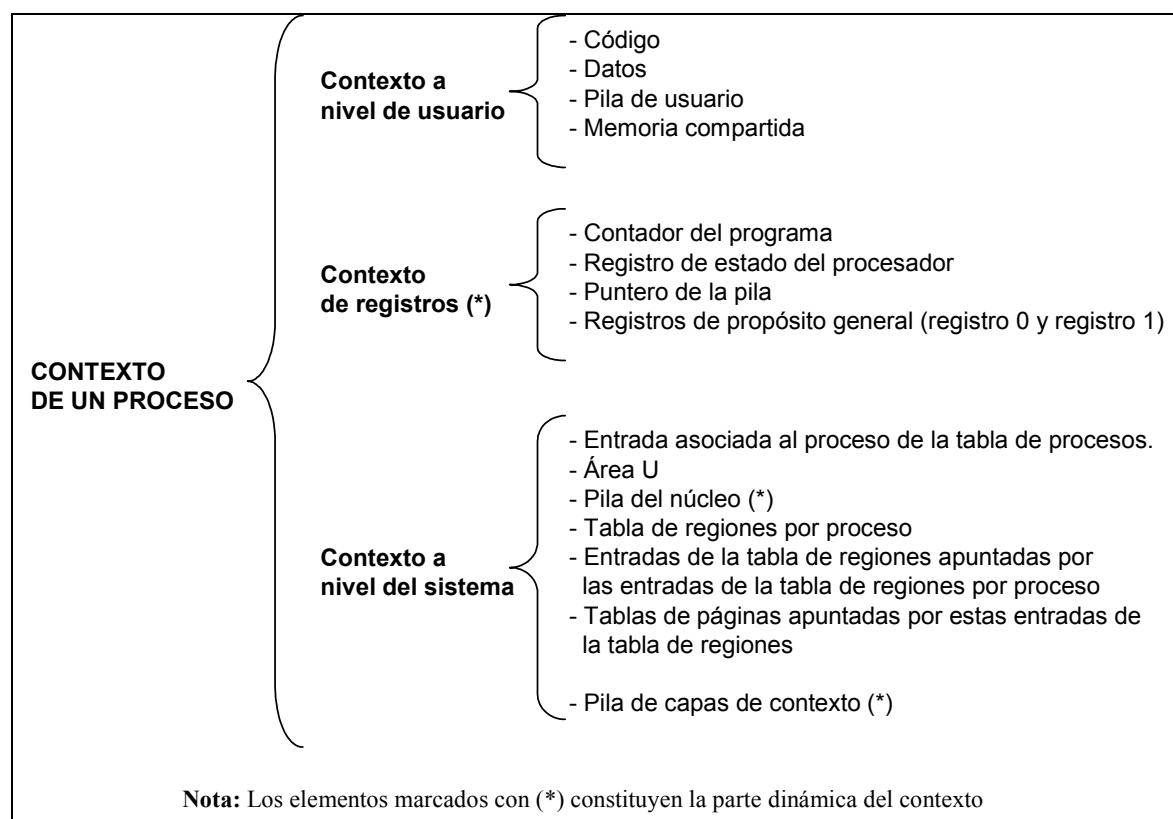
4.5.2 Parte estática y parte dinámica del contexto de un proceso

Durante el tiempo de vida de un proceso pueden producirse distintos sucesos, como por ejemplo: llamadas al sistema, interrupciones, cambios de contexto... Al atenderse a estos sucesos algunos elementos del contexto del proceso van a cambiar su contenido. Se hace necesario por tanto almacenar el contenido de estos elementos para que una vez atendido el suceso, y si no existe otro suceso pendiente, continuar con la ejecución en modo usuario del proceso.

Por tanto, en el contexto de un proceso se distinguen una *parte dinámica*, cuyo contenido es necesario salvar ante la aparición de ciertos sucesos, y una *parte estática*, cuyo contenido no es necesario salvar. La *parte dinámica* de un proceso está formado por el *contexto a nivel de registros* y su *pila del núcleo*. Los restantes elementos del contexto de un proceso constituyen su *parte estática*.

A la parte dinámica del contexto de un proceso que ha sido salvada se le denomina *capa de contexto*. Así durante el tiempo de vida de un proceso dependiendo de la secuencia de sucesos que se produzcan pueden existir varias capas de contexto. La

manipulación que el núcleo realiza de las capas de contexto puede visualizarse como una pila, denominada *pila de capas de contexto*. Esta pila es local a cada proceso, es decir, cada proceso tiene su propia *pila de capas de contexto*. La *pila de capas de contexto* también se considera como un elemento de la parte dinámica del contexto de un proceso, en concreto, del contexto a nivel del sistema. En el Cuadro 4.1 se resume la información que forma parte del contexto de un proceso.



Cuadro 4.1: Contexto de un proceso

El núcleo almacena en la entrada de la *tabla de procesos* asociada al proceso en ejecución la información necesaria para localizar la capa de contexto superior de la pila de capas de contexto asociada al proceso. De esta forma el núcleo conoce donde debe almacenar una nueva capa de contexto o donde debe buscar la última capa de contexto almacenada.

El núcleo añade una capa de contexto en la *pila de capas de contexto* de un proceso en los siguientes casos:

- Cuando se produce una interrupción.
- Cuando el proceso realiza una llamada al sistema.

- Cuando se produce un cambio de contexto.

Asimismo, el núcleo extrae una capa de contexto de la *pila de capas de contexto* de un proceso cuando:

- El núcleo vuelve de manipular una interrupción.
- El proceso vuelve al modo usuario después de que el núcleo completa la ejecución de una llamada al sistema.
- Se produce un cambio de contexto.

Se observa por tanto, que la realización de un cambio de contexto (se estaba ejecutando un proceso A y se pasa a ejecutar otro proceso B) provoca tanto la acción de añadir una capa de contexto (en la pila de capas de contexto del proceso A) como la de extraer una capa de contexto (en la pila de capas de contexto del proceso B).

El número de capas de contexto de la parte dinámica está limitado por el número de niveles de interrupción que soporte la máquina. Por ejemplo, supóngase que una máquina soporta seis niveles de interrupción (ver Figura 2.2). En este caso, la *pila de capas de contexto* de un proceso podrá contener como máximo ocho capas de contexto: una para cada nivel de interrupción, una para las llamadas al sistema y otra más para mantener el nivel de usuario. Estas ocho capas son suficientes para mantener a todas las interrupciones aunque las interrupciones ocurran en la peor secuencia posible, ya que una interrupción dada estará bloqueada mientras el núcleo manipula interrupciones de ese nivel o superior.

♦ Ejemplo 4.6:

En el siguiente ejemplo se va a usar la siguiente notación:

- RE contexto a nivel de registros del proceso A.
- PN pila del núcleo asociada al proceso A.
- RE_{ti} contenido de RE en un cierto instante de tiempo ti .
- PN_{ti} contenido (marcos de pila) de PN en un cierto instante de tiempo ti .
- PCC pila de capas de contexto asociada al proceso A.

Supóngase que un proceso A se está ejecutando en modo usuario y que se produce la siguiente secuencia de sucesos:

- 1) En el instante de tiempo t_0 el proceso A invoca a una llamada al sistema, por lo que se cambia de modo usuario a modo núcleo y se comienza atender la llamada al sistema. En este caso se añade una capa de contexto en su PCC que estaba inicialmente vacía al estar el proceso ejecutándose en modo usuario. A dicha capa de contexto se la a denotar como *capa 0* y su contenido será únicamente RE_{t_0} ya que en modo usuario su PN está vacía. Esta capa contiene la información necesaria para poder continuar con la ejecución del proceso A en modo usuario una vez se haya atendido la llamada al sistema.
- 2) En el instante de tiempo t_1 mientras se está ejecutando la llamada al sistema llega una interrupción del disco duro que por su nivel de prioridad debe ser atendida. Por ello se detiene la ejecución de la llamada al sistema y se comienza a ejecutar la rutina de servicio o manipulador de la interrupción del disco duro. En este caso se añade una capa de contexto en su PCC. A dicha capa de contexto se la a denotar como *capa 1* y su contenido será RE_{t_1} y PN_{t_1} . Esta capa contiene la información necesaria para poder continuar con la ejecución de la llamada al sistema una vez sea atendida la interrupción del disco duro.
- 3) En el instante de tiempo t_2 mientras se está ejecutando la rutina de servicio del disco duro llega una interrupción del reloj que por su nivel de prioridad debe ser atendida. Por ello se detiene la ejecución de rutina de servicio del disco duro y se comienza a ejecutar la rutina de servicio del reloj. En este caso se añade una capa de contexto en su PCC. A dicha capa de contexto se la a denotar como *capa 2* y su contenido será RE_{t_2} y PN_{t_2} . Esta capa contiene la información necesaria para poder continuar con la ejecución de la rutina de servicio del disco duro una vez sea atendida la interrupción del reloj.
- 4) En el instante de tiempo t_3 finaliza la ejecución de la rutina de servicio del reloj y se continúa con la ejecución de la rutina de servicio del disco duro. Para poder continuar atendiendo la interrupción del disco duro desde el mismo punto donde lo dejó el núcleo extrae la *capa 2* de la PCC e inicializa RE y PN con los valores RE_{t_2} y PN_{t_2} , respectivamente.
- 5) En el instante de tiempo t_4 finaliza la ejecución de la rutina de servicio del disco duro y se continúa con la ejecución de la llamada al sistema. Para poder continuar ejecutando la llamada al sistema desde el mismo punto donde lo dejó el núcleo extrae la *capa 1* de la PCC e inicializa RE y PN con los valores RE_{t_1} y PN_{t_1} , respectivamente.
- 6) En el instante de tiempo t_5 finaliza la ejecución de la llamada al sistema, por lo que se cambia a modo usuario y se continua con la ejecución del proceso A. Para poder continuar ejecutando el código del proceso en modo usuario extrae la *capa 0* de la PCC e inicializa RE con el valor RE_{t_0} recuérdese que en modo usuario la pila del núcleo del proceso A está vacía.

En la Figura 4.5 se representa un diagrama con la configuración de la PCC del proceso A durante los distintos sucesos considerados.

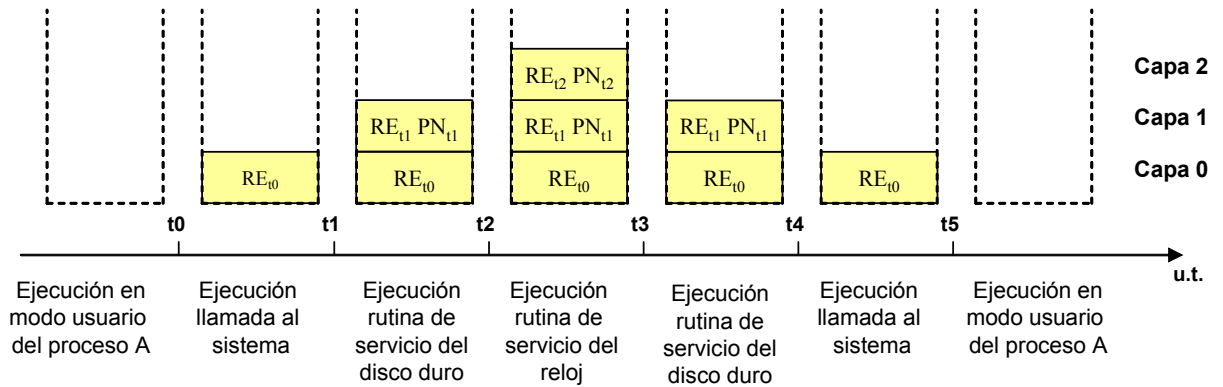


Figura 4.5: Configuración de la pila de capas de contexto del proceso A durante los distintos sucesos considerados

♦

4.5.3 Salvar y restaurar el contexto de un proceso

Se entiende por *salvar el contexto de un proceso* a la acción del núcleo de añadir una capa de contexto en la pila de capas de contexto asociada a dicho proceso. Por lo tanto, cuando se produce un cierto suceso, como una interrupción, una llamada al sistema o un cambio de contexto, no se salva todo el contexto del proceso propiamente dicho, sino solamente la parte dinámica del mismo, en concreto, el contexto de registros y la pila del núcleo.

Asimismo se entiende por *restaurar el contexto de un proceso* a la acción del núcleo de extraer la capa superior de la pila de capas de contexto asociada a dicho proceso e inicializar el contexto de registros y la pila del núcleo con los valores que se habían salvado en dicha capa. Por lo tanto, se deberá *restaurar el contexto de un proceso* cuando se termina de atender una interrupción, cuando se vuelve a modo usuario tras finalizar una llamada al sistema o cuando se realiza un cambio de contexto.

Las operaciones asociadas a salvar o restaurar el contexto de un proceso se suelen implementar por hardware o microcódigo, ya que se obtiene una mayor velocidad en su realización. En consecuencia, la implementación de estas acciones es fuertemente dependiente de la máquina.

Por otra parte, el núcleo también puede añadir una capa de contexto en el área U del proceso actualmente en ejecución. En ciertas circunstancias, el núcleo debe realizar una

vuelta abortiva, es decir, abortar su secuencia de ejecución actual, restaurar una capa de contexto salvada con anterioridad en el área U del proceso, y reiniciar su ejecución dentro del contexto restaurado. El núcleo usa el algoritmo `setjmp()` para salvar una capa de contexto en el área U del proceso. Asimismo, usa el algoritmo `longjmp()` para extraer dicha capa salvada en el área U e inicializar el contexto de registros y la pila del núcleo con los valores que se habían salvado en dicha capa. Los algoritmos del núcleo `setjmp()` y `longjmp()` no deben ser confundidos con las funciones de librería que tienen el mismo nombre. No obstante, su utilidad es muy parecida.

4.5.4 Cambio de contexto

Se define el *cambio de contexto* como el conjunto de tareas que debe realizar el núcleo para aplazar o finalizar la ejecución del proceso (A) actualmente en ejecución, y comenzar o continuar con la ejecución de otro proceso (B).

Cuando se ejecuta un proceso se dice que el sistema se ejecuta en el contexto de dicho proceso. Por lo que cuando el núcleo realiza un cambio de contexto, pasará de ejecutarse en el contexto del proceso A a ejecutarse en el contexto del proceso B.

Entre las principales circunstancias que motivan la realización de un *cambio de contexto* se encuentran:

- *La entrada de un proceso en el estado dormido.* Como se verá en la sección 4.8 uno de los posibles estados en que se puede encontrar un proceso es en el estado *dormido*. Un proceso entra en dicho estado cuando por ejemplo tiene que esperar por una operación de E/S con el disco duro. La realización de un cambio de contexto en esta circunstancia está plenamente justificada ya que puede transcurrir una cierta cantidad de tiempo hasta que el proceso despierte, con lo que mientras tanto se pueden ejecutar otros procesos.
- *La finalización de la ejecución de una llamada al sistema `exit`.* Esta llamada al sistema provoca la terminación del proceso en cuyo contexto se está ejecutando el núcleo. Por lo tanto, puesto que el proceso actual ha sido finalizado el núcleo debe hacer un cambio de contexto para continuar o iniciar la ejecución de otro proceso.
- *La vuelta al modo usuario tras ejecutarse una llamada al sistema y la existencia de un proceso esperando para ser ejecutado con mayor prioridad de planificación que el actual.* En este caso se produce un cambio de contexto

porque si hay otro proceso con mayor prioridad de planificación, sería injusto mantenerle esperando.

- *La vuelta al modo usuario tras atenderse una interrupción y la existencia de un proceso esperando para ser ejecutado con mayor prioridad de planificación que el actual.* La justificación del cambio de contexto en esta circunstancia es análoga al caso anterior.
- *La finalización del tiempo de uso del procesador de un proceso ejecutándose en modo usuario.* Como se estudiará en el Tema 6, de acuerdo con el tipo de algoritmo de planificación utilizado en UNIX, en esta circunstancia se planifica otro proceso para ser ejecutado en modo usuario y por tanto es necesario realizar un cambio de contexto.

El algoritmo del núcleo que implementa un *cambio de contexto* es de los más difíciles de entender del sistema operativo. Desde un punto de vista introductorio, bastará con conocer que las principales tareas que realiza el algoritmo del núcleo para implementar un *cambio de contexto* son:

- 1) Decidir si hay que hacer un cambio de contexto, de acuerdo con las circunstancias que lo motivan expuestas anteriormente, y si puede realizarse en el instante actual.
- 2) Salvar el contexto del proceso actual (A).
- 3) Usar el algoritmo de planificación de procesos para elegir el próximo proceso (B) cuya ejecución se va a iniciar o se va a continuar.
- 4) Restaurar el contexto del proceso (B) que ha sido elegido para ser ejecutado.

Antes de hacer un cambio de contexto, el núcleo debe asegurarse de que el estado de sus estructuras de datos sea consistente, es decir, que se hayan hecho todas las actualizaciones correctamente, que las colas estén enlazadas apropiadamente, que se han colocado los bloqueos adecuados para evitar la intrusión de otros procesos, que no se ha quedado bloqueada innecesariamente ninguna estructura de datos, etc.

4.6 TRATAMIENTO DE LAS INTERRUPCIONES

Las interrupciones (hardware o software) son atendidas en modo núcleo dentro del contexto del proceso que se encuentra actualmente en ejecución, aunque dicha interrupción no tenga nada que ver con la ejecución de dicho proceso.

En la siguiente descripción del tratamiento de las interrupciones se va a utilizar, por simplificar, las siguientes abreviaturas:

- *npi*, es el *nivel de prioridad de interrupción* actual almacenado en el registro de estado del procesador.
- *np_{ii}*, es el nivel de prioridad de interrupción asociado a un determinado tipo de interrupción.

Cuando se produce una interrupción, ésta es tratada por el núcleo si *np_{ii}* > *npi*, en dicho caso el núcleo invoca al algoritmo `inthand()` para el tratamiento de las interrupciones. Este algoritmo realiza principalmente las siguientes acciones:

- 1) *Salvar el contexto del proceso actual.*
- 2) *Elevar el nivel de prioridad de interrupción.* Es decir, se hace *npi*=*np_{ii}*. Por tanto, las interrupciones que lleguen con un nivel de prioridad de interrupción igual o menor que *npi* quedan bloqueadas o enmascaradas temporalmente. De esta forma se logra preservar la integridad de las estructuras de datos del núcleo.
- 3) *Obtener el vector de interrupción.* Normalmente, las interrupciones aparte del *np_{ii}* pasan al núcleo alguna información que le permite identificar el tipo de interrupción de que se trata. En un sistema con interrupciones vectorizadas, cada dispositivo suministra al núcleo un número único denominado *número del vector de interrupción* que se utiliza como un índice en una tabla, denominada *tabla de vectores de interrupción*. Cada entrada de esta tabla es un *vector de interrupción*, que contiene, entre otras informaciones, un puntero al manejador o rutina de servicio de la interrupción apropiado.
- 4) *Invocar al manipulador o rutina de servicio de la interrupción.*
- 5) *Restaurar el contexto del proceso*, una vez que se ha concluido la rutina de servicio de la interrupción.

En consecuencia cuando finaliza `inthand()` el nivel del *npi* es restaurado al valor que tenía antes de atenderse la interrupción.

Las peticiones de interrupción que pudieran haber quedado bloqueadas o enmascaradas durante la ejecución de `inthand()` son almacenadas en un registro

especial de peticiones de interrupción. Estas interrupciones serán atendidas cuando el *npi* disminuya suficientemente.

Algunas máquinas disponen de una pila especial denominada *pila de interrupciones* que es utilizada por todos los manejadores de interrupción. En las máquinas que no disponen de una pila de interrupciones los manejadores utilizan la *pila del núcleo* asociada al proceso.

Por otra parte, el tratamiento de la interrupción provoca un pequeño impacto en el proceso en cuyo contexto es atendida, ya que el tiempo utilizado para servir la interrupción es cargado al cuanto del proceso. Asimismo, es importante resaltar que el contexto del proceso no está protegido de forma explícita de ser accedido por los manipuladores de interrupciones. Un manipulador incorrectamente escrito potencialmente puede corromper cualquier parte del espacio de direcciones del proceso.

4.7 INTERFAZ DE LAS LLAMADAS AL SISTEMA

Las *llamadas al sistema* son el mecanismo que los procesos de usuario utilizan para solicitar al núcleo el uso de los recursos del sistema. Un proceso invoca a una llamada al sistema como si se tratase de una función de librería cualquiera. El compilador de C utiliza una librería predefinida de funciones, denominada *librería C* que contiene los nombres de las llamadas al sistema y que es enlazada, por defecto, con todos los programas de usuario. Estas funciones de librería, entre otras instrucciones, ejecutan una instrucción que provoca una interrupción software especial denominada *trap del sistema operativo*.

El tratamiento del *trap* por parte del núcleo provoca el cambio de modo de ejecución a modo núcleo, salvar el contexto del proceso actual y la invocación del algoritmo del núcleo que trata las llamadas al sistema, típicamente denominado `syscall()`.

El algoritmo del núcleo `syscall()` para el tratamiento de las llamadas al sistema requiere de un único parámetro de entrada que le es pasado por la función de librería. Este parámetro es un identificador numérico que sirve al núcleo para identificar la llamada al sistema que debe ejecutar. Distintas funciones de librería pueden hacer referencia a la misma llamada al sistema, al pasar al núcleo el mismo identificador numérico. Por ejemplo, las funciones de librería `execl` y `execle` hacen referencia a la misma llamada al sistema `exec`. La diferencia entre estas funciones únicamente radica en sus parámetros de entrada.

La primera acción que realiza `syscall()` es encontrar la entrada asociada al identificador numérico en una tabla de llamadas al sistema. Allí podrá obtener la dirección de comienzo de la rutina del núcleo asociada a la llamada al sistema y el número de parámetros de entrada que necesita dicha rutina para poder ejecutarse. Después el núcleo copia en el área U los parámetros de entrada de la llamada al sistema situados en la pila de usuario. A continuación salva el contexto del proceso en el área U (usando el algoritmo `setjmp`) en previsión de una posible vuelta abortiva, e invoca a la rutina del núcleo asociada a la llamada al sistema.

Después de ejecutar la rutina de la llamada al sistema, `syscall()` comprueba si el indicador de errores durante la ejecución de una llamada al sistema del área U está activado. En caso afirmativo, guarda un identificador numérico del error producido en el contenido del registro 0 salvado en la capa superior de la pila de capas de contexto del proceso. Además activa el bit de acarreo en el contenido del registro de estado del procesador salvado en dicha capa.

Si no hubo ningún error en la ejecución de la llamada al sistema, `syscall()` guarda el resultado de la llamada al sistema en el contenido de los registros 0 y 1 salvado en la capa superior de la pila de capas de contexto del proceso. Además desactiva (si no lo estaba ya) el bit de acarreo en el contenido del registro de estado del procesador salvado en dicha capa. La Figura 4.6 muestra un diagrama que resume las principales acciones realizadas por el algoritmo `syscall()`.

Una vez finalizado `syscall()`, el núcleo concluye el tratamiento del *trap* restaurando el contexto del proceso y cambiando el modo de ejecución a modo usuario, donde se continuará con la ejecución del código de la función de librería asociada a la llamada al sistema.

La función de librería, comprobará si el bit de acarreo del registro de estado del procesador está activado, es decir, si se produjo algún error durante la ejecución de la llamada al sistema. En caso afirmativo invoca a una función de librería de notificación de errores en las llamadas al sistema que mueve el identificador numérico del error almacenado en el registro 0 a la variable externa `errno`. Además esta función guarda en el registro 0 el valor -1. Finalizada la rutina de tratamiento de errores, se vuelve a la función de librería asociada a la llamada al sistema que también concluye, su valor de retorno es -1.

Por el contrario, si el bit de acarreo del registro de estado del procesador está desactivado, es decir, no se produjo ningún error durante la llamada al sistema, concluye la función de librería asociada a la llamada al sistema. Su valor de retorno es el contenido de los registros 0 y 1.

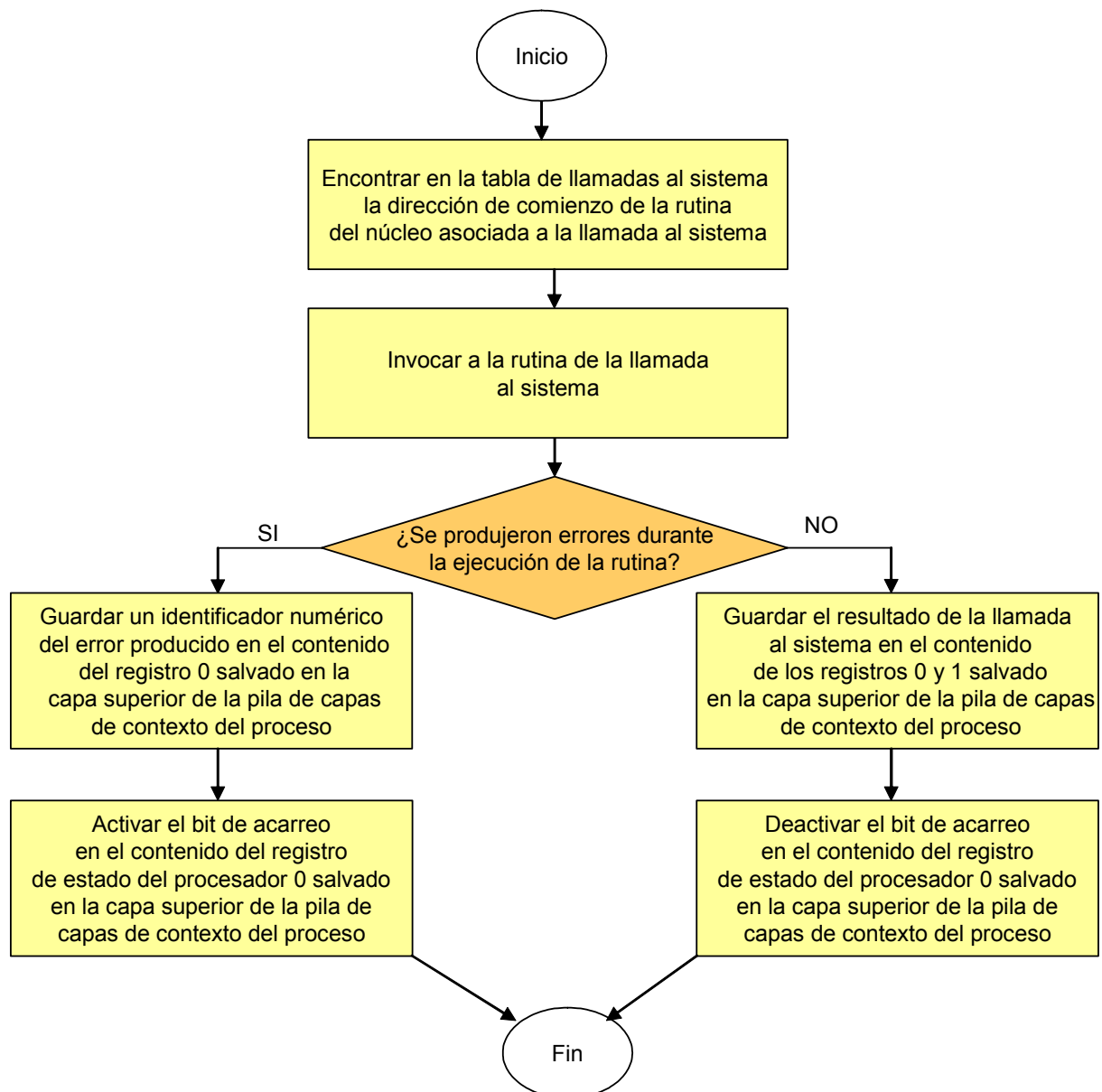


Figura 4.6: Principales acciones realizadas por el algoritmo `syscall()`

Una vez finalizada la ejecución de la función de librería asociada a la llamada al sistema se continuará con la ejecución del código del proceso. En la Figura 4.7 se muestra, a modo de resumen, un posible diagrama con el interfaz de las llamadas al sistema que se acaba de describir.

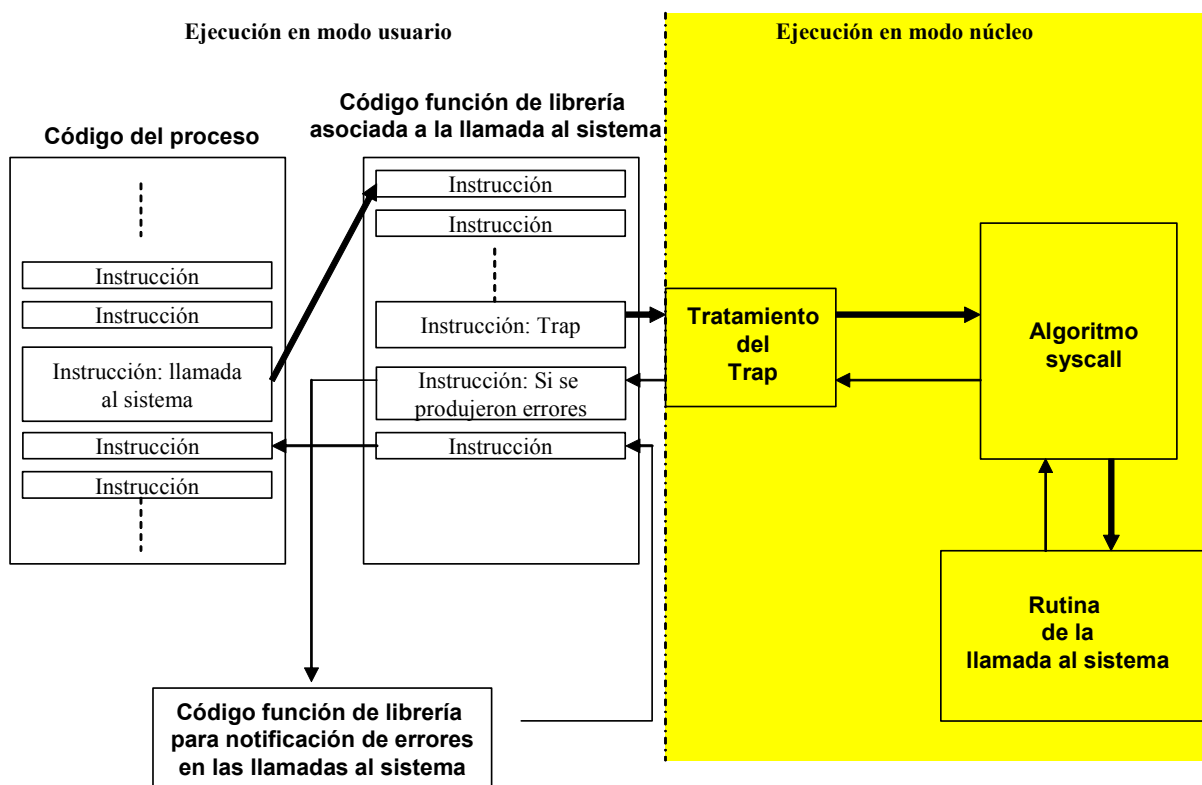


Figura 4.7: Interfaz de las llamadas al sistema

En dicha figura se observa como la invocación de una llamada al sistema en el código del proceso que se está ejecutando produce el salto a la función de librería asociada a dicha llamada al sistema. Cuando en dicha función se ejecuta un trap, éste es tratado por el núcleo, lo que entre otras acciones produce la invocación del algoritmo del núcleo `syscall()` para el tratamiento de las llamadas al sistema. Este algoritmo se encargará, entre otras cosas, de invocar a la rutina del núcleo apropiada para cada llamada al sistema. Una vez finalizada la rutina, se vuelve a `syscall()`. Cuando el algoritmo finaliza se procede a concluir el tratamiento del trap. Así se vuelve al modo usuario, y se sigue con la ejecución de la función de librería asociada a la llamada al sistema, que comprueba si se han producido errores. En caso afirmativo salta a una función de librería de notificación de errores en las llamadas al sistema. Una vez finalizada la ejecución de la función notificación de errores o si no hizo falta invocarla, se vuelve a la función de librería asociada a la llamada al sistema, que concluirá saltando de vuelta a la siguiente instrucción del código del proceso que seguía a la invocación de la llamada al sistema.

♦ Ejemplo 4.7:

Considérese el Programa 4.3 escrito en lenguaje C, que invoca a la llamada al sistema `creat` para crear un archivo llamado `prueba` con permisos de lectura y escritura para todos los usuarios.

```
char name[]="prueba";
main()
{
    int fd;
    fd=creat(name,0666);
}
```

Programa 4.3

Supóngase que el Programa 4.3 es compilado en una computadora que tiene un procesador Motorola 68000. En el Cuadro 4.2 se muestra una porción editada del código ensamblador generado por el compilador de C. Se observa que existen tres zonas diferenciadas: el código de la función `main`, el código de la función de librería asociada a la llamada al sistema `creat` y el código de la función de librería para la notificación de errores en las llamadas al sistema.

Dir.	Instrucción	
	...	
	# Código para main	
	...	
58:	mov	&0x1b6, (%sp) # mover 0666 dentro de la pila
5e:	mov	&0x204, -(%sp) # mover puntero de la pila
		# y mover la variable "name" dentro de la pila
64:	jsr	0x7a # llamada a la librería C para creat
	...	
	# Código de la función de librería asociada a creat	
7a:	movq	&0x8, %d0 # mover el valor del dato (8) dentro del registro 0
7c:	trap	&0x0 # trap
7e:	bcc	&0x6 <86> # bifurcación a la dirección 86 si el bit de acarreo es 0
80:	jmp	0x13c # saltar a la dirección 13c
86:	rts	# volver de la subrutina
	...	
	# Código de la función de notificación de errores en las llamadas al sistema	
13c:	mov	%d0, &0x20e # mover el contenido del registro 0 a la posición 20e (errno)
142:	movq	&-0x1, %d0 # mover constante -1 dentro del registro 0
146:	rts	# volver de la subrutina

Cuadro 4.2: Porción editada del código ensamblador generado por el compilador de C al compilar el Programa 4.3 en una computadora con procesador Motorola 68000

En el código mostrado para la función `main` se observa que (direcciones 58 y 5e) se copian los parámetros de la llamada al sistema `creat`, es decir, la máscara de modo del fichero 0666 y la variable `name`, dentro de un marco de la pila de usuario,. A continuación (dirección 64) se llama a la función de librería asociada a la llamada al sistema `creat`, cuya dirección es 7a. Aunque no

aparece en el Cuadro 4.3 se va a suponer que la dirección de retorno desde la función de librería es 6a. Esta dirección de retorno también se copia dentro del mismo marco de la pila de usuario.

En el código mostrado para la función de librería asociada a la llamada al sistema `creat` se observa que (dirección 7a) se mueve el identificador numérico (8) de la llamada al sistema, dentro del registro 0. A continuación (dirección 7c) se invoca el `trap`. El tratamiento del `trap` provoca en entre otras acciones provoca la invocación del algoritmo del núcleo `syscall()` para el tratamiento las llamadas al sistema. El algoritmo `syscall()` obtendrá del registro 0 el identificador numérico 8 que le permitirá determinar que la rutina que debe ejecutar es la de la llamada al sistema `creat`.

Cuando se vuelva a modo usuario después de ejecutar la rutina asociada a `creat`, concluir el algoritmo `syscall()` y finalizar el tratamiento del `trap`, se continúa con la ejecución del código de la función de librería asociada a la llamada al sistema. En concreto, se retorna a la instrucción cuya dirección es 7e que comprueba si el bit de acarreo del registro de estado del procesador está activado, es decir, si se produjo algún error durante la ejecución de la llamada al sistema. Si no hubo errores, salta de la dirección 7e a la dirección 86, última instrucción de esta función que retorna la ejecución al código de la función `main`, en concreto a la dirección 6a. Su valor de retorno es el contenido de los registros 0 y 1.

Por el contrario si está activado el bit de acarreo, el proceso salta a la dirección 13c, que es la dirección de comienzo del código de la función de librería para la notificación de errores en las llamadas al sistema. En el código asociado a dicha función se observa (dirección 13c) que se mueve el código de error localizado en el registro 0 a la dirección 20e asociada a la variable global `errno`. A continuación (dirección 142) coloca el valor -1 en el registro 0. Finalmente (dirección 146) la función finaliza y se retorna la ejecución al código de la función de librería asociada a `creat`, en concreto a la dirección 86. Esta es la última instrucción de esta función que retorna la ejecución al código de la función `main`, en concreto a la dirección 6a. Su valor de retorno es -1.

◆

De acuerdo con el interfaz de las llamadas al sistema descrito, si una llamada al sistema falla, la función de librería asociada devolverá el valor -1. Para averiguar cuál es el error que se ha producido, se ha de consultar el identificador numérico del error almacenado en la variable global `errno`. En el fichero de cabecera `<errno.h>` hay una descripción de todos los valores que puede tomar `errno`. Algunos valores de `errno` tienen asociado una constante, por ejemplo, la constante `EINTR` indica que la llamada al sistema fue interrumpida por la recepción de una señal.

Por otra parte, en la variable global `sys_errlist` definida en el fichero de cabecera `<stdio.h>` se almacena una tabla con las cadenas descriptoras de todos los códigos de error del sistema. El número de cadenas descriptoras que contiene es `sys_nerr`.

Existen dos formas de obtener el mensaje de error asociado a la variable `errno`: la primera es usar `errno` como índice para acceder a la cadena correspondiente de `sys_errlist`. La segunda es usar la función `perror`, cuya sintaxis es:

```
perror(cadena);
```

donde `cadena` es un array de caracteres. Su ejecución muestra el contenido de `cadena` seguido de ":" y del mensaje asociado al identificador de error contenido en la variable `errno`.

♦ Ejemplo 4.8:

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

main()
{
    char buffer[100];
    int iden=20;
    if(read(iden,buffer,100)==-1);
    {
        printf("\n%d: %s\n",errno,sys_errlist[errno]);
    }
}
```

Programa 4.4

El Programa 4.4 es un ejemplo de como obtener el mensaje asociado a la variable `errno`. En este programa se invoca a la llamada al sistema `read` para leer un archivo con identificador de archivo igual a 20. Esta llamada al sistema va a fallar ya que no se ha creado previamente un fichero al que se le haya asociado dicho descriptor. En consecuencia la función de librería asociada a la llamada al sistema devuelve el valor -1 y coloca en la variable `errno` el identificador numérico del error cometido. Así, este programa genera la siguiente salida en pantalla:

```
9: Bad file descriptor
```

Donde, 9 es el identificador numérico del error cometido, que estaba almacenado en `errno` y `Bad file descriptor` (descriptor de fichero incorrecto) es la cadena de texto asociada a dicho error, que estaba almacenada en `sys_errlist[errno]`.

◆

◆ **Ejemplo 4.9:**

```
main()
{
    char buffer[100];
    int iden=20;
    if(read(iden,buffer,100)==-1);
    {
        perror("Mensaje");
    }
}
```

Programa 4.5

El Programa 4.5 prácticamente equivalente al Programa 4.4 es un ejemplo del uso de la función `perror`. Este programa al ser ejecutado presenta la siguiente salida en pantalla:

```
Mensaje: Bad file descriptor
```

◆

Finalmente comentar, que cada llamada al sistema dispone de una página en el manual de ayuda de UNIX. Dicha página contiene la declaración de la función de biblioteca asociada a la llamada al sistema correspondiente, una explicación del uso de la llamada al sistema y una descripción de los valores de retorno y de los posibles mensajes de error.

4.8 ESTADOS DE UN PROCESO

4.8.1 Consideraciones generales

El tiempo de vida de un proceso en un sistema UNIX puede ser conceptualmente dividido en un conjunto de estados que describen el comportamiento del proceso. Los nueve estados en que se puede encontrar un proceso en un sistema UNIX (SVR2 o SVR3) son:

- *Ejecutándose en modo usuario.*
- *Ejecutándose en modo núcleo o supervisor.*
- *Preparado en memoria principal para ser ejecutado.* El proceso no está ejecutándose, pero está cargado en memoria principal listo para ser ejecutado tan pronto lo planifique el núcleo.
- *Dormido o bloqueado en memoria principal.* El proceso se encuentra esperando en memoria principal a que se produzca un determinado evento, como por ejemplo, la finalización de una operación de E/S.
- *Preparado en memoria secundaria para ser ejecutado.* El proceso está listo para ser ejecutado pero se encuentra en memoria secundaria.
- *Dormido o bloqueado en memoria secundaria.* El proceso está esperando en memoria secundaria a que se produzca un determinado evento.
- *Expropiado.* Cuando un proceso (A) ejecutándose en modo usuario ha finalizado su cuanto, llega una interrupción del reloj del sistema para avisar de este hecho. El tratamiento de esta interrupción en modo núcleo, provoca que el proceso A sea expropiado de la CPU y que otro proceso B pase a ser planificado para ser ejecutado. En esencia, el estado *expropiado* es el mismo que el estado *preparado en memoria principal para ser ejecutado*, pero se describen separadamente para enfatizar que un proceso *expropiado* tiene garantizado que su próximo estado será *ejecución en modo usuario* cuando vuelva a ser planificado para ser ejecutado.
- *Creado.* El proceso se ha creado recientemente y está en un estado de transición. El proceso existe, pero no se encuentra *preparado para ser ejecutado* ni tampoco está *dormido*. Este estado es el inicial para todos los procesos excepto para el proceso con *pid=0*.
- *Zombi.* Este es el estado final de un proceso al que se llega mediante la ejecución explícita o implícita de la llamada al sistema `exit`.

En la Figura 4.8 se representa el *diagrama de transición de estados* de los procesos en un sistema UNIX (SVR2 o SVR3). En dicho diagrama los *nodos* representan a los posibles *estados* de un proceso. Asimismo las *líneas de conexión* representan las posibles *transiciones* entre los estados. Estas líneas de conexión se encuentran rotuladas

con el evento que provoca que un proceso pase de un estado a otro. Una transición entre dos estados es legal si existe una línea de conexión en el sentido adecuado que los una.

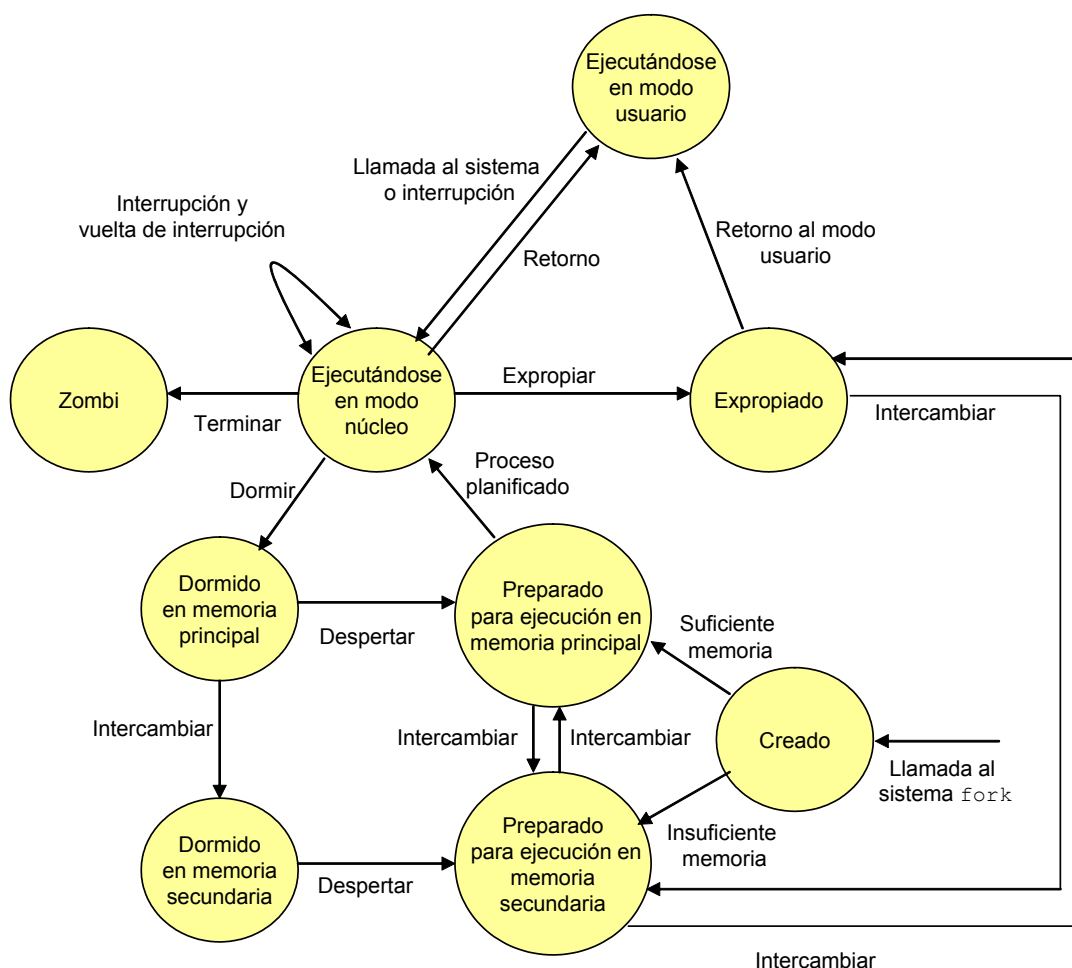


Figura 4.8: Diagrama de transiciones de estado en un sistema UNIX (SVR2 o SVR3)

Se van a analizar a continuación las posibles transiciones de estado, partiendo del nacimiento de un proceso. Cuando un nuevo proceso (A) se crea, mediante una llamada al sistema `fork` realiza por otro proceso (B), el primer estado en el que entra A es el estado *creado*. Desde aquí puede pasar, dependiendo de si existe suficiente espacio en memoria principal, a dos estados distintos: *preparado para ejecución en memoria principal* o *preparado para ejecución en memoria secundaria*.

Si el proceso se encuentra en el estado *preparado para ejecución en memoria principal* entonces el planificador de procesos puede escogerlo para ser ejecutado, por lo que pasará al estado *ejecución en modo supervisor*. Cuando el proceso finalice la ejecución de su parte de la llamada al sistema `fork` entonces pasará al estado *ejecución*

en *modo usuario*, donde comenzará a ejecutarse las instrucciones de la región de código del proceso.

Cuando el proceso agota su cuanto, el reloj del sistema mandará una interrupción al procesador. El tratamiento de la interrupciones se realiza en modo núcleo, en conclusión, el proceso debe pasar de nuevo al estado *ejecutándose en modo núcleo*. Cuando el manipulador de la interrupción de reloj finaliza, el planificador expropiará de la CPU al proceso A y planificará otro proceso C para ser ejecutado. De estado forma el proceso A pasa al estado *expropiado*. Cuando el planificador vuelva a seleccionar al proceso A para ser ejecutado este volverá al estado *ejecutándose en modo usuario*.

Si el proceso A invoca durante su ejecución en modo usuario a una llamada al sistema, entonces pasa al estado *ejecución en modo núcleo*. Supóngase que la llamada al sistema necesita realizar una operación de E/S con el disco, entonces el núcleo debe esperar a que se complete la operación, en consecuencia el proceso (A) pasa al estado *dormido en memoria principal*. Cuando se completa la operación de E/S, el hardware interrumpe a la CPU y el manipulador de la interrupción despertará al proceso, lo que provocará que pase al estado *preparado para ejecución en memoria principal*.

Supóngase que en el sistema se están ejecutando muchos procesos y que no existe suficiente espacio en memoria. En esta situación el *intercambiador* elige para ser intercambiados a memoria secundaria a algunos procesos (entre ellos el proceso A) que se encuentran en el estado *preparado para ejecución en memoria principal* o en el estado *expropiado*. Estos procesos pasarán al estado *preparado para ejecución en memoria secundaria*.

En un momento dado, el *intercambiador* elige al proceso más apropiado para intercambiarlo de vuelta a la memoria principal, supóngase que se trata del proceso A. Éste pasa al estado *listo para ejecución en memoria*. A continuación, el planificador en algún instante elegirá el proceso para ejecutarse y entonces pasará al estado *ejecución en modo supervisor* donde continuará con la ejecución de la llamada al sistema. Cuando finalice la llamada al sistema pasará de nuevo al estado *ejecución en modo usuario*.

Cuando el proceso se complete, invocará explícitamente o implícitamente a la llamada al sistema `exit`, en consecuencia pasará al estado *ejecución en modo supervisor*. Cuando se complete esta llamada al sistema pasará finalmente al estado *zombi*.

Un proceso tiene control sobre algunas transiciones de estado. En primer lugar, un proceso puede crear otro proceso. Sin embargo, es el núcleo quién decide en que momento se realizan la transición desde el estado *creado* al estado *preparado para ejecución en memoria principal* o al estado *preparado para ejecución en memoria secundaria*.

En segundo lugar, un proceso puede invocar a una llamada al sistema lo que provocará que pase del estado *ejecución en modo usuario* al estado *ejecución en modo núcleo*. Sin embargo, el proceso no tiene control de cuando volverá de este estado, incluso algunos eventos pueden producir que nunca retorne y pase al estado *zombi*.

En tercer lugar, un proceso puede finalizar realizando una invocación explícita de la llamada al sistema `exit`, pero por otra parte eventos externos también pueden hacer que se produzca la terminación del proceso.

El resto de las transiciones de estado siguen un modelo rígido codificado en el núcleo. Por lo tanto, el cambio de estado de un proceso ante la aparición de ciertos eventos se realiza de acuerdo a unas reglas predefinidas.

4.8.2 Estados adicionales

UNIX BSD4 define algunos estados adicionales que no son soportados en SVR2 ni SVR3, pero si en SVR4. Como por ejemplo, el estado *parado o suspendido* (en memoria principal o secundaria) y el estado *dormido y parado* (en memoria principal o secundaria). En el estado parado, la ejecución del proceso es detenida, pero posteriormente puede retomarse. En la sección 5.3.1.2 se describirá como puede un proceso entrar en estos estados.

4.8.3 El estado *dormido*

El estado *dormido en memoria principal* es uno de los posibles estados de un proceso que por su importancia requiere de una atención especial. Un proceso siempre pasa al estado *dormido en memoria principal* desde el estado *ejecución en modo supervisor*. Principalmente, un proceso pasa al estado *dormido* cuando se produce alguna de las siguientes circunstancias:

- Durante la ejecución de una llamada al sistema el núcleo requiere usar un recurso que se encuentra ocupado, o debe esperar a que termine una transferencia de E/S.

- Se produce un fallo de página como resultado de acceder a una dirección virtual que no está cargada en memoria principal.

Un proceso permanecerá en el estado *dormido* hasta que tenga lugar el evento por el que se encuentra esperando. Cuando dicho evento ocurra, el proceso será despertado y pasará al estado *preparado para ejecución en memoria (principal o secundaria)*.

Cada evento que debe ocurrir para que un proceso se despierte está asociado con un *canal o dirección de dormir*. Este canal es una dirección virtual del núcleo asociada a un determinado *recurso*. Distintos eventos pueden estar asociados a un mismo canal.

Por otra parte, cuando un proceso pasa al estado *dormido* en espera de un determinado evento el núcleo lo añade a una *lista de procesos dormidos*. Además almacena la *dirección de dormir* en el campo correspondiente de la entrada asociada al proceso en la tabla de procesos.

♦ Ejemplo 4.10:

En la Figura 4.9 se observa como la lista de procesos dormidos contiene 8 procesos. Los procesos están en el estado dormido esperando por que se produzcan los siguientes eventos:

- Finalización de una operación de E/S (proceso C).
- Desbloqueo del buffer (procesos A, E, F y H).
- Desbloqueo de un nodo-i (procesos B y G).
- Entrada en el terminal (proceso D).

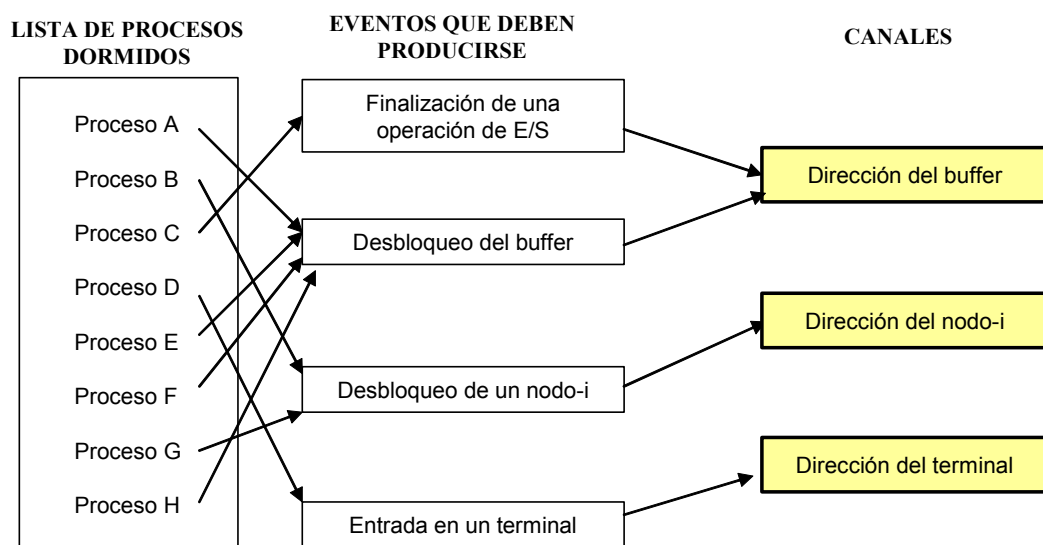


Figura 4.9: Lista de procesos dormidos, eventos que deben producirse y canales asociados

Se observa como existen tres canales o direcciones de dormir, las direcciones del buffer, del nodo-i y del terminal, respectivamente. Los eventos *finalización de una operación de E/S* y *desbloqueo del buffer* tienen asociados la dirección del buffer. El evento *desbloqueo de un nodo-i* tiene asociada la dirección del nodo-i. Finalmente, el evento *entrada en el terminal* tiene asociada la dirección del terminal.



Esta implementación del estado dormido presenta dos anomalías. En primer lugar, cuando un evento tiene lugar, el núcleo despierta a todos los procesos de la lista de procesos dormidos que se encuentran esperando por la ocurrencia de dicho evento, y los pasa al estado *preparado para ejecución en memoria (principal o secundaria)*. Puesto que sólo uno de ellos puede ser planificado para ser ejecutado y usar el recurso por el que espera, el resto de los procesos tendrán que volver al estado *dormido* después de una breve visita al estado *ejecución en modo supervisor*, lo que genera cambios de contextos y procesamientos innecesarios. Obviamente, la implementación sería más eficiente si solamente se despertará a aquel proceso dormido que tiene una mayor probabilidad de ser planificado para ser ejecutado, es decir, su *prioridad de planificación* es mayor.

La segunda anomalía, es que distintos eventos pueden estar asociados a un mismo canal o dirección de dormir. La implementación sería más eficiente si cada evento tuviese asociada su propio canal. Curiosamente, en la práctica el rendimiento del sistema no se ve muy perturbado por la existencia de esta anomalía puesto que es raro que se asocien muchos eventos a un mismo canal. Además, un proceso ejecutándose normalmente libera los recursos bloqueados antes de que otro proceso sea planificado para ejecución.

◆ **Ejemplo 4.11:**

En el esquema de la Figura 4.9 se tiene un ejemplo de la segunda anomalía comentada. Puesto que tanto el evento *finalización de una operación de E/S* como el evento *desbloqueo del buffer* tienen asociados la misma dirección de dormir (la dirección del buffer) cuando la operación de E/S con el buffer se completa, el núcleo despierta tanto al proceso C como a los procesos A, E, F y H. Puesto que el proceso C esperando por la terminación de una operación de E/S mantiene el buffer bloqueado, los procesos A, E, F y H que esperan por el desbloqueo del buffer para poder utilizarlo volverán al estado dormido, si el buffer sigue bloqueado, cuando vayan a ejecutarse. En consecuencia A, E, F y H han sido despertados inútilmente. Obviamente el sistema sería más eficiente si los procesos fuesen despertados cuando se estuviese seguro de que el buffer no está bloqueado.

Un ejemplo de la primera anomalía se da cuando el núcleo despierta a los procesos A, E, F y H al estar el buffer disponible (se supone que el proceso C ya completó su operación de E/S y ha desbloqueado el buffer). Los cuatro procesos compiten por el mismo recurso, solamente uno de ellos será planificado para ser ejecutado el resto volverá al estado *dormido*. Obviamente el sistema sería más eficiente si solo se despertase al proceso con una mayor prioridad de planificación.



Antes de comentar otra característica importante del estado dormido conviene recordar lo que se entiende por *señal*. Una *señal* es un mecanismo de comunicación que utiliza el núcleo para informar a un proceso de la ocurrencia de algún evento asíncrono. La posibilidad de poder interrumpir a un proceso en el estado dormido, es decir, de poder despertarlo cuando llega una señal para él, permite distinguir entre dos tipos de estado dormido:

- *Estado dormido no interrumpible por señales*. En este estado el proceso no puede ser interrumpido (no puede ser despertado) cuando llegue una señal para él. Si bien conviene matizar que existen algunas señales que no pueden ser ignoradas. Un proceso entra en este estado si se encuentra esperando por un evento que no tardará mucho en producirse, como por ejemplo que se complete una operación de E/S con el disco o que se libere por un recurso (nodo-i o buffer) bloqueado.
- *Estado dormido interrumpible por señales*. En este estado el proceso puede ser interrumpido (puede ser despertado) cuando llegue una señal para él. Un proceso entra en este estado si se encuentra esperando por un evento que puede tardar en producirse, como por ejemplo que el usuario pulse alguna tecla del teclado.

Como se estudiará en el Tema 6, el parámetro que usa el núcleo para determinar si un proceso entra en el estado dormido interrumpible o no interrumpible es el valor de la *prioridad de planificación de un proceso en modo núcleo*. El núcleo asigna una determinada *prioridad de planificación en modo núcleo* en función del evento por el proceso se encuentra esperando. A dicha prioridad se le suele denominar *prioridad para dormir*. Además define un *valor límite o umbral* de tal forma que si la *prioridad para dormir* de un proceso es mayor que dicho valor umbral el proceso entrará en el *estado dormido no interrumpible*. En caso contrario, el proceso entrará en el *estado dormido*

interrumpible. En la Tabla 4.1 se muestra a modo de ejemplo las prioridades para dormir utilizadas en la distribución BSD4.3.

Prioridad	Valor	Descripción
PSWP	0	Prioridad en la que duerme el proceso intercambiador
PSWP + 1	1	Prioridad en la que duerme el ladrón de páginas
PSWP + 1/2/4	1/2/4	Prioridades en las que duermen otras actividades de administración de memoria
PINOD	10	<i>Evento</i> : Desbloqueo de un nodo-i
PRIBIO	20	<i>Evento</i> : Finalización de una operación de E/S en disco
PRIBIO+1	21	<i>Evento</i> : Desbloqueo del buffer
PZERO	25	Prioridad umbral
TTIPRI	28	<i>Evento</i> : Entrada en un terminal
TTOPRI	29	<i>Evento</i> : Salida en un terminal
PWAIT	30	<i>Evento</i> : Terminación de un proceso hijo
PLOCK	35	<i>Evento</i> : Bloqueo de un recurso
PSLEP	40	<i>Evento</i> : Recepción de una señal

Tabla 4.1: Prioridades para dormir en el UNIX BSD4.3

TEMA 5

CONTROL DE PROCESOS EN UNIX

5.1 INTRODUCCION

Este tema está dedicado al estudio del uso y la implementación de las llamadas al sistema y los algoritmos del núcleo que permiten controlar a un proceso. En primer lugar se describe la llamada al sistema `fork` que permite crear un nuevo proceso (hijo) a partir de otro proceso (padre). En segundo lugar, se describen las *señales*, que permiten informar a los procesos de eventos asíncronos. Su estudio en profundidad es imprescindible para poder comprender los algoritmos `sleep()` y `wakeup()` que el núcleo utiliza dentro de la ejecución de las llamadas al sistema para pasar a un proceso al estado dormido (interrumpible o no interrumpible por señales) y para despertarlo, respectivamente. Ambos algoritmos también son explicados en este tema.

A continuación se describen la llamada al sistema `exit`, que permite terminar la ejecución de un proceso, y la llamada al sistema `wait`, que permite sincronizar la ejecución de un proceso con la terminación de alguno de sus procesos hijos. El núcleo sincroniza la ejecución de `exit` y `wait` mediante el uso de señales.

Además, se presenta la llamada al sistema `exec` que permite a un proceso invocar a un “nuevo” programa ejecutable. Finalmente se describen de forma general los mecanismos de UNIX para la sincronización de procesos.

En la explicación de las llamadas al sistema que se tratan en este tema se va a tomar como referencia principalmente el núcleo de una distribución clásica como SVR3.

5.2 CREACION DE PROCESOS

En un sistema UNIX la única forma que tiene un usuario de crear un nuevo proceso es invocando la llamada al sistema `fork`. El único proceso no creado mediante `fork` es el proceso 0 ($pid=0$), este proceso es creado internamente por el núcleo cuando arranca el sistema. Al proceso que invoca a `fork` se le denomina *proceso padre*, mientras que al nuevo proceso que se crea se le denomina *proceso hijo*. Todo proceso tiene un padre (excepto el proceso 0) y puede tener uno o más hijos.

La implementación de la rutina de la llamada al sistema `fork` no es trivial y varía ligeramente dependiendo de la política de gestión de memoria principal que implemente el sistema: demanda de páginas o intercambio. La descripción siguiente se centra en el funcionamiento de la llamada al sistema `fork` en un sistema con una política de gestión de memoria de intercambio. Además también se supone que el sistema tiene disponible suficiente memoria principal para almacenar al proceso hijo. En la sección 9.2.2 se describirá la implementación de `fork` en un sistema con una política de gestión de memoria por demanda de páginas.

Supóngase que el proceso A propiedad del usuario `usuario1` invoca a una llamada al sistema `fork`. El núcleo ejecutará el algoritmo `syscall()` para el tratamiento de las llamadas al sistema. Este algoritmo busca e invoca a la rutina asociada a esta llamada al sistema. La primera acción que realiza el núcleo al ejecutar la rutina de la llamada al sistema `fork` es comprobar la existencia de recursos suficientes en el sistema para poder crear al nuevo proceso. En un sistema con gestión de memoria mediante intercambio, debe existir suficiente espacio en memoria principal o en memoria secundaria para poder almacenar al nuevo proceso. En un sistema con gestión de memoria por demanda de páginas, el núcleo tiene que poder asignar memoria para alojar nuevas tablas de páginas.

Además el núcleo también comprueba que `usuario1` no tiene demasiados procesos ejecutándose. El sistema impone un límite, que es configurable, al número de procesos que un usuario puede ejecutar simultáneamente. Con este límite se pretende evitar que el sistema se cuelgue por haber sobrepasado la capacidad de la tabla de procesos. Solamente el superusuario puede ejecutar tantos procesos como quiera, limitado obviamente por el tamaño de la tabla de procesos.

Si estas comprobaciones no son positivas, es decir, no existen recursos suficientes o `usuario1` tiene demasiados procesos ejecutándose, la rutina del núcleo asociada a la

llamada al sistema `fork` finaliza. Se habrá producido, por tanto, un error durante el tratamiento de la llamada al sistema.

Si las comprobaciones son positivas, el núcleo asigna al nuevo proceso hijo una entrada en la tabla de procesos y un `pid`. Asimismo copia el contenido de la tabla de procesos asociada al proceso padre en la entrada de la tabla de procesos asociada al proceso hijo. De esta forma el hijo hereda, entre otras informaciones, los identificadores de usuario (`uid`, `euid`) y de grupo (`gid`, `egid`) del padre.

En el campo de información genealógica de la entrada de la tabla de procesos asociada al proceso hijo, copia el `pid` del proceso padre. Además, configura el campo del estado al estado *creado*, e inicializa varios parámetros necesarios para la planificación del proceso como temporizadores y valores de prioridad. Asimismo en el campo de información genealógica de la entrada de la tabla de procesos asociada al proceso padre copia el `pid` del proceso hijo.

A continuación, el núcleo incrementa el contador de referencias del nodo-i asociado al directorio de trabajo actual del proceso padre, ya que el proceso hijo también va a residir en dicho directorio. También, si el proceso padre o alguno de sus antepasados han ejecutado la llamada al sistema `chroot` para cambiar el directorio raíz, el proceso hijo hereda este directorio raíz cambiado, y en consecuencia el núcleo debe incrementar el contador de referencias de su nodo-i.

Asimismo, el núcleo busca en la tabla de archivos los ficheros abiertos por el proceso padre, e incrementa en una unidad el contador de referencias en sus entradas asociadas en dicha tabla. Por tanto el proceso hijo no solamente hereda los permisos de acceso a estos ficheros, sino que comparte el acceso a estos ficheros con el proceso padre puesto que ambos procesos manipulan las mismas entradas de la tabla de ficheros.

Hasta el momento, el único elemento del contexto del proceso hijo que se ha creado es su entrada en la tabla de procesos. Ahora, el núcleo crea los elementos de la parte estática del contexto del proceso hijo que le faltaban (contexto a nivel de usuario, área U, etc). Para ello hace una copia en memoria de la parte estática del contexto del proceso padre (usando el algoritmo `dupreg()`) y se la asocia al proceso hijo (algoritmo `attachreg()`). A continuación, el núcleo, modifica el campo del área U que contiene un puntero a la entrada de la tabla de procesos asociada al proceso hijo, para que apunte a la entrada del hijo.

Una vez finalizada la creación de la parte estática del contexto del proceso hijo, el núcleo procede a crear la parte dinámica. En primer lugar asigna memoria para la pila de capas de contexto del proceso hijo y añade en ella una copia de la capa 0 de la pila de capas de contexto del proceso padre. Después, si la pila del núcleo se implementa en un área de memoria independiente entonces asigna espacio para ella. En el caso de que se implemente dentro del área U, entonces el núcleo automáticamente crea la pila del núcleo al crear el área U. En ambos casos el contenido de la pila del núcleo asociado al padre y de la pila del núcleo asociado al hijo es idéntico.

A continuación, añade otra capa de contexto (capa 1) en la pila de capas de contexto del proceso hijo y configura el valor del contexto de registros salvado en ella para que el proceso hijo pueda comenzar a ejecutarse cuando sea planificado. Entre las configuraciones que realiza en el contexto de registros salvado en la capa 1 se encuentra el guardar en el registro 0 un determinado valor para posteriormente poder determinar si se está ejecutando el proceso padre o el proceso hijo, y el fijar en el contador de programa la dirección de la instrucción de la rutina de la llamada al sistema `fork` donde tiene que comenzar a ejecutarse el proceso hijo. Esta instrucción, que se va a denotar por *Instr_C*, típicamente es una instrucción condicional que chequea el valor almacenado en el registro 0 para determinar si se está ejecutando el padre o hijo.

Una vez concluida la creación del contexto del proceso hijo, el núcleo cambia el estado del proceso hijo al estado *preparado para ejecutarse en memoria principal*. La ejecución de la rutina del núcleo asociada a `fork` finaliza en el contexto del proceso padre devolviendo a `syscall()` el valor del *pid* del proceso hijo.

Asimismo, cuando el proceso hijo sea planificado para ser ejecutado, su contexto será restaurado, es decir, el núcleo extraerá la capa 1 de la pila de capas de contexto asociada al proceso hijo e inicializará el contexto de registros y la pila del núcleo con los valores que se habían salvado en dicha capa. Así el proceso hijo comienza a ejecutarse en la instrucción *Instr_C* de la rutina de `fork`. Finalmente, la ejecución de la rutina del núcleo asociada a `fork` finaliza en el contexto del hijo devolviendo a `syscall()` el valor 0.

A modo de resumen, la Figura 5.1 muestra un diagrama con las principales acciones que realiza el núcleo durante la ejecución de la rutina asociada a la llamada al sistema `fork`. Obsérvese como se han enmarcado con línea continua las acciones de la rutina de `fork` que se ejecutan en el contexto del proceso padre. Asimismo se han enmarcado con línea discontinua las acciones de la rutina de `fork` que se ejecutan en el contexto del

proceso hijo. En la intersección de ambos marcos se encuentran las acciones de `fork` que se realizan primero en el contexto del proceso padre y posteriormente también se realizan en el contexto del proceso hijo. Por lo tanto, la rutina de `fork` tiene la peculiaridad, con respecto a las rutinas asociadas a otras llamadas al sistema, de que se ejecuta en dos partes, la primera parte la ejecuta el proceso padre y la segunda parte la ejecuta el proceso hijo.

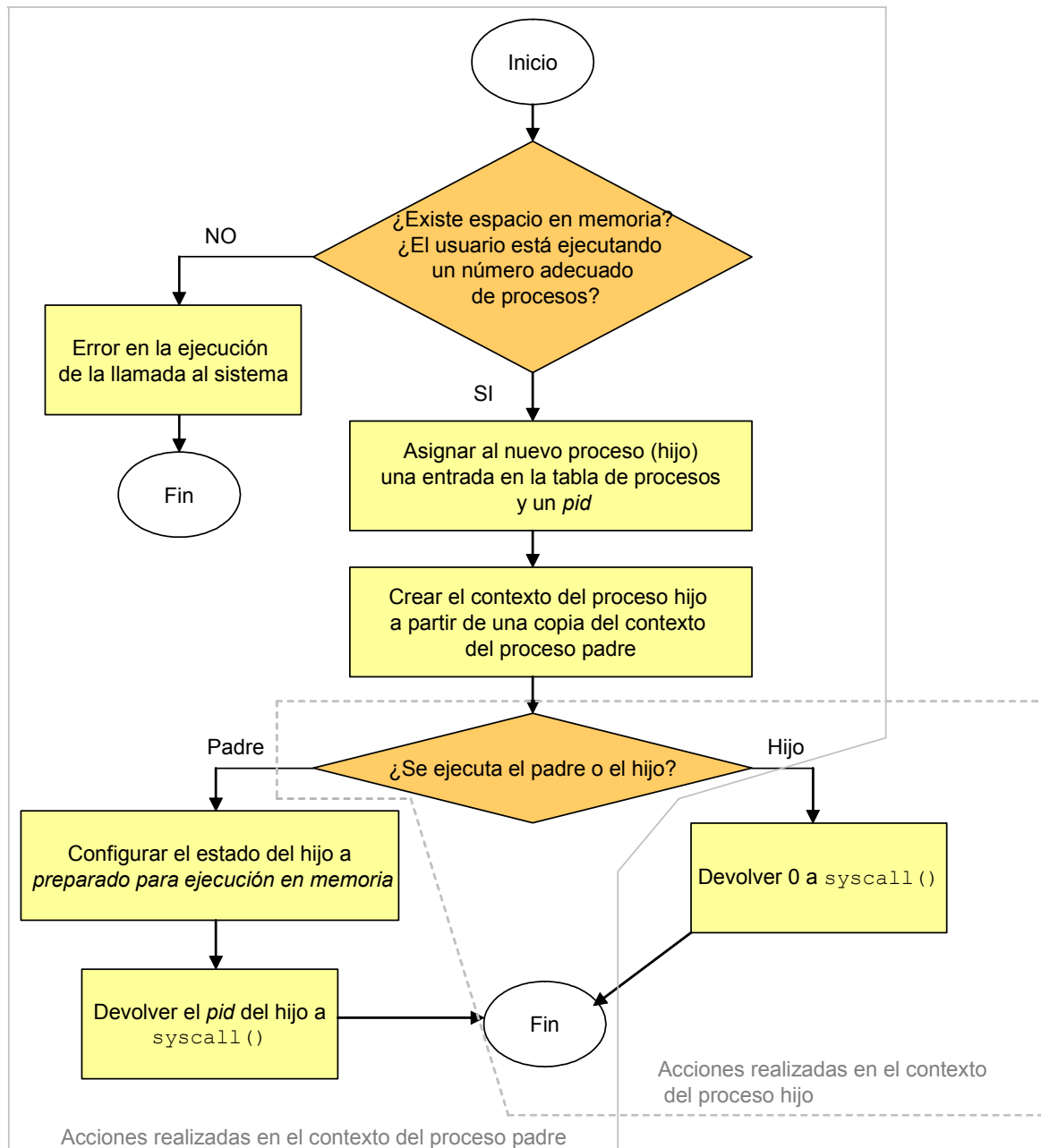


Figura 5.1: Principales acciones realizadas por el núcleo durante la ejecución de la rutina asociada a la llamada al sistema `fork`

Cuando el proceso hijo finaliza su parte de la llamada al sistema `fork` su contexto a nivel de usuario (código, datos, pila de usuario y memoria compartida) es una copia idéntica del contexto a nivel de usuario del proceso padre. Asimismo la tabla de descriptors de ficheros del proceso hijo es una copia de la tabla de descriptors de ficheros del proceso padre (recuérdese que esta tabla se implementaba en el área U asociada a un proceso). En conclusión el proceso hijo comparte el acceso a los ficheros abiertos por el proceso padre antes de la ejecución de `fork`.

Una vez descritas las acciones que realiza el núcleo durante la ejecución de la llamada al sistema `fork` es mucho más sencillo comprender su sintaxis:

```
par = fork();
```

Se observa que no requiere ningún parámetro de entrada y que posee un único parámetro de salida `par`, que puede tomar los siguientes valores:

- Para el proceso padre, `par` es igual al *pid* que le asigne el sistema al proceso hijo.
- Para el proceso hijo, `par` es igual a 0.
- En caso de error durante la ejecución de la llamada al sistema `par` es igual a -1.

♦ Ejemplo 5.1:

```
main()
{
[1]   int par;
[2]   int x=0;
[3]   if ((par=fork())== -1)
    {
[4]       printf("Error en la ejecución del fork");
[5]       exit(0);
    }
[6]   else if (par==0)
    {
        /* Este código nada más lo ejecuta el proceso hijo*/
[7]       x=x+2;
[8]       printf("\nProceso hijo, x= %d\n", x);
    }
}
```

```

[9]   else
      {
        /* Este código nada más lo ejecuta el proceso padre */
[10]       printf("\nProceso padre, x=%d\n", x);
      }
      /*Este código lo ejecuta el proceso padre y el proceso hijo */
[11]  printf("\nFinalizar\n");
}

```

Programa 5.1

Al ejecutar el programa ejecutable asociado al Programa 5.1, en primer lugar **[3]** se invoca a la llamada al sistema `fork`. Si durante la ejecución de esta llamada al sistema se produce un error `par` valdría -1, entonces se imprimiría **[4]** en pantalla el mensaje:

```
Error en la ejecución del fork
```

A continuación **[5]** se invocaría a la llamada al sistema `exit` para finalizar el programa.

Si no se producen errores, al terminar el proceso padre su parte de la llamada al sistema `fork` volverá a modo usuario dentro del `if` de la sentencia **[3]**. Como para el padre `par` es igual `pid` del proceso hijo, entonces se ejecutará la instrucción **[10]**, es decir, se imprime en pantalla el mensaje:

```
Proceso padre, x=0
```

Asimismo, cuando el proceso hijo sea planificado concluirá su parte de la llamada al sistema `fork` y volverá a modo usuario dentro del `if` de la sentencia **[3]**, como para hijo `par=0` ejecutará las instrucciones **[7]** y **[8]**, es decir, le suma 2 a la variable `x` e imprime en pantalla el mensaje:

```
Proceso hijo, x=2
```

Finalmente tanto el proceso padre como el hijo ejecutan la sentencia **[11]**, es decir, imprimen en pantalla el mensaje: `Finalizar`. Luego `Finalizar` se muestra dos veces en pantalla, una vez por la ejecución del padre y otra por la ejecución del hijo.

La ejecución descrita merece dos comentarios. En primer lugar, dependiendo del orden de planificación de los procesos, se podrían tener trazas de salida distintas. En otras palabras, el orden en que aparecerán los mensajes en pantalla dependerá de como el sistema planifique a los procesos padre e hijo. En segundo lugar, puesto que la región de datos del proceso hijo es una copia de la del padre, el hijo hereda los valores de las variables definidas en el programa hasta el momento de realizarse la ejecución del `fork`. Puesto que son dos procesos diferentes y por tanto con contextos distintos, los cambios que efectuó el hijo sobre dichas variables sólo se reflejarán

en su contexto y no en el del padre, salvo, claro está, que se haga sobre un segmento de memoria compartido. De acuerdo con este razonamiento al ejecutarse este programa, desde el punto del contexto del proceso hijo, inicialmente $x=0$ y al finalizar $x=2$. Mientras que desde el punto de vista del proceso padre, inicialmente $x=0$ y al finalizar $x=0$.

◆

◆ Ejemplo 5.2:

```
[1] int fichero1, fichero2;
[2] char character;

[3] main (int argc, char *argv[])
{
[4]     if (argc!=3)
[5]         exit(1);

[6]     fichero1=open(argv[1],0444);
[7]     fichero2=open(argv[2],0666);

[8]     if (fichero1==-1)
[9]         printf("\nError al abrir el fichero 1\n");
[10]    if (fichero2==-1)
[11]        printf("\nError al abrir el fichero 2\n");

[12]    fork();

[13]    leer_escribir();
[14]    exit(0);
}

[15] int leer_escribir()
{
[16]    for(;;)
    {
[17]        if (read(fichero1, &character, 1)!=1)
[18]            return(0);
[19]        write(fichero2, &character,1);
    }
}
```

Programa 5.2

Supóngase que el nombre del fichero ejecutable que se crea después de compilar el Programa 5.2 es `ejfork`. Un usuario invocaría a este programa desde un terminal (\$) escribiendo:

```
$ ejfork file_R file_W
```

donde `file_R` debe ser un archivo ya existente que contenga un determinado texto y `file_W` es el nombre de un fichero nuevo que va a ser creado.

El significado de las sentencias de este programa es el siguiente. En primer lugar se declaran unas variables globales: en **[1]** las variables enteras `fichero1` y `fichero2`, y en **[2]** la variable tipo carácter `carácter`. A continuación **[3]** se declara la función principal `main` del programa, que en este caso indica que el ejecutable `ejfork` podrá recibir argumentos desde la línea de comandos (\$) cuando sea invocado. El parámetro `argc` de tipo entero contendrá el número de argumentos que recibe el ejecutable desde la línea de comandos, recuerde que el nombre del ejecutable es considerado como argumento. Por otro lado `*argv[]` es un array de punteros a caracteres, cada elemento del array apunta a un argumento de la línea de ordenes.

Si **[4]** el número de parámetros de entrada es distinto de 3 (recordar que el nombre del fichero ejecutable cuenta como parámetro) entonces invoca **[5]** a la llamada al sistema `exit` para terminar la ejecución del programa.

En **[6]** invoca a la llamada al sistema `open` para que abra con permisos de sólo lectura para todos los usuarios (máscara de modo octal 0444) el fichero `file_R`. Esta llamada devuelve un *descriptor del fichero* que es asociado a la variable `fichero1`. Asimismo, en **[7]** invoca a la llamada al sistema `open` para que cree (puesto que no existe en el sistema) con permisos de lectura y escritura para todos los usuarios el fichero `file_W`. Esta llamada devuelve un descriptor del fichero que es asociado a la variable `fichero2`.

Si **[8]** `fichero 1` es igual a -1, es decir, se ha producido un error durante la ejecución de la llamada al sistema `open`, entonces **[9]** imprime en pantalla el mensaje

```
Error al abrir el fichero
```

La misma comprobación se realiza para el descriptor `fichero2` (sentencias **[10]** y **[11]**)

A continuación **[12]** invoca a la llamada al sistema `fork`, para crear un proceso hijo, llama **[13]** a la función `leer_escribir` e invoca **[14]** a la llamada al sistema `exit` para finalizar el programa. Con esta sentencia finaliza el código de la función `main`.

En **[15]** declara la función `leer_escribir` que no recibe ningún parámetro de entrada y devuelve un número entero como salida. En **[16]** ejecuta un bucle infinito, dentro del cual **[17]** invoca a la llamada al sistema `read` que lee un carácter (un byte) del fichero `file_R` (cuyo descriptor es `fichero1`) y lo almacena en la dirección asociada a la variable `carácter`. Además comprueba, que no se ha alcanzado el final del fichero analizando el valor que devuelve la llamada al sistema, que es el número de bytes leídos, en este caso 1. Por eso si devuelve otro valor distinto de 1 habrá llegado al final del fichero. Si llega al final del fichero **[18]** sale del bucle y

devuelve el valor 0. En caso contrario, invoca [19] a la llamada al sistema `write` que escribe en el fichero `file_W` (cuyo descriptor es `fichero2`) el contenido de la variable `carácter`, es decir, 1 byte.

Cuando se ejecuta `ejfork` el sistema le asocia un proceso. A raíz de la sentencia [12] se crea otro proceso, hijo del anterior. Cada proceso (padre e hijo) puede acceder a copias privadas de las variables globales `fichero1`, `fichero2` y `carácter` así como a copias privadas de las variables `argc` y `argv`, pero ninguno puede acceder a las variables del otro proceso. Asimismo, comparten el acceso a los ficheros `file_R` y `file_W`. Además, los dos procesos nunca leen (o escriben) en la misma posición del archivo, ya que el núcleo incrementa el puntero de lectura/escritura para el archivo después de cada llamada a `read` y `write`.

Aunque parece que los procesos copian el archivo `file_R` el doble de rápido ya que comparten la carga de trabajo, el contenido del archivo `file_W` depende del orden en el que el núcleo planifique a los procesos. Si se planifican los procesos de forma que se alterne sus llamadas al sistema, o si se alternan la ejecución del par de llamadas al sistema `read-write`, el contenido del archivo `file_W` será igual al contenido del `file_R`.

Supóngase el caso en el que los procesos van a leer la secuencia de caracteres "ab" del archivo `file_R`, y supóngase también que el proceso padre lee el carácter 'a' y el núcleo hace un cambio de contexto al proceso hijo antes de que el padre escriba el carácter 'a'. Si el proceso hijo lee el carácter 'b' y lo escribe en el archivo `file_W` antes de que se replanifique al padre, el archivo destino no contendrá la cadena "ab" sino la cadena "ba".



5.3 SEÑALES

5.3.1 Generación y tratamiento de señales

Las señales proporcionan un mecanismo para notificar a los procesos los eventos que se producen en el sistema. Los eventos se identifican mediante números enteros, aunque también tiene asignados constantes simbólicas que facilitan su identificación al programador. Algunos de estos eventos son notificaciones asíncronas (por ejemplo, cuando un usuario envía una señal de interrupción a un proceso pulsando simultáneamente las teclas `ctrl+c` en el terminal), mientras que otros son errores síncronos o excepciones (por ejemplo, acceder a una dirección ilegal).

Las señales también se pueden utilizar como un mecanismo de comunicación y sincronización entre procesos.

En el mecanismo de señalización se distinguen dos fases principalmente: *generación* y *recepción* o *tratamiento*. Una señal es generada cuando ocurre un evento que debe ser notificado a un proceso. La señal es recibida o tratada, cuando el proceso para el cual fue enviada la señal reconoce su llegada y toma las acciones apropiadas. Asimismo, se dice que una señal está *pendiente* para el proceso si ha sido generada pero no ha sido tratada todavía.

Señal	Descripción	Acción por defecto	Disponible en	Notas
SIGABRT	Proceso abortado	abortar	APSB	
SIGALRM	Alarma de tiempo real	terminar	OPSB	
SIGBUS	Error en el bus	abortar	OSB	
SIGCHLD	Terminación o suspensión de un proceso hijo	ignorar	OJSB	6
SIGCONT	Continuar un proceso suspendido	continuar/ignorar	JSB	4
SIGEMT	Fallo hardware	abortar	OSB	
SIGFPE	Fallo aritmético	abortar	OAPSB	
SIGHUP	Desconexión	terminar	OPSB	
SIGILL	Instrucción ilegal	abortar	OAPSB	2
SIGINFO	Petición del estado (se genera al pulsar ctrl+t)	ignorar	B	
SIGINT	Interrupción desde el terminal (ctrl+c)	terminar	OAPSB	
SIGIO	Evento de E/S asíncrono	terminar/ignorar	SB	3
SIGIOT	Fallo hardware	abortar	OSB	
SIGKILL	Finalizar un proceso	terminar	OPSB	1
SIGPIPE	Escritura en una tubería que no posee lectores	terminar	OPSB	
SIGPOLL	Evento de E/S asíncrono.	terminar	S	
SIGPROF	Alarma de perfil	terminar	SB	
SIGPWR	Fallo en la alimentación	ignorar	OS	
SIGQUIT	Señal de terminación de sesión en el terminal (ctrl+\)	abortar	OPSB	
SIGSEGV	Violación de segmento	abortar	OAPSB	
SIGSTOP	Parar o suspender un proceso	parar	JSB	1
SIGSYS	Llamada al sistema no válida	terminar	OAPSB	
SIGTERM	Finalizar un proceso	terminar	OASPB	
SIGTRAP	Fallo hardware	abortar	OSB	2
SIGTSTP	Señal de parar desde el terminal (ctrl+z)	parar	JSB	
SIGTTIN	Lectura del terminal desde un proceso en segundo plano	parar	JSB	
SIGTTOU	Escritura del terminal desde un proceso en segundo plano	parar	JSB	5
SIGURG	Evento urgente en canal de E/S	ignorar	SB	
SIGUSR1	Señal definida por el usuario	terminar	OPSB	
SIGUSR2	Señal definida por el usuario	terminar	OPSB	
SIGVTALRM	Alarma de tiempo virtual	terminar	SB	
SIGWINCH	Cambio en el tamaño de la ventana	ignorar	SB	
SIGXCPU	Excedido el tiempo de uso de CPU	abortar	SB	
SIGXFSZ	Excedido el tamaño máximo del fichero	abortar	SB	
Disponibilidad	O Señal original del SVR2 B Señal de BSD4.3 P POSIX.1 A ANSI C S SVR4 J POSIX.1, si soporta control de tareas			
Notas	1 No puede ser capturada, bloqueada o ignorada 2 No puede ser reiniciada a su valor por defecto, incluso en las implementaciones System V 3 Su acción por defecto es terminar en SVR4, ignorar en 4.3BSD. 4 Su acción por defecto es continuar el proceso si es suspendido, sino se ignorar. No puede ser bloqueada 5 El proceso no puede elegir escribir en segundo plano sin generar esta señal 6 Denominada SIGCLD en SVR3 y versiones anteriores.			

Tabla 5.1: Señales en UNIX

En la distribución original del UNIX System V únicamente existían definidas 15 señales distintas. Posteriormente, las distribuciones BSD4 y SVR4 aumentaron el número

de señales definidas a 31 (ver Tabla 5.1). Cada señal tiene asignada un número entero entre 1 y 31 (configurar un número de señal a 0 tiene significados especiales para algunas funciones y llamadas a sistema). Un mismo número puede representar a una señal distinta dependiendo de la distribución de UNIX que se considere. Afortunadamente para los programadores, cada señal tiene asignado una constante simbólica común en todas las distribuciones. Así por ejemplo, la señal SIGSTOP tiene asignado el número 17 en BSD4.3 y el número 23 en SVR4.

5.3.1.1 Generación de señales

El núcleo genera señales para los procesos en respuesta a distintos eventos que pueden ser causados por: el propio proceso receptor, otro proceso, interrupciones o acciones externas. Así, las principales fuentes de generación de señales son:

- *Excepciones.* Cuando durante la ejecución de un proceso se produce una excepción (por ejemplo, un intento de ejecutar una instrucción ilegal), el núcleo se lo notifica al proceso mediante el envío de una señal.
- *Otros procesos.* Un proceso puede enviar una señal a otro proceso, o a un conjunto de procesos, mediante el uso de las llamadas al sistema `kill` o `sigsend`. También, un proceso puede enviarse una señal asimismo.
- *Interrupciones del terminal.* La pulsación simultánea por parte de un usuario de teclas, como `ctrl+c` o `ctrl+\`, produce el envío de señales a los procesos que se encuentran ejecutándose en el primer plano de un terminal.
- *Control de tareas.* Los interpretes de comandos generan señales para manipular tanto a los procesos que se encuentran ejecutándose en primer plano como a los que se encuentran ejecutándose en segundo plano. Cuando un proceso termina o es suspendido, el núcleo se lo notifica a su padre mediante el envío de una señal.
- *Cuotas.* Cuando un proceso excede su tiempo de uso de la CPU o el tamaño máximo de un fichero, el núcleo envía una señal a un proceso.
- *Notificaciones.* Un proceso puede requerir la notificación de ciertos eventos, como por ejemplo que un dispositivo se encuentra listo para realizar una operación de E/S. El núcleo informa al proceso de este evento enviándole una señal.

- **Alarmas.** Un proceso puede configurar una alarma para un cierto tiempo, cuando éste expira, el núcleo se lo notifica enviándole una señal.

5.3.1.2 Tratamiento de las señales

Cada señal tiene asignada una *acción por defecto*, que es la que el núcleo realizará para tratar la señal si el proceso no ha especificado alguna acción alternativa. Existen cinco posibles *acciones por defecto*:

- **Abortar el proceso.** El proceso finaliza después de generar un fichero llamado `core`¹ en el directorio de trabajo actual del proceso. En `core` se escribe el contenido del contexto a nivel de usuario y del contexto de registros. Este fichero puede ser consultado con posterioridad por otros programas, como por ejemplo depuradores. Un proceso es abortado cuando se produce algún error durante su ejecución, como por ejemplo, el acceso a una dirección fuera del espacio de direcciones del proceso o el intento de ejecutar una instrucción ilegal.
- **Finalizar el proceso.**
- **Ignorar la señal.**
- **Parar o suspender el proceso.** Un proceso entra en el estado *parado o suspendido* al recibir una señal de parada como SIGSTOP, SIGTSTP, SIGTTIN o SIGTTOU. Si el proceso estaba en el estado *dormido* cuando se generó la señal de parar, su estado cambia al estado *dormido y parado*.
- **Continuar el proceso.** Un proceso puede ser reanudado mediante una señal de continuar como SIGCONT que devuelve al proceso al estado *preparado para ejecutar*. Si el proceso estaba en el estado *parado y dormido*, SIGCONT devuelve al proceso al estado *dormido*.

Las acciones por defecto *parar* y *continuar* no estaban implementadas en las distribuciones SVR2 y SVR3.

Un proceso puede evitar la realización de la acción por defecto asociada a una determinada señal especificando otra acción alternativa. Esta acción alternativa puede

¹ La distribución BSD4.4 llama a este fichero `core.prog` donde `prog` son los 16 primeros caracteres del fichero ejecutable del que era instancia el proceso.

ser ignorar la señal, o invocar a una función definida por el usuario denominada *manejador de la señal*. Cuando la acción que se realiza al tratar una señal es ejecutar una manejador definido por el usuario para dicha señal se suele decir que la señal ha sido *capturada*. En cualquier momento, el proceso puede especificar una nueva acción, es decir, especificar otro manejador de señal, o asignar de nuevo la acción por defecto.

Es posible que de forma simultánea un mismo proceso tenga pendientes varias señales. En dicho caso las señales son procesadas de una en una. Asimismo, una señal podría llegar durante la ejecución del manejador de otra señal, produciéndose el anidamiento de manejadores.

Un proceso también puede *bloquear o enmascarar*² una señal, en cuyo caso la señal no será tratada hasta que sea desbloqueada. Existen señales especiales, como SIGKILL y SIGSTOP, que los usuarios no pueden ignorar, bloquear, o capturar (especificar un manejador de la señal).

Cualquier acción, incluyendo la terminación del proceso, solamente puede ser realizada por el proceso receptor de la señal. Por lo tanto, el proceso tiene que ser planificado para ser ejecutado. En un sistema con una gran carga de trabajo, si el proceso tiene una baja prioridad de planificación, esto puede tomar algún tiempo. Además habrá un retardo adicional si el proceso se encontraba intercambiado en memoria secundaria, en el estado suspendido, o en el estado dormido no interrumpible.

El proceso receptor se da cuenta de la existencia de la señal cuando el núcleo (en el nombre del proceso) invoca al algoritmo `issig()` para comprobar la existencia de señales pendientes. El núcleo llama a `issig()` únicamente en las siguientes casos:

- Antes de volver al estado *ejecución en modo usuario* desde el estado *ejecución en modo supervisor* después de atender una llamada al sistema o una interrupción.
- Justo antes de entrar en el estado *dormido interrumpible*.
- Inmediatamente después de despertar del estado *dormido interrumpible*.

El algoritmo `issig()` comprueba el campo `p_sig` en la entrada asociada al proceso en la tabla de procesos. Este campo es una máscara o mapa de bits, cada bit está asociado a un tipo de señal. Si el bit asociado a una cierta señal está activado entonces

² El bloqueo de señales no estaba implementado en la distribución SVR2 ni en las anteriores

significa que existe al menos una señal pendiente de ese tipo. Puesto que `p_sig` es solo un mapa de bits con un bit por señal, el núcleo no puede notificar el número de apariciones de una misma señal.

Si existe alguna señal pendiente `issig()` desactiva el bit del campo correspondiente y devuelve `VERDADERO`. En este caso el núcleo llama al algoritmo `psig()` para tratar la señal. Este algoritmo realiza distintas acciones en función de la existencia o no de un manejador definido por el usuario para la señal. Si no existe un manejador definido se ejecuta la acción por defecto asociada a la señal, típicamente finalizar o abortar el proceso.

Si existe un manejador definido, `psig()` llama al algoritmo `sendsig()`. Este algoritmo en primer lugar busca la dirección del manejador en el campo `u_signal` del área U del proceso. Este campo (que es un array) contiene una entrada por cada tipo de señal. Cada entrada puede contener la dirección de inicio del manejador definido por el usuario, o un valor constante como `SIG_DFL` (que indica que se debe realizar la acción por defecto) o `SIG_IGN` (que indica que se debe ignorar la señal).

A continuación `sendsig()` hace los arreglos pertinentes en la pila de capas de contexto asociada al proceso para que éste pueda continuar su ejecución después de que el manejador termine de ejecutarse:

- 1) El núcleo accede a la capa 0 de la pila de capas de contexto del proceso receptor para recuperar los valores del contador del programa y del registro de pila salvados allí. Recuérdese que estos valores permiten retomar la ejecución del proceso en modo usuario.
- 2) El núcleo crea un nuevo marco de pila en la pila de usuario, y escribe en él los valores del contador del programa y del registro de pila que recuperó en el paso anterior. La pila de usuario queda entonces como si el proceso hubiese llamado a una función a nivel de usuario (el manejador de la señal) en el punto donde se hizo la llamada al sistema o donde el núcleo lo había interrumpido (antes de reconocer la señal).

Finalmente, accede al contexto de registros salvado en la capa 0 de la pila de capas de contexto y escribe en el contenido del contador del programa la dirección del manejador de la señal. Además configura el contenido del registro de pila de dicha capa para que tenga en cuenta el crecimiento de la pila de usuario realizado en el paso anterior. De esta forma, cuando el proceso vuelva a modo usuario, se ejecutará el

manejador de la señal. Cuando éste finalice el proceso continuará su ejecución desde el punto donde se hizo la llamada al sistema o donde el núcleo lo había interrumpido (antes de reconocer la señal).

La implementación de `sendsig()` es muy dependiente de la máquina puesto que debe manipular la pila de usuario y salvar, restaurar y modificar el contexto del proceso.

Las señales generadas por eventos asíncronos pueden ser tratadas después de ejecutar cualquier instrucción del código del proceso. Es decir, la llamada al manipulador es asíncrona. Cuando el manipulador de la señal se completa, el proceso continua su ejecución desde el punto donde fue interrumpido por la señal.

Si la señal llega cuando se estaba en modo núcleo ejecutando una llamada al sistema, el núcleo normalmente aborta la llamada al sistema (usando el algoritmo `longjmp()`) y el proceso retorna a modo usuario. Entonces en primer lugar ejecutará el manejador de la señal y luego continuará la ejecución del código normal del proceso, desde el punto en el que había invocado a la llamada al sistema. Pero en este caso no se habrá ejecutado correctamente y por lo tanto la función asociada a la llamada al sistema habrá devuelto el valor -1, en la variable `errno` se tendrá la constante `EINTR`, que significa que la llamada al sistema fue interrumpida por la recepción de una señal. El usuario puede comprobar si la llamada al sistema devolvió este error, para en dicho caso, reiniciar la llamada al sistema. Pero en algunas ocasiones sería más conveniente si el núcleo reiniciará de forma automática la llamada al sistema, como ocurre en las distribuciones BSD.

♦ Ejemplo 5.3:

Supóngase que un proceso se encuentra en modo núcleo ejecutando una llamada al sistema. La pila de capas de contexto asociado a este proceso contiene solamente la capa 0 (ver Figura 5.2) donde se ha salvado el contenido del contexto de registros en modo usuario. Supóngase que el contenido del contador del programa (PC) salvado en esta capa es la dirección hexadecimal `10c`. Esta será la dirección de la próxima instrucción, que en principio, se ejecutará cuando el proceso retorne a modo usuario. Esta instrucción típicamente corresponderá a código de la función de librería asociada a la llamada al sistema.

Por otra parte, supóngase que la pila de usuario del proceso contiene dos marcos, el marco 1 que contiene información de la función `main` del programa y el marco 2 que contiene información de la función de librería asociada a la llamada al sistema.

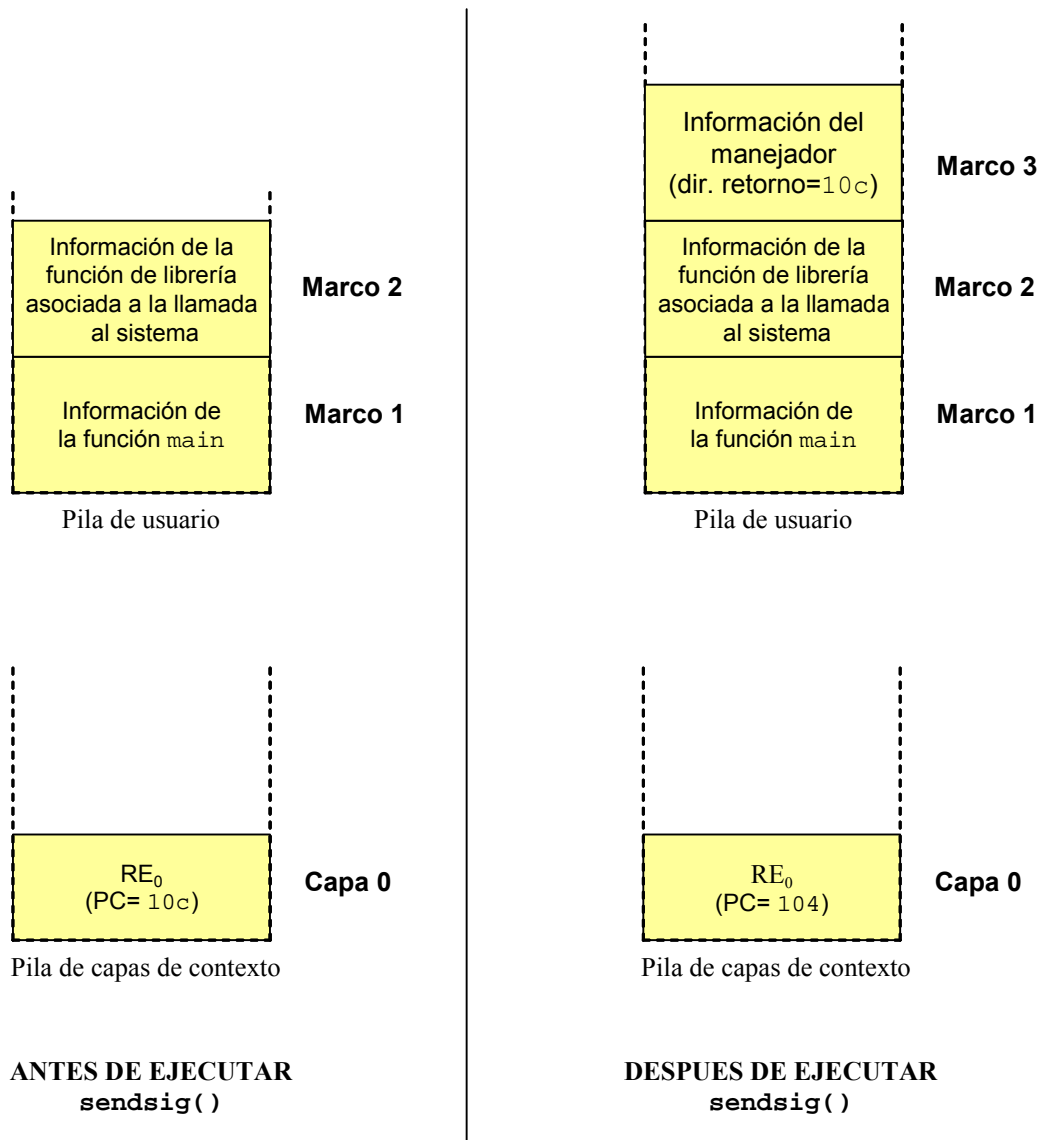


Figura 5.2: Estado de la pila de usuario y de la pila de capas de contexto asociadas a un cierto proceso antes y después de ejecutar el algoritmo `sendsig()`

Finalmente, supóngase que se dan las circunstancias necesarias para que el núcleo tenga que invocar al algoritmo `sendsig()`, es decir, se ha detectado una señal pendiente y existe un manejador definido para dicha señal. La dirección de inicio del manejador (en hexadecimal) se supone que es 104.

En la ejecución de `sendsig()` el núcleo accede a la capa 0 de la pila de capas de contexto del proceso para recuperar entre otras informaciones el contenido salvado del PC, que es la dirección 10c. El núcleo crea el marco 3 en la pila de usuario para el manejador y entre otras acciones establece que la dirección de retorno del manejador sea 10c. Finalmente, accede al contexto de registros salvado en la capa 0 de la pila de capas de contexto y escribe en el contenido del PC la dirección del manejador de la señal que es 104.

De esta forma cuando el proceso vaya a retornar a modo usuario, se recuperará la capa 0 de la pila de capas de contexto y el contexto de registros se inicializará con los valores que estaban almacenados en esta capa. Así el PC se cargará con la dirección 104 y comenzará a ejecutarse el manejador de la señal. Cuando éste finalice su ejecución, se extraerá el marco 3 de la pila de usuario y se continuará la ejecución en la dirección 10c que será una instrucción de la función asociada a la llamada al sistema que devolverá a la función `main` el valor -1 indicando que la llamada al sistema no se ejecutó con éxito. En la variable `errno` se tendrá la constante `EINTR`, que significa que la llamada al sistema fue interrumpida por la recepción de una señal.

♦

5.3.1.3 Escenarios típicos

Supóngase que un usuario pulsa simultáneamente las teclas `control+c` en su terminal. La pulsación de estas teclas (como la de cualquier otra) produce una interrupción del terminal. En el contexto del proceso B actualmente en ejecución, el núcleo invocará al algoritmo de tratamiento de las interrupciones `inthand()`, quién reconocerá e invocará a la rutina de servicio asociada a esta interrupción la cual enviará la señal `SIGINT` al proceso A en el primer plano del terminal. Cuando este proceso (A) sea planificado para ejecución y vaya a retornar al estado *ejecución en modo usuario* el núcleo invocará al algoritmo `issig()` para comprobar la existencia de señales pendientes. Será entonces cuando el proceso se percatará de la existencia de esta señal y ésta sea tratada.

En algunas ocasiones, en el momento de producirse la interrupción el proceso en ejecución es precisamente el proceso en primer plano del terminal. En este caso no es necesario esperar a que el proceso sea planificado, cuando concluya `inthand` y el proceso vaya a retornar al estado *ejecución en modo usuario*, el núcleo invocará al algoritmo `issig()`.

Las *excepciones* son usualmente causadas por un error de programación (división por cero, instrucción ilegal, etc) y siempre ocurren en el mismo punto de ejecución del programa. A diferencia de las interrupciones hardware, las *excepciones* provocan la generación de señales síncronas. Cuando se produce una excepción, se provoca una interrupción software, lo que provocará el paso a modo núcleo y la invocación del algoritmo de tratamiento de las interrupciones `inthand()`, quién reconocerá la excepción e invocará a la rutina de servicio asociada a esta excepción la cual enviará la señal apropiada al proceso actual. Cuando finalice `inthand()` y el proceso vaya a regresar al *estado ejecución en modo usuario*, el núcleo invocará al algoritmo `issig()`.

5.3.2 Problemas de consistencia en el mecanismo de señalización

5.3.2.1 Planteamientos de los problemas

La implementación del mecanismo de señalización que se utilizó en la distribución SVR2 (y anteriores) era poco fiable y defectuosa. Esta implementación aunque sigue el modelo básico descrito en la sección anterior posee varios problemas.

Un problema que presenta esta implementación es que los manejadores de las señales no son persistentes. Supóngase que un usuario instala un manejador para tratar un cierto tipo de señales. Cuando dicha señal es tratada, el núcleo antes de invocar al manipulador establece que la acción, que debe realizar la próxima vez que se genera esta señal, debe ser la acción asociada a dicha señal por defecto (por ejemplo, terminar el proceso). Por lo tanto, si un usuario desea tratar con el mismo manejador las diferentes apariciones de una misma señal deberá reinstalar el manipulador en cada ocasión.

Supóngase que un usuario tiene definido un manejador para la señal SIGINT, y que pulsa `ctrl+c` dos veces. La primera pulsación genera una señal SIGINT, el núcleo cuando va a tratar dicha señal, antes de invocar al manipulador, establece que la acción que debe realizar la próxima vez que se genera esta señal, debe ser la acción asociada a dicha señal por defecto (que de acuerdo con la Tabla 5.1 es terminar el proceso en primer plano del terminal). Si el segundo `control+c` es pulsado antes de que el manipulador sea reinstalado, el núcleo tomará la acción por defecto y terminará el proceso. Existe por tanto una ventana de tiempo entre el instante en que el manipulador es invocado y el instante en que es reinstalado, durante la cual la señal no será capturada.

Otro problema que presenta esta implementación está asociado a los procesos en estado dormido. Toda la información sobre el tratamiento de las señales asociadas a un proceso se encontraba almacenada en el campo `u_signal` de su área U.

Puesto que el núcleo únicamente puede leer el área U del proceso actualmente en ejecución, no existe manera de conocer como otro proceso tratará a una cierta señal. En concreto, si el núcleo ha notificado una señal a un proceso en el estado dormido interrumpible, no puede saber de antemano si este proceso va a ignorar la señal. Así, el núcleo notificará la señal y despertará al proceso, suponiendo que el proceso va a tratar la señal. Cuando el proceso descubre que ha sido despertado por una señal que ignora, volverá de nuevo al estado dormido, lo que genera un cambio de contexto y un

procesamiento innecesarios. El rendimiento del sistema mejoraría si el núcleo pudiera reconocer y descartar las señales que van a ser ignoradas sin tener que despertar al proceso.

Finalmente, otro problema de esta implementación es que carece de la posibilidad de bloquear o enmascarar señales.

5.3.2.2 Soluciones de los problemas de consistencia

Los problemas de consistencia que presentaba el mecanismo de señalización de las distribuciones SVR2 (y anteriores) fueron solventados en primer lugar en la distribución BSD4.2. Asimismo System V los solventó en SVR3. Por lo tanto, estas distribuciones poseen un mecanismo de señalización consistente y fiable que posee las siguientes características:

- *Manejadores Persistentes.* Los manejadores de señales permanecer instalados después de la primera aparición de las señales y no es necesario reinstalarlos.
- *Procesos dormidos.* La información de control de la señales no se encuentra únicamente en el área U, sino que parte de la misma se encuentra en la tabla de procesos. De esta forma, el núcleo puede acceder a dicha información aunque el proceso no se esté ejecutando. Por lo tanto si el núcleo genera una señal para un proceso que está en el estado dormido interrumpible y el proceso va ignorar o a bloquear la señal el núcleo no tiene necesidad de despertarlo.
- *Enmascarado.* Una señal puede ser enmascarada (bloqueada) temporalmente. Si se genera una señal que está enmascarada, el núcleo lo recordará pero no se lo notifica inmediatamente al proceso. Cuando el proceso desbloquee la señal, la señal será notificada y tratada. Esto permite a los programadores proteger ciertas regiones críticas de código de ser interrumpidas por señales.

5.3.2.3 Un ejemplo de mecanismo de señalización consistente

Un mecanismo de señalización consistente es por ejemplo, el implementado en la distribución SVR4, el cual es semejante a la de la distribución BSD4.2, diferenciándose principalmente en los nombres de algunas variables y funciones.

En el área U de un proceso se mantiene distintos campos asociados a los manejadores de las señales, siendo el más importante `u_signal`, que es un vector que contiene la dirección de inicio del manejador asociado a cada señal.

Asimismo, en la entrada de la tabla de procesos asociada a un proceso se mantiene información asociada a la notificación de señales. Los campos más importantes son:

- `p_cursig`. Número de la señal actualmente tratada.
- `p_sig`. Máscara de señales pendientes.
- `p_hold`. Máscara de señales bloqueadas.
- `p_ignore`. Máscara de señales ignoradas.

Cuando una señal es generada, el núcleo comprueba el campo `p_ignore` de la entrada de la tabla de procesos del proceso receptor. Si la señal va a ser ignorada, el núcleo no realiza ninguna acción. En caso contrario, notifica la aparición de la señal activando el bit asociado a la señal en el campo `p_sig`. Puesto que `p_sig` es solo un mapa de bits con un bit por señal, el núcleo no puede notificar el número de apariciones de una misma señal.

Si el proceso está en el estado dormido interrumpible el núcleo comprueba el campo `p_hold` para comprobar si la señal se encuentra bloqueada por el proceso. Si la señal no está bloqueada, el núcleo despierta al proceso para que pueda recibir la señal. Algunas señales de control de tareas como `SIGSTOP` o `SIGCONT` directamente suspenden o continúan la ejecución del proceso sin necesidad de ser notificadas.

Cuando el núcleo invoca al algoritmo `issig()` para comprobar la existencia de señales pendientes. Esta función busca la existencia de bits activados en `p_sig`. Si algún bit está activado, entonces `issig()` comprueba `p_hold` para descubrir si la señal se encuentra actualmente bloqueada. Si no, entonces almacena el número de la señal en `p_cursig` y devuelve VERDADERO.

Si una señal está pendiente, el núcleo llama a `psig()` para tratarla. Este algoritmo inspecciona la información asociada a esta señal en el área U del proceso. Si no se ha definido ningún manejador para la señal, `psig()` realiza la acción por defecto, normalmente finalizar o abortar el proceso. Si se ha definido un manejador `psig()` llama a `sendsig()` que realiza las acciones comentadas en la sección 5.3.1.2.

5.3.3 Llamadas al sistema para el manejo de señales

5.3.3.1 Llamada al sistema kill

La llamada al sistema `kill` permite a un proceso enviar una señal a otro proceso o a un grupo de procesos. Su sintaxis es:

```
resultado = kill(par, señal);
```

Se observa que tiene dos parámetros de entrada:

- `par`. Es un número entero que permite identificar al proceso o conjunto de procesos a los que el núcleo va a enviar una señal, puede tomar los siguientes valores:
 - Si `par > 0`, el núcleo envía la señal al proceso cuyo `pid` sea igual a `par`.
 - Si `par = 0`, el núcleo envía la señal a todos los procesos que pertenezcan al mismo grupo que el proceso emisor.
 - Si `par = -1`, el núcleo envía la señal a todos los procesos cuyo `uid` sea igual al `euid` del proceso emisor. Si el proceso emisor que lo envía tiene el `euid` del superusuario, entonces el núcleo envía la señal a todos los procesos, excepto al proceso intercambiador (`pid=0`) y al proceso inicial (`pid=1`).
 - Si `par < -1`, el núcleo envía la señal a todos los procesos cuyo `gid` sea igual al valor absoluto de `par`.
- `señal`. Es una constante entera que identifica a la señal para la cual el proceso está especificando la acción. También se puede introducir directamente el número asociado a la señal.

Asimismo, `kill` devuelve un único parámetro de salida `resultado` que vale 0 si la llamada al sistema se ejecuta con éxito. En caso contrario, vale -1.

En todos los casos, si el proceso emisor no tiene un `euid` de superusuario, o si el proceso que envía la señal no tiene privilegios sobre el proceso que va a recibir la señal, la llamada al sistema `kill` falla.

♦ **Ejemplo 5.4:**

```

#include <signal.h>
main ()
{
    [1]  int a;
    [2]  if ((a=fork())==0)
        {
    [3]      while (1)
            {
    [4]          printf("pid del proceso hijo=%d \n",getpid());
    [5]          sleep(1);
            }
        }
    [6]  sleep(10);
    [7]  printf("Terminación del proceso con pid= %d\n",a);
    [8]  kill(a,SIGTERM);
}

```

Programa 5.3

El código de la función `main` del Programa 5.3 comienza con la declaración **[1]** de la variable `a` de tipo entero. A continuación se invoca **[2]** a la llamada al sistema `fork` que crea un proceso hijo y cuyo resultado se almacena en la variable `a`. Así para el proceso padre `a` será igual al `pid` del proceso hijo, mientras que para el proceso hijo `a` valdrá 0.

Si `a` es igual a 0 (se está ejecutando el proceso hijo), se entra en un bucle de tipo `while` **[3]** dentro del cual se ejecutan la sentencia **[4]**, que muestra en pantalla el mensaje

"PID del proceso hijo=[pid]"

donde `[pid]` es el valor del `pid` del proceso hijo obtenido mediante la invocación de la llamada al sistema `getpid`, y la sentencia **[5]** que invoca a la llamada al sistema `sleep` para suspender su ejecución durante un segundo. Obsérvese que debido a la condición que rige el bucle, el proceso hijo nunca podrá salir del mismo.

Por otra parte, el proceso padre invoca **[6]** a la llamada al sistema `sleep` para suspender su ejecución durante diez segundos. Transcurrido ese tiempo ejecuta **[7]** que muestra en pantalla el mensaje

"Terminación del proceso con PID=[a]"

donde `[a]` denota el contenido de la variable `a` y se ejecuta la sentencia **[8]** que invoca a la llamada al sistema `kill` para enviar al proceso hijo una señal `SIGTERM` que produce su finalización.

La ejecución del ejecutable asociado a este programa produce la siguiente traza de ejecución en pantalla:

```
pid del proceso hijo =5645
pid del proceso hijo =5645
pid del proceso hijo =5645
pid del proceso hijo =5645
pid del proceso hijo =5645
pid del proceso hijo =5645
pid del proceso hijo =5645
pid del proceso hijo =5645
pid del proceso hijo =5645
pid del proceso hijo =5645
Terminación del proceso con pid=5645
```

Es decir se ejecuta diez veces el código del proceso hijo cuyo *pid* es 5645 antes de que el proceso padre lo finalice enviándole una señal `SIGTERM`.

◆

Una versión mejorada de `kill` es la llamada al sistema `sigsendset` disponible en SVR4.

Por otra parte, existe un comando denominado `kill`, que permite al usuario enviar señales a los procesos a través de la línea de órdenes de la consola del sistema. Su sintaxis es:

```
$ kill -señal pid
```

Donde *señal* es la señal que se desea mandar y *pid* es el *pid* al que se va a enviar la señal. En la sección 3.6.3 se describió el uso de este comando para la terminación de procesos.

5.3.3.2 Llamada al sistema raise

La llamada al sistema `raise` permite a un proceso enviarse una señal a sí mismo. Su sintaxis es:

```
resultado = raise(señal);
```

Se observa que `raise` tiene únicamente un parámetro de entrada `señal`, que es una constante entera que identifica a la señal para la cual el proceso está especificando la acción. Asimismo, `raise` devuelve un único parámetro de salida `resultado` que vale 0 si la llamada al sistema se ejecuta con éxito. En caso contrario, vale -1.

5.3.3.3 Llamada al sistema `signal`

La llamada al sistema `signal` permite especificar el tratamiento de una determinada señal recibida por un proceso. Su sintaxis es:

```
resultado = signal(señal, acción);
```

Se observa que tiene dos parámetros de entrada:

- `señal`. Es una constante entera que identifica a la señal para la cual el proceso está especificando la acción. También se puede introducir directamente el número asociado a la señal.
- `acción`. Este parámetro especifica la acción que se debe realizar cuando se trate la señal, puede tomar los siguientes valores:
 - `SIG_DFL`. Constante entera que indica que la acción a realizar es la acción por defecto asociada a dicha señal
 - `SIG_IGN`. Constante entera que Indica que la señal se debe ignorar.
 - *Dirección del manejador de la señal* definido por el usuario.

Asimismo, `signal` devuelve un único parámetro de salida `resultado` que es la acción que tenía asignada dicha señal antes de ejecutar esta llamada al sistema. Este valor puede ser útil para restaurarlo en cualquier instante posterior. Por otra parte, si se produce algún error durante la ejecución de la llamada al sistema `resultado` tomará el valor `SIG_ERR` (constante entera asociada al valor -1).

♦ Ejemplo 5.5:

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

void manejador(int sig);
```

```
main()
{
[1]   if (signal(SIGUSR1,manejador)==SIG_ERR)
        {
[2]       perror("\nError");
[3]       exit(1);
        }
[4]   for (;;)

[5] void manejador(int sig)
{
[6]   printf ("\n\n%s recibida. \n",strsignal(sig));
[7]   exit(2);
}
```

Programa 5.4

En el Programa 5.4 se presenta un ejemplo de uso de la llamada al sistema `signal`. La primera acción que se realiza dentro de la función `main` es invocar **[1]** a la llamada al sistema `signal`. Sus parámetros de entrada están especificando que cuando el proceso reciba la señal `SIGUSR1` la acción que debe realiza el núcleo es ejecutar la función `manejador`. Si durante la ejecución de esta llamada al sistema se produce algún error, `signal` devolverá el valor `SIG_ERR`. En ese caso, se imprimirá **[2]** en pantalla el mensaje:

Error: [Texto error]

Donde `[Texto error]` es el mensaje de texto asociado al identificador de error contenido en la variable `errno`. Y a continuación **[3]** se invocará a la llamada al sistema `exit` para terminar el proceso. Si `signal` se ejecuta correctamente el programa entra **[4]** en un bucle infinito.

Por otro parte, la función `manejador` recibe como entrada el número de la señal `sig` y no posee ningún parámetro de salida. Esta función lo único que hace es mostrar **[6]** en la pantalla el mensaje

"[Texto señal] recibida."

donde `[Texto señal]` es el mensaje de texto que describe la señal y que se obtiene invocando a la función `strsignal`. Y a continuación invocar **[7]** a la llamada al sistema `exit` para terminar el proceso.

Supóngase que el ejecutable que resulta de compilar este programa se llama `ex_signal` y que un usuario lo invoca desde un terminal de la siguiente forma:

```
$ ex_signal &
```

Este proceso se ejecutará en segundo plano del terminal. Al ejecutar la sentencia anterior en la pantalla aparece el número de tarea y el *pid* que asigna el sistema a este programa:

```
[1] 5565
```

Obsérvese que si este programa se ejecutará en primer plano la línea de comandos no estaría disponible ya que el programa entra en un bucle `for` infinito y no se podría interactuar con él al no ser que se suspendiese su ejecución pulsando las teclas `ctrl+z`.

Se puede ejecutar el comando `jobs` para comprobar que el proceso efectivamente se está ejecutando

```
$ jobs
```

En pantalla aparece lo siguiente:

```
[1]+  Running                  ex_signal &
```

A continuación haciendo uso del comando `kill` se envía la señal `SIGUSR1` al proceso (su *pid* es 5565). Se debe escribir la siguiente sentencia:

```
$ kill -SIGUSR1 5565
```

De acuerdo con el código del programa al enviar una señal del tipo `SIGUSR1` cuando está sea tratada se ejecutará el manejador definido para dicha señal. Por lo tanto en pantalla aparece el mensaje.

```
User defined signal 1 recibida.
```

y el programa finaliza. Esto se puede comprobar escribiendo:

```
$ jobs
```

En pantalla aparece lo siguiente:

```
[1]+ Terminated              ex_signal
```

◆

Otras llamadas al sistema semejantes a `signal` pero con una mayor versatilidad son `sigaction` (disponible en SVR4) y `sigvec` (disponible en BSD).

5.3.3.4 Llamada al sistema `pause`

La llamada al sistema `pause` hace que el proceso que la invoca quede a la espera de la recepción de una señal que no ignore o que no tenga bloqueada. Su sintaxis es:

```
resultado=pause();
```

Se observa que no requiere parámetros de entrada y que posee un único parámetro de salida `resultado` que vale -1 si la llamada al sistema se ejecuta con éxito. En otras llamadas al sistema, ésta es una condición de error, pero en el caso de `pause` es su forma correcta de operar.

Cuando un proceso que ha invocado a la llamada al sistema `pause` reciba una señal que no ignore o que no tenga bloqueada, al retornar al modo usuario en primer lugar el núcleo ejecutará la acción asociada a la señal. Luego concluirá la función de librería asociada a la llamada al sistema `pause` que devuelve el valor -1 al proceso. En la variable `errno` se tendrá la constante `EINTR`, que significa que la llamada al sistema fue interrumpida por la recepción de una señal.

♦ **Ejemplo 5.6:**

```
main()
{
    pause();
    printf("\nFinal\n");
}
```

Programa 5.5

Supóngase que el ejecutable que resulta de ejecutar el Programa 5.5 se llama `ex_pause` y que se invoca desde la línea de comandos para que sea ejecutado en segundo plano:

```
$ ex_pause &
```

En la pantalla aparece el número de tarea y el *pid* que asigna el sistema a este programa:

```
[1] 10023
```

A continuación haciendo uso del comando `kill` se envía la señal `SIGCHLD`:

```
$ kill -SIGCHLD 10023
```

Al recibir la señal `SIGCHLD`, el núcleo puesto que no se ha definido un manejador realiza la acción por defecto asociada a la misma, que según la Tabla 5.1 es ignorar la señal. Luego el proceso seguirá a la espera de recibir alguna otra señal que no ignore.

Si ahora se envía al proceso la señal `SIGTERM`:

```
$ kill -SIGTERM 10023
```

Al recibir la señal SIGTERM, el núcleo puesto que no se ha definido un manejador realiza la acción por defecto asociada a la misma, que según la Tabla 5.1 es terminar el proceso.

◆

Otras llamadas al sistema semejantes a `pause` pero de mayor versatilidad son `sigpause` (BSD4.3) y `sigsuspend` (SVR4).

5.3.3.5 Llamadas al sistema `sigsetmask` y `sigblock`

La llamada al sistema `sigsetmask` fija la máscara actual de señales, es decir, permite especificar que señales van a estar bloqueadas. Obviamente, aquellas señales que no pueden ser ignoradas ni capturadas, tampoco van a poder ser bloqueadas. Su sintaxis es:

```
resultado=sigsetmask(máscara);
```

Se observa que tiene un único parámetro de entrada `máscara` que es un entero largo asociado a la máscara de señales. Se considera que la señal número j está bloqueada si el j -ésimo bit de `máscara` está a 1. Este bit puede ser fijado con la macro `sigmask(j)`.

Asimismo se observa que posee un único parámetro de salida `resultado` que es la máscara de señales que se tenía especificada antes de ejecutar esta llamada al sistema. En caso de error `resultado` vale -1.

Por otra parte, la llamada al sistema `sigblock` permite añadir nuevas señales bloqueadas a la máscara actual de señales. Su sintaxis:

```
resultado=sigblock(máscara2);
```

Se observa que tiene un único parámetro de entrada `máscara2` que es un entero largo que se utilizará como operando junto con la máscara actual de señales `máscara` para realizar una operación lógica de tipo OR a nivel de bits:

```
máscara = máscara | máscara 2;
```

Se considera que la señal número j está bloqueada si el j -ésimo bit de `máscara2` está a 1. Este bit puede ser fijado con la macro `sigmask(j)`.

Asimismo se observa que `sigblock` posee un único parámetro de salida `resultado` que es la máscara de señales que se tenía especificada antes de ejecutar esta llamada al sistema. En caso de error `resultado` vale -1.

La principal diferencia entre `sigsetmask` y `sigblock` es que la primera fija la máscara de señales de forma absoluta, y la segunda, de forma relativa.

♦ **Ejemplo 5.7:**

```
#include <signal.h>

main()
{
    [1]  long mask0;
    [2]  mask0=sigsetmask(sigmask(SIGUSR1) | sigmask(SIGUSR2));
    [3]  sigblock(sigmask(SIGINT));
    [4]  sigsetmask(mask0);
}
```

Programa 5.6

En la sentencia **[1]** se declara la variable `mask0` de tipo entero largo. En la sentencia **[2]** se invoca a `sigsetmask` para bloquear la recepción de las señales del tipo `SIGUSR1` y `SIGUSR2`. En la variable `mask0` se almacena la máscara de señales original que se tenía especificada antes de invocar a esta llamada al sistema.

En la sentencia **[3]** se invoca a `sigblock` para añadir las señales del tipo `SIGINT` al grupo de señales bloqueadas. Finalmente en **[4]** se vuelve a invocar a `sigsetmask` para restaurar la máscara original de señales, es decir, sin tener bloqueadas a las señales del tipo `SIGUSR1`, `SIGUSR2` y `SIGINT`.

♦

Otras llamada al sistema para el manejo de la máscara de señales es `sigprocmask` (SVR4).

5.4 DORMIR Y DESPERTAR A UN PROCESO

5.4.1 Algoritmo `sleep()`

El núcleo usa el algoritmo `sleep()` para pasar a un proceso A al estado dormido. Este algoritmo requiere como parámetros de entrada la *prioridad para dormir* y la *dirección de dormir o canal* asociada al evento por el que estará esperando el proceso.

La primera acción que realiza `sleep()` es salvar el nivel de prioridad de interrupción (*npi*) actual, típicamente en el registro de estado del procesador. A continuación eleva el *npi* para bloquear todas las interrupciones.

Posteriormente en los campos correspondientes de la entrada de la tabla de procesos asociada al proceso A marca el estado del proceso a *dormido en memoria principal*, salva el valor de la *prioridad para dormir* y de la *dirección de dormir*. Asimismo coloca al proceso en una lista de procesos dormidos.

A continuación compara la *prioridad para dormir* con un cierto valor umbral para averiguar si el proceso puede ser interrumpido por señales. Si la prioridad para dormir es mayor que dicho valor umbral entonces el proceso no puede ser interrumpido por señales. En caso contrario el proceso si puede ser interrumpido por señales. Se distinguen por tanto dos casos:

- Caso 1: *El proceso no puede ser interrumpido por señales*. En este caso el núcleo realiza un cambio de contexto, en consecuencia otro proceso B pasará a ser ejecutado. De esta forma la ejecución del algoritmo `sleep()` es momentáneamente detenida. Más tarde, cuando el proceso A sea despertado y planificado para ejecución, continuará su ejecución en modo núcleo en la siguiente instrucción del algoritmo `sleep()`, que consiste en restaurar el valor del *npi* al valor que tenía antes de comenzar a ejecutar el algoritmo. A continuación, el algoritmo finaliza.
- Caso 2: *El proceso puede ser interrumpido por señales*. En este caso el núcleo invoca al algoritmo `issig()` para comprobar la existencia de señales pendientes. Se pueden dar dos casos:
 - Caso 2.1: *Existen señales pendientes*. Entonces el núcleo borra al proceso A de la lista de procesos dormidos, restaura el valor del *npi* (al valor que tenía antes de comenzar a ejecutar el algoritmo) e invoca al algoritmo `psig()` para tratar la señal.

- **Caso 2.2: No existen señales pendientes.** Entonces el núcleo realiza un cambio de contexto, en consecuencia otro proceso D pasará a ser ejecutado. De esta forma la ejecución del algoritmo `sleep()` es momentáneamente detenida. Más tarde, cuando el proceso A sea despertado (bien por se produjo el evento por el que estaba esperando o porque es interrumpido por una señal) y planificado para ejecución, el núcleo invocará nuevamente al algoritmo `issig()` para comprobar la existencia de señales pendientes que han podido ser notificadas durante el tiempo que pasó dormido. Existen dos posibilidades:

2.2.1) Si no existen señales pendientes, entonces el núcleo restaura el *npi* al valor que tenía antes de comenzar a ejecutar `sleep()` y finaliza el algoritmo.

2.2.2) Existen señales pendientes, entonces el núcleo restaura el *npi* al valor que tenía antes de comenzar a ejecutar el algoritmo `sleep()` e invoca a `psig()` para tratar la señal,

En la Figura 5.3 se resumen las principales acciones que realiza el núcleo durante la ejecución del algoritmo `sleep()`.

De la descripción realizada del algoritmo `sleep()` se deducen las siguientes conclusiones:

- Al contrario de lo que podría pensarse, el algoritmo `sleep()` no requiere ejecutarse hasta el final para lograr su objetivo de pasar a un proceso al estado dormido. Un proceso entra formalmente en el estado dormido cuando dentro del algoritmo se ejecuta el paso del cambio de contexto, momento en el que se suspende la ejecución del algoritmo. En conclusión en el caso 2.1, debido a la existencia de señales pendientes, el proceso nunca llega a entrar formalmente en el estado dormido.
- La pila de capas de contexto de un proceso dormido contiene dos capas de contexto, la capa 1 que contiene la información necesaria para poder continuar con la ejecución del algoritmo `sleep()`, y la capa 0 que contiene la información necesaria para poder retomar la ejecución del proceso en modo usuario.

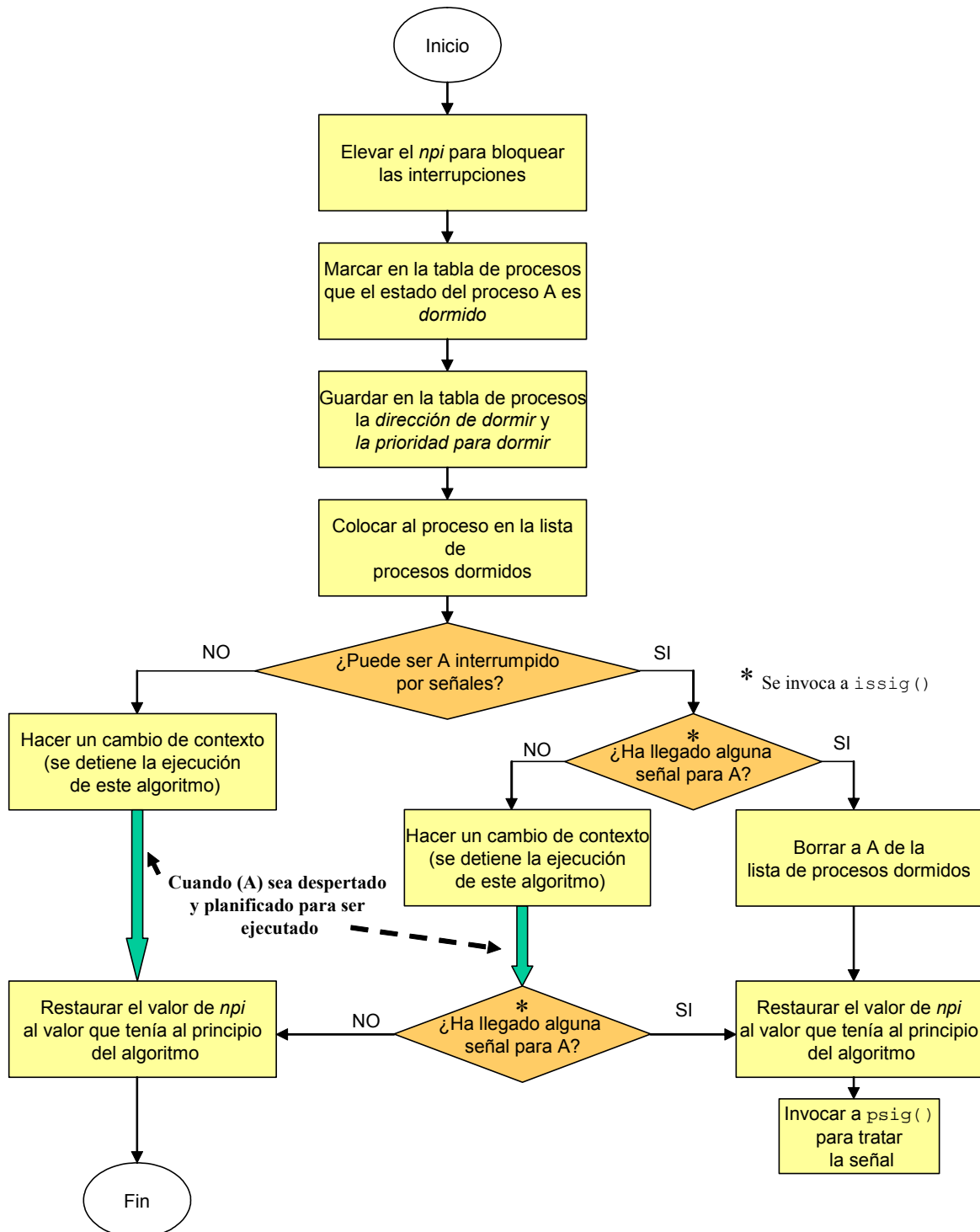


Figura 5.3: Principales acciones realizadas por el núcleo durante la ejecución del algoritmo

`sleep()`

- En este algoritmo se dan dos de las tres situaciones (ver sección 5.3.1.2) en los cuales el núcleo invoca al algoritmo `issig()` para comprobar la existencia de señales pendientes:

- Justo antes de entrar en el estado *dormido interrumpible* (caso 2).

- Inmediatamente después de despertar porque se produjo el evento por el que estaba esperando o porque es interrumpido por una señal (caso 2.2).
- Si se genera una señal para A mientras éste se encuentra en el estado dormido no interrumpible, la señal será marcada como pendiente, pero el proceso no se dará cuenta de la existencia de esta señal hasta que no vuelva al estado ejecución en modo usuario o entre en el estado dormido interrumpible.

5.4.2 Algoritmo `wakeup()`

El núcleo usa el algoritmo `wakeup()` para despertar a un proceso que se encuentra en el estado *dormido* a la espera de la aparición de un determinado evento. Típicamente la invocación de `wakeup()` se realiza dentro de algún otro algoritmo del núcleo como los asociados a las llamadas al sistema o las rutinas de manipulación de interrupciones. Por ejemplo, si el núcleo usa el algoritmo `iput()` para liberar a un nodo-i que estaba bloqueado deberá invocar dentro del mismo a `wakeup()` para despertar a aquellos procesos que estaban esperando por la liberación de dicho nodo-i. Asimismo durante la ejecución de la rutina de tratamiento de una interrupción del disco duro, el núcleo deberá invocar al algoritmo `wakeup()` para despertar a aquellos procesos que estaban esperando que se completará una operación de E/S con el disco.

El núcleo también llama al algoritmo `wakeup()` cuando genera una señal para un proceso en estado dormido interrumpible, siempre y cuando el proceso no ignore ni tenga bloqueadas dicho tipo de señales.

El algoritmo `wakeup()` requiere como parámetro de entrada la *dirección de dormir o canal* asociada a dicho evento. La primera acción que realiza `wakeup()` es salvar el nivel de prioridad de interrupción (*npi*) actual. A continuación eleva el *npi* para bloquear todas las interrupciones.

Posteriormente, busca en la lista de procesos dormidos a aquellos procesos que están a la espera de la aparición del evento asociado a la *dirección de dormir*. Para cada uno de estos procesos realiza las siguientes acciones: elimina al proceso de la lista de procesos dormidos, marca en el campo *estado* de su entrada asociada en la tabla de procesos el estado de *preparado para ejecución en memoria principal (o en memoria secundaria)*, coloca al proceso en una lista de procesos elegibles para ser planificados y

borra el contenido del campo *dirección de dormir o canal* de su entrada asociada en la tabla de procesos.

Además, si el proceso A que ha sido despertado no estaba cargado en memoria principal, el núcleo despierta al *proceso intercambiador* para intercambiar al proceso A a memoria principal desde memoria secundaria (supuesto que la política de gestión de memoria es de intercambio). En caso contrario (el proceso A estaba cargado en memoria principal) si el proceso despertado A es más elegible para ser ejecutado que el proceso actualmente en ejecución D, entonces el núcleo activa un indicador denominado *runrun* en la entrada de la tabla de procesos asociada a A para que esta circunstancia sea tomada en cuenta por el algoritmo de planificación cuando el proceso vaya a retornar al modo usuario.

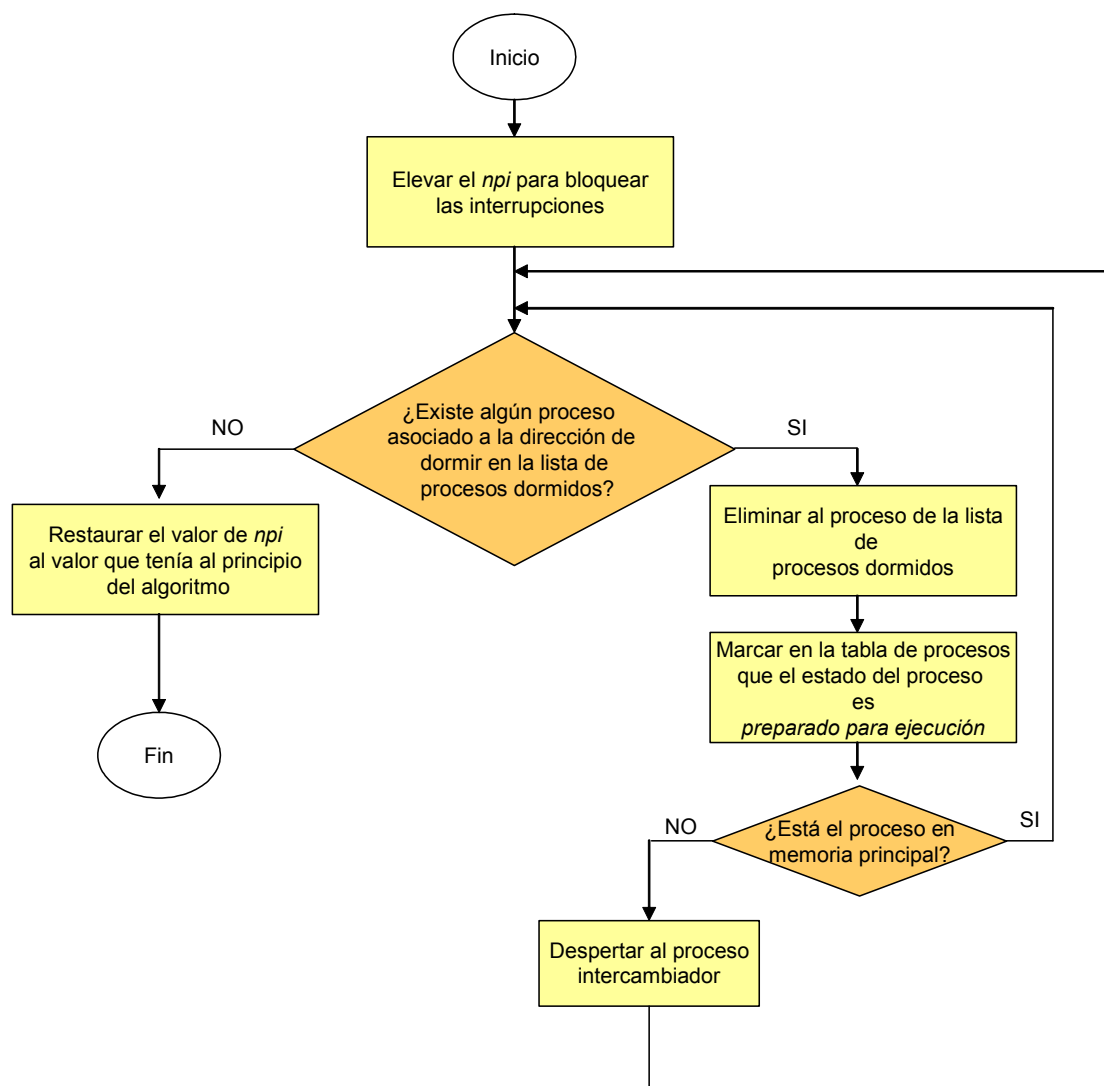


Figura 5.4: Principales acciones realizadas por el núcleo durante la ejecución del algoritmo

wakeup ()

Cuando ya no quedan más procesos en la lista de procesos dormidos a la espera de la aparición del evento asociado a la *dirección de dormir* el núcleo restaura el *npi* al valor que tenía antes de comenzar a ejecutar `wakeup()` y el algoritmo finaliza.

En la Figura 5.4 se resumen las principales acciones que realiza el núcleo durante la ejecución del algoritmo `wakeup()`.

Debe quedar claro que el algoritmo `wakeup()` no hace que un proceso sea inmediatamente planificado; sólo hace que el proceso sea elegible para ser planificado.

5.5 TERMINACION DE PROCESOS

En un sistema UNIX un proceso finaliza cuando se ejecuta la llamada al sistema `exit`. Cuando un proceso B invoca a esta llamada el núcleo lo pasa al estado *zombi*, y elimina todo su contexto excepto su entrada en la tabla de procesos. La sintaxis de esta llamada al sistema es:

```
exit(condición);
```

Se observa que posee un único parámetro de entrada *condición* que es número entero que será devuelto al proceso padre del proceso B. Al parámetro *condición* se le suele denominar *código de retorno para el proceso padre*. El proceso padre puede examinar, si lo desea, el valor de *condición* para identificar la causa por la que finalizó el proceso B de acuerdo a unos criterios que haya previamente establecido el usuario. Así por ejemplo, se podría establecer como criterio que si *condición*=0 el proceso finalizó normalmente, mientras que si *condición*=1 el proceso finalizó porque se produjo algún error durante su ejecución. También es posible no fijar ningún criterio por lo que el valor de *condición* no tendrá ningún significado en especial.

Asimismo se observa que `exit` es de las pocas llamadas al sistema que no genera parámetros de salida. Esto es lógico, ya que el proceso B que la había invocado deja de existir después de haber ejecutado `exit`.

Un proceso puede invocar a la llamada al sistema `exit` explícitamente como una sentencia de su código. Asimismo los programas escritos en C llaman a `exit` implícitamente cuando un programa finaliza su función `main`. En ambos casos el núcleo ejecuta el algoritmo `exit`.

Alternativamente, el núcleo puede invocar al algoritmo `exit` internamente, como por ejemplo para terminar a un proceso durante el tratamiento de una señal no capturada. En ese caso el *código de retorno para el proceso padre* es el número de dicha señal.

El algoritmo `exit` requiere como parámetro de entrada el *código de retorno para el proceso padre*. La primera acción que realiza el núcleo durante la ejecución del algoritmo `exit` es deshabilitar el tratamiento de las señales para el proceso, puesto que como el proceso va a finalizar ya no tiene sentido tratar una señal.

A continuación, el núcleo recorre la tabla de descriptores de ficheros asociada al proceso para ir cerrando todos los ficheros abiertos por el proceso. Además libera el nodo-i del directorio de trabajo actual y el nodo-i del directorio raíz (si éste se hubiese cambiado).

Después el núcleo libera la memoria principal usada por el proceso, utilizando los algoritmos `detachreg()` y `freereg()` sobre las regiones asociadas al proceso. Asimismo en la entrada de la tabla de procesos asociada al proceso B cambia el estado del proceso a *zombi* y salva el *código de retorno para el proceso padre* y otras informaciones de tipo estadístico (tiempo de ejecución en modo usuario, tiempo de ejecución en modo núcleo, etc). También se escribe en un fichero de contabilidad global la información estadística sobre la ejecución del proceso, como por ejemplo: el *uid*, el uso de la CPU, el uso de la memoria, el uso de los recursos de E/S, etc. Estos datos podrán ser leídos con posterioridad por otros programas de monitorización del sistema.

Por otra parte, el núcleo desconecta al proceso B del árbol de procesos y hace que el *proceso inicial* (*pid*=1) adopte a los procesos hijos de B (si los tuviera), para ello configura adecuadamente el campo información genealógica de la entrada asociada a cada proceso hijo en la tabla de procesos. En consecuencia, el *proceso inicial* se convierte en el padre legal de todos los hijos vivos que el proceso B haya creado.

Si existen procesos hijos del proceso B en estado *zombi*, entonces el núcleo envía una señal `SIGCHLD` al *proceso inicial*, para que éste borre los contenidos de sus entradas de la tabla de procesos.

El núcleo también envía una señal `SIGCHLD` al proceso padre del proceso B. Esta señal es ignorada por defecto, y sólo tendrá efecto si el padre deseaba conocer la muerte de su hijo. En un escenario típico, el proceso padre se encuentra ejecutando una llamada al sistema `wait` a la espera de que un proceso hijo termine para proseguir su ejecución, tal y como se describirá en la siguiente sección.

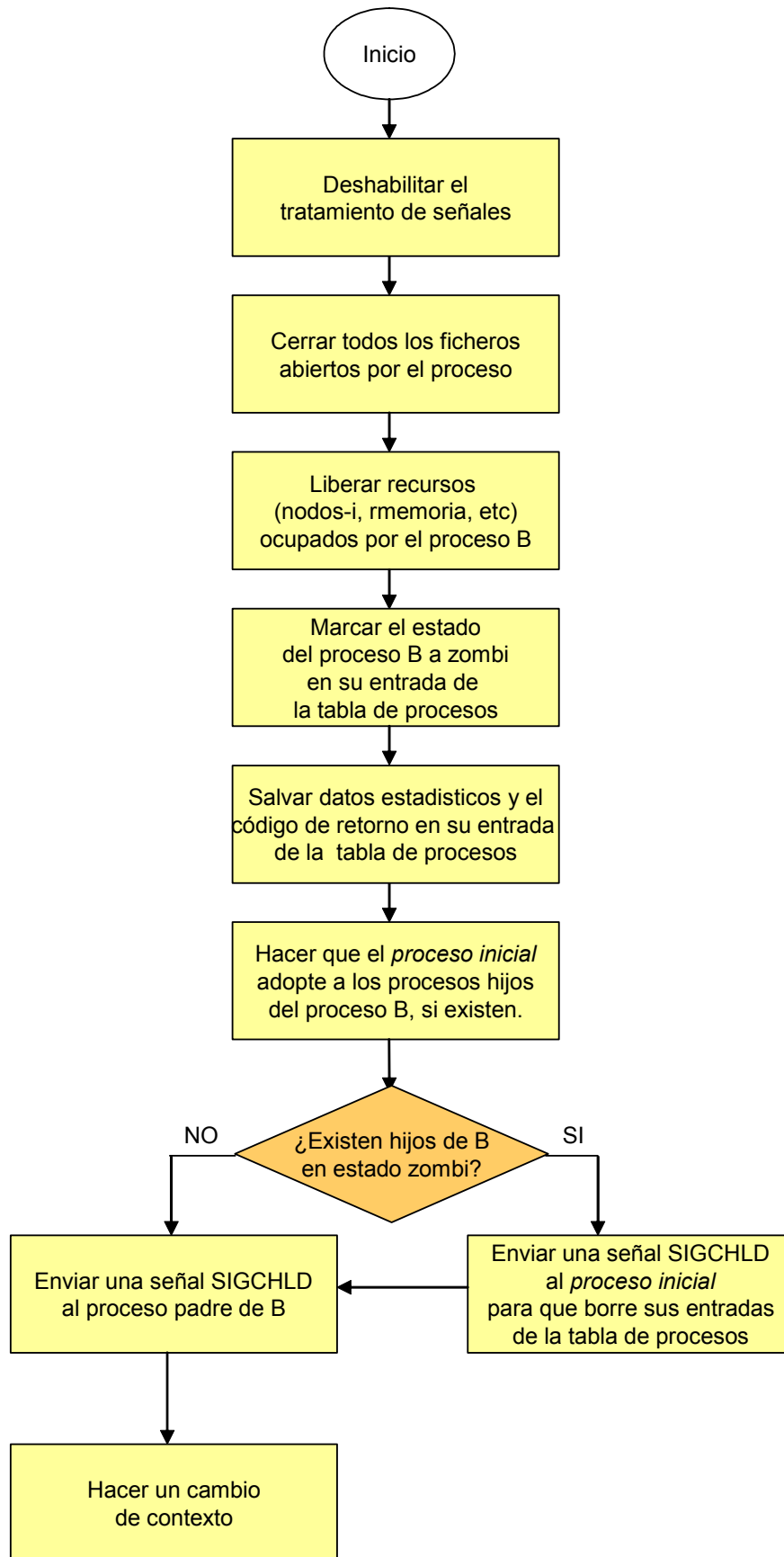


Figura 5.5: Principales acciones realizadas por el núcleo durante la ejecución del algoritmo `exit`

Finalmente el núcleo hace un cambio de contexto, con lo que se pasa a ejecutar otro proceso que haya sido previamente planificado.

En la Figura 5.5 se resumen las principales acciones que realiza el núcleo durante la ejecución del algoritmo `exit`.

5.6 ESPERAR LA TERMINACIÓN DE UN PROCESO

Un proceso A puede sincronizar su ejecución con la terminación de un proceso hijo ejecutando la llamada al sistema `wait`. La sintaxis de esta llamada es:

```
resultado = wait(direc);
```

Se observa que posee un único parámetro de entrada `direc` que es la dirección de una variable entera donde se almacenará el *código de retorno para el proceso padre* generado por el algoritmo `exit` al terminar un proceso hijo.

Asimismo se observa que `wait` posee un único parámetro de salida `resultado` que contiene, si la llamada al sistema se ha ejecutado con éxito, el *pid* del proceso hijo que ha terminado. En caso contrario, contiene el valor -1.

El algoritmo del núcleo o rutina del núcleo asociado a esta llamada al sistema es `wait`, que requiere como parámetro de entrada la dirección de la variable donde se va almacenar el código de retorno para el proceso padre.

La primera acción que realiza el núcleo durante la ejecución del algoritmo `wait` es comprobar que el proceso A posee algún proceso hijo. Si no tiene ningún hijo el algoritmo finaliza y devuelve un error.

En caso contrario, si A posee algún proceso hijo se entra en un bucle, dentro del cual el núcleo comprueba la existencia de procesos hijos de A en estado zombi. Si existe alguno, el núcleo escoge al primero que encuentra (supóngase que es el proceso B), y extrae de la entrada asociada a B en la tabla de procesos, su *pid* y su código de retorno para el proceso padre. A continuación, el núcleo añade, en el campo apropiado del área U del proceso A, el tiempo que el proceso hijo B se ejecutó en modo usuario y en modo núcleo. Finalmente, el núcleo borra los contenidos de la entrada de la tabla de procesos asociada al proceso hijo B, dicha entrada estará ahora disponible para nuevos procesos. El algoritmo finaliza devolviendo al proceso padre el *pid del proceso hijo* y el código de retorno.

Por el contrario, si no existe ninguno proceso hijo de A en estado zombi, el núcleo invoca al algoritmo `sleep()` para pasar al proceso al estado dormido interrumpible en espera de la llegada de alguna señal.

Como se ha descrito en la sección anterior, una de las acciones que realiza el núcleo cuando ejecuta el algoritmo `exit` para finalizar un cierto proceso (supóngase que es el proceso B) es generar una señal `SIGCHLD` para el proceso padre de B (supóngase que es el proceso A). A continuación invoca al algoritmo `wakeup()` para despertar al proceso A.

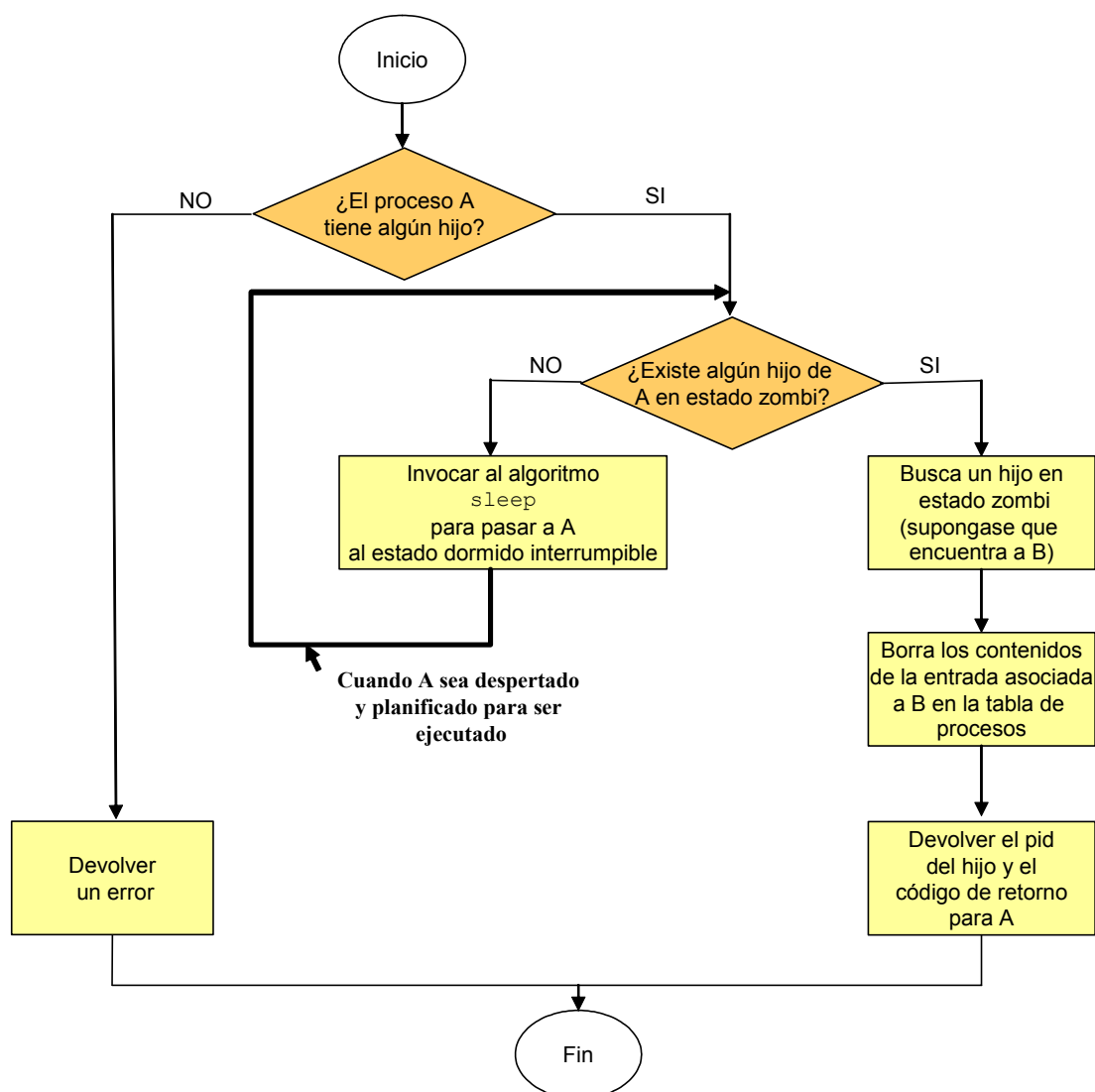


Figura 5.6: Principales acciones realizadas por el núcleo durante la ejecución del algoritmo `wait`

Cuando el proceso A sea planificado para ejecución, éste continuará su ejecución dentro del algoritmo `sleep()`. Por lo tanto, de acuerdo con lo explicado en la sección 5.4.1 se invocará al algoritmo `issig()` para comprobar la existencia de señales pendientes. Al menos existirá una señal pendiente, la señal `SIGCHLD` enviada al terminar

al proceso hijo B. Por lo tanto `issig()` devolverá VERDADERO. Así que el núcleo invocará al algoritmo `psig()` para tratar la señal.

El algoritmo `psig()`, si no existe un manejador definido para tratar este tipo de señales, realiza la acción por defecto, que para este tipo de señales es ignorarla. Por lo tanto, el núcleo continúa con la ejecución del algoritmo `wait`. Recuérdese que el proceso A se puso a dormir dentro de un bucle del algoritmo `wait`. En consecuencia, el núcleo vuelve a comprobar la existencia de procesos hijos de A en estado zombi. En este caso, ya existirá al menos uno, el proceso B. Por lo que se realizarán las acciones comentadas con anterioridad y el algoritmo `wait` finalizará devolviendo el *pid del proceso hijo* y el código de retorno para el proceso padre.

En la Figura 5.6 se resumen las principales acciones que realiza el núcleo durante la ejecución del algoritmo `wait`.

♦ Ejemplo 5.8:

Considérese el siguiente programa escrito en C:

```
main()
{
    int par, estado;
[1]   if(fork()==0)
    {
[2]       printf("\nMensaje 1\n");
[3]       sleep(4);
[4]       exit(3);
    }
    else
    {
[5]       par=wait(&estado);
[6]       printf("\nFinalizar");
    }
}
```

Programa 5.7

En primer lugar, se invoca [1] a la llamada al sistema `fork` para crear un proceso hijo. Recuérdese que cuando finaliza la llamada `fork` devuelve un cero para el proceso hijo. En dicho caso se cumpliría la condición del `if` y el hijo ejecuta en exclusiva, cuando sea planificado, las sentencias [2], [3] y [4]. Es decir, se imprimiría en pantalla el mensaje: `Mensaje 1`, el proceso

hijo suspende su ejecución [3] durante cuatro segundos, y a continuación [4] finaliza devolviendo como código de retorno para el proceso padre el valor 3, que éste recibe del sistema en [5] a través de la variable `estado`, que tomará el valor `estado=3`. La llamada al sistema `wait` devuelve en la variable `par` el valor del identificador `pid` del proceso hijo que ha finalizado. A continuación el proceso padre ejecutará [6] imprimiendo en pantalla el mensaje `Finalizar`.

De esta forma gracias a la llamada al sistema `wait` el proceso padre sincroniza su ejecución con la finalización de su proceso hijo. Es decir, usando `wait` se asegura que no se ejecutarán las sentencias colocadas a continuación de ésta llamada al sistema hasta que no finalice el proceso hijo.

♦

5.7 INVOCACION DE OTROS PROGRAMAS

La llamada al sistema `exec` sirve para invocar desde un proceso a otro programa ejecutable. Básicamente `exec` carga las regiones de código, datos y pila del nuevo programa en el contexto de usuario del proceso que invoca a `exec`. Por lo tanto, una vez concluida esta llamada al sistema, ya no se podrá acceder a las regiones de código, datos y pila del proceso invocador, ya que han sido sustituidas por las del programa invocado.

Existen toda una familia de funciones de librería asociadas a esta llamada al sistema: `execl`, `execv`, `execle`, `execve`, `execlp` y `execvp`. La principal diferencia entre todas ellas radica en sus parámetros de entrada. La función que presenta una sintaxis más sencilla es `execv`:

```
resultado=execv(ruta,argv);
```

donde `ruta` es la ruta (absoluta o relativa) del archivo ejecutable que es invocado, y `argv` es un array de cadenas de caracteres que constituyen la lista de parámetros para el programa ejecutable. Si la llamada al sistema no se realiza con éxito, entonces en `resultado` se almacenará el valor -1.

Otras funciones de biblioteca asociadas a `exec`, como por ejemplo `execve`, cuya sintaxis es

```
resultado=execve(ruta,argv,entorno);
```

que también permiten introducir como parámetro de entrada (`entorno`) un array de punteros a cadenas de caracteres asociadas a las variables de entorno.

Al tratar esta llamada al sistema el núcleo invoca a la rutina o algoritmo `exec`. Este algoritmo requiere como parámetros de entrada, la ruta, la lista de parámetros y las variables de entorno del fichero ejecutable.

La primera acción que realiza el núcleo al ejecutar este algoritmo es usando la ruta encontrar el nodo-i del archivo. A continuación verifica que el proceso invocador tiene permiso para ejecutar el fichero, y acto seguido lee la cabecera del archivo para comprobar que efectivamente se trata de un fichero ejecutable válido. Si el fichero tiene en su máscara de modo los bits `S_ISUID` o `S_ISGID` activados, entonces cambia el `euid` o el `egid`, respectivamente, del proceso invocador para que sean iguales al `uid` o `gid` del propietario del fichero.

Puesto que el contexto a nivel de usuario del proceso invocador va a ser destruido, es necesario salvar en un área de memoria del núcleo los parámetros de entrada de la función de biblioteca asociada a la llamada al sistema `exec` que se encontraban almacenados en la región de datos. Acto seguido, desliga (algoritmo `detachreg()`) del proceso invocador las regiones que conforman su contexto a nivel de usuario. Realizada esta operación el proceso no tiene definido contexto a nivel de usuario por tanto cualquier error conducirá necesariamente a la terminación del proceso.

A continuación asigna (algoritmo `allocreg()`) y liga (algoritmo `attachreg()`) al proceso las nuevas regiones de código y datos. Asimismo, carga (algoritmo `loadreg()`) en estas regiones los contenidos del nuevo programa ejecutable. Recuérdese que la región de datos se divide en dos regiones: datos inicializados y datos no inicializados. Primero se rellena la región de datos inicializados y luego el núcleo (algoritmo `growreg()`) aumentará el tamaño de la región de datos para incluir la región de datos no inicializados. Posteriormente, asigna y liga una región de memoria para la pila del proceso. Entonces copia en ella los parámetros de entrada de la función de biblioteca asociada a la llamada al sistema `exec` que se habían salvado en un área de memoria del núcleo.

Concluida la configuración del nuevo contexto de usuario, el núcleo borra en el área U las direcciones de los manejadores de señales y establece las acciones por defecto. Ha de tenerse en cuenta que las funciones de los manejadores ya no existen en el nuevo programa. Las señales que eran ignoradas o bloqueadas antes de invocar a `exec` permanecerán ignoradas o bloqueadas.

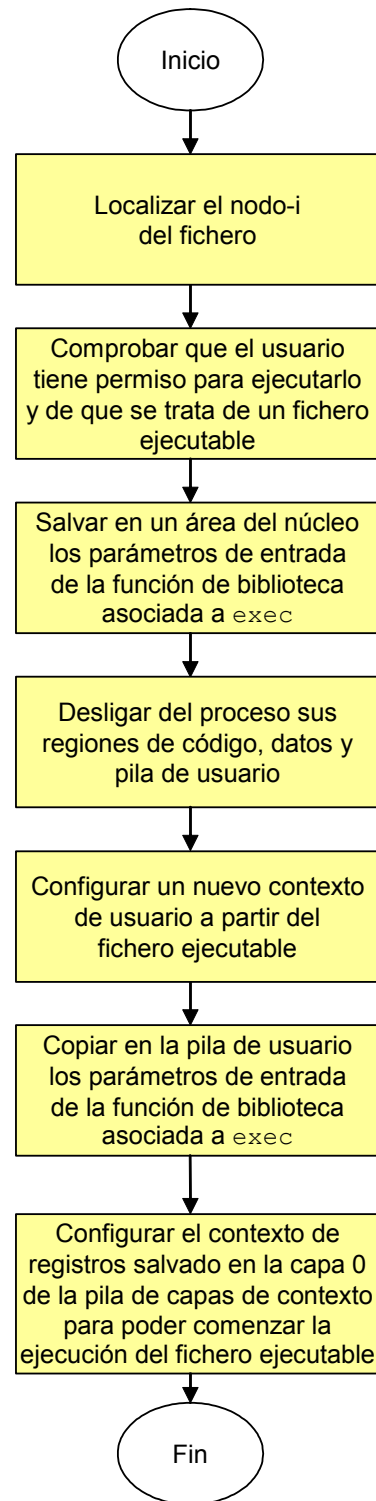


Figura 5.7: Principales acciones realizadas por el núcleo durante la ejecución del algoritmo `exec`

Finalmente, el núcleo modifica el contexto de registros salvado en la capa 0 de la pila de capas de contexto asociada al proceso invocador para que el nuevo programa pueda comenzar a ejecutarse cuando se retorne a modo usuario. Entre las acciones que realiza el núcleo se encuentra el cargar en el contador del programa salvado en la capa 0 la

dirección de inicio del nuevo programa, la cual se obtiene de la cabecera primaria del fichero ejecutable.

Es importante darse cuenta que cuando el proceso vuelva a modo usuario se ejecutará el código del nuevo programa, sin embargo seguirá siendo el mismo proceso, solo habrá cambiado su contexto a nivel de usuario.

En la Figura 5.7 se resumen las principales acciones que realiza el núcleo durante la ejecución del algoritmo `exec`.

♦ Ejemplo 5.9:

Considérese el siguiente programa escrito en C:

```
main()
{
[1]   if (fork()==0)
      {
[2]       execv("/bin/date",0);
      }
[3]   printf("\nFinalizar\n");
}
```

Programa 5.8

En este programa en primer lugar **[1]** se invoca a la llamada al sistema `fork` para crear un proceso hijo. Recuérdese que cuando finaliza la llamada `fork` devuelve un cero para el proceso hijo por lo que se cumple la condición del `if` y el hijo ejecuta en exclusiva, cuando sea planificado, la sentencia **[2]**, que es una llamada al sistema `exec` para ejecutar el fichero ejecutable `date`, que se encuentra en el directorio `bin` y que muestra en la pantalla la fecha y la hora. El contexto a nivel de usuario del proceso hijo es sustituido por los contenidos del fichero ejecutable. En consecuencia, después de la llamada a `exec` el proceso hijo no vuelve a ejecutar el programa antiguo, es decir, no imprimirá **[3]** en pantalla el mensaje `Finalizar`, sino que al ejecutar `date` aparecerá en pantalla la fecha y la hora actuales, y el proceso hijo finalizará.

Por otra parte cuando sea planificado el proceso padre ejecutará la sentencia **[3]**, es decir, se imprimirá en pantalla el mensaje `Finalizar`, y el proceso padre finalizará.

♦

TEMA 6

PLANIFICACION DE PROCESOS EN UNIX

6.1 INTRODUCCION

La CPU es un recurso compartido de la computadora por cuyo uso compiten los procesos. El sistema operativo debe decidir como reparte este recurso entre todos los procesos. El *planificador* es el componente del sistema operativo que determina que proceso debe ejecutarse en cada instante. UNIX es esencialmente un sistema de tiempo compartido, lo que significa que permite a varios procesos ejecutarse concurrentemente. En un sistema con un único procesador, la concurrencia no es más que una ilusión, puesto que en realidad solamente se puede estar ejecutando un único proceso en un instante de tiempo dado. El *planificador* cede el uso de la CPU a cada proceso durante un breve periodo de tiempo antes de planificar para ejecución a otro proceso. A este periodo se le denomina *cuanto*.

Naturalmente, conforme la carga del sistema va aumentando cada proceso recibirá un tiempo de CPU más pequeño, y por tanto se ejecutará más lentamente que si el sistema tuviese poca carga. El planificador debe asegurarse de que todos los procesos progresan en su ejecución.

En un sistema típico se ejecutarán distintas aplicaciones de forma concurrente. Estas aplicaciones pueden ser clasificadas de acuerdo a sus requerimientos y expectativas de planificación en las siguientes tipos:

- *Aplicaciones interactivas.* Se trata de aplicaciones que interaccionan constantemente con sus usuarios. Ejemplo de estas aplicaciones son los intérpretes de comandos, los editores, los programas con interfaces gráficos de usuario, etc. Estas aplicaciones se encuentran a la espera de una entrada del

usuario desde el terminal, bien mediante la pulsación de una tecla o mediante el movimiento del ratón. Cuando la entrada es recibida, ésta debe ser procesada rápidamente ya que en caso contrario el usuario encontrará que el sistema es insensible a sus acciones.

- *Aplicaciones batch.* Se trata de actividades planificadas por el usuario que típicamente se suelen realizar en segundo plano. Ejemplos de estas actividades son los compiladores, programas de cálculo científico, etc. Para este tipo de actividades, una medida de la eficiencia del planificador es la diferencia entre el tiempo que tarda en completarse estas tareas en presencia de otro tipo de actividades y el tiempo que tardan en completarse cuando son el único tipo de tareas presentes en el sistema.
- *Aplicaciones en tiempo real.* Se trata de actividades que son a menudo muy sensibles al tiempo de respuesta del sistema. Aunque existen muchos tipos de aplicaciones en tiempo real (control de sistemas físicos, adquisición de datos, procesamiento de video, etc), cada una con sus propios requerimientos, todas ellas comparten características comunes. En general este tipo de aplicaciones necesitan un comportamiento de planificación predecible con unos límites garantizados para el tiempo de respuesta.

En un sistema que presente un buen comportamiento, todas las aplicaciones independientemente de su tipo deben seguir progresando. Ninguna aplicación debería ser capaz de impedir que otras progresen, excepto si el usuario lo permite explícitamente. Además, el sistema debería siempre ser capaz de recibir y procesar entradas de lo usuarios interactivos, ya que en caso contrario el usuario no tendría forma de controlar el sistema.

Una descripción adecuada del planificador de cualquier sistema operativo, entre ellos UNIX, debe centrarse en dos aspectos: la *política de planificación* y la *implementación*.

La *política de planificación*, es el conjunto de reglas que utiliza el planificador para decidir que proceso debe ser planificado para ser ejecutado en un cierto instante y cuando debe planificar a otro proceso. La política de planificación elegida debe intentar cumplir, entre otros, los siguientes objetivos:

- Dar una respuesta rápida a las aplicaciones interactivas.
- Conseguir una productividad alta de los trabajos batch.

- Evitar el abandono de procesos, es decir, que los procesos pasen mucho tiempo sin recibir el uso de la CPU.
- Asegurar que las funciones del núcleo tales como, paginación, tratamiento de interrupciones, y administración de procesos pueden ser ejecutadas apropiadamente cuando se necesita.

Estos objetivos a menudo entran en conflicto, y el planificador debe buscar el mejor equilibrio entre todos ellos.

La *implementación* del planificador hace referencia a las estructuras de datos y los algoritmos utilizados para implementar la política de planificación. La implementación del planificador debe ser eficiente y producir una sobrecarga mínima en el sistema.

En este tema en primer lugar se describe el tratamiento de la interrupciones del reloj y las tareas basadas en consideraciones temporales, tales como los callouts y las alarmas. Asimismo se estudian algunas llamadas al sistema asociadas con el tiempo. En segundo lugar, se describe y analiza el planificador implementado en las distribuciones SVR3 y BSD4.3. Finalmente, habiéndose cubierto ya todo el material necesario para su adecuada comprensión, se describe de forma general como se realiza la sincronización de procesos en UNIX.

6.2 TRATAMIENTO DE LAS INTERRUPTCIONES DEL RELOJ

6.2.1 Consideraciones generales

Cada máquina UNIX tiene un reloj hardware que interrumpe al sistema a intervalos fijos de tiempo. Muchas máquinas requieren cuando se produce una interrupción del reloj que éste sea preparado, mediante instrucciones software, para que vuelva a interrumpir al procesador transcurrido el intervalo de tiempo adecuado. Estas instrucciones son fuertemente dependientes del hardware. Por el contrario, en otras máquinas el reloj se rearma de forma automática.

Al periodo de tiempo entre dos interrupciones del reloj se le denomina *tic de la CPU*, *tic del reloj*, o simplemente *tic*. La mayoría de las computadoras soportan una variedad de intervalos de tics. UNIX típicamente configura el tic de la CPU a 10 milisegundos¹.

¹ Éste no es un valor universal y depende de cada variante de UNIX. También depende de la resolución del reloj hardware del sistema.

Se denomina *frecuencia del reloj* al número de tics por segundo. Por ejemplo, para un tic de 10 milisegundos, la frecuencia del reloj sería 100.

El tratamiento de la interrupción del reloj depende fuertemente del sistema. La interrupción del reloj tiene un *npi* bastante elevado, solamente superado por las interrupciones asociadas a los errores de la máquina. Es por ello que la rutina de tratamiento de la interrupción del reloj se implementa para que realice lo más rápidamente las siguientes tareas:

- Rearmar el reloj hardware si fuera necesario.
- Adaptar las estadísticas de uso de la CPU para el proceso actual, es decir, usando la CPU.
- Enviar una señal SIGXCPU al proceso actual si éste ha excedido su cuota de uso de la CPU, es decir, su cuanto.
- Adaptar el reloj de la hora del día, y otros relojes relacionados.
- Comprobación de los callouts.
- Despertar a los procesos del sistema, como por ejemplo el intercambiador o el ladrón de páginas, cuando sea necesario.
- Comprobación de las alarmas.

6.2.2 Callouts

Los *callouts* son un mecanismo interno del núcleo que le permite invocar funciones transcurrido un cierto tiempo. Un *callout* típicamente almacena el nombre de la función que debe ser invocada, un argumento para dicha función y el tiempo en tics transcurrido el cual la función debe ser invocada.

Los usuarios no tiene ningún control sobre este mecanismo. Los callouts se pueden utilizar para la invocación de tareas periódicas tales como: la transmisión de paquetes de red, ciertas funciones de administración de memoria y del planificador, la monitorización de dispositivos para evitar la pérdida de interrupciones, etc.

Es importante resaltar que la rutina de tratamiento de la interrupción del reloj no invoca directamente a los callouts. En cada tic, la rutina comprueba si se debe realizar

algún callout. Si es así, activa un indicador para indicar que el manipulador de los callouts debe ser ejecutado. El sistema comprueba este indicador cuando retorna a su *npi* base, y si está activado, invoca al manipulador de los callouts, el cual invocará al callout que sea necesario. Por lo tanto, un callout se ejecutará tan pronto como sea posible, pero sólo cuando todas las interrupciones que estaban pendientes hayan sido atendidas.

El núcleo mantiene una *lista de callouts*. La organización de esta lista puede afectar el rendimiento del sistema, si existen varios callouts pendientes, ya que la lista es comprobada por la rutina de tratamiento de la interrupción del reloj a un *npi* elevado en cada tic de reloj. En consecuencia, la rutina debe intentar optimizar el tiempo de comprobación. Por el contrario, el tiempo requerido para insertar un nuevo callout dentro de la lista es menos crítico puesto que la inserción típicamente ocurre a un *npi* más bajo y con mucha menos frecuencia que una vez cada tic.

Existen varias formas de implementar la lista de callouts. Un método usado en BSD4.3 es ordenar la lista en función del tiempo que le resta al callout para ser invocado. A este tiempo comúnmente se le denomina *tiempo de disparo*. Cada entrada de la lista de callouts almacena la diferencia entre el tiempo de disparo de su callout asociado y el tiempo de disparo del callout asociado a la entrada anterior. El núcleo decrementa el tiempo de la primera entrada de la lista en cada tic de reloj y lanza el callout si el tiempo alcanza el valor 0.

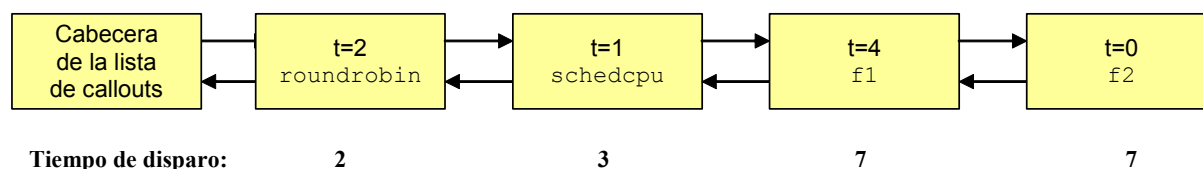
Otra posible aproximación sería utilizar también una lista ordenada, pero almacenar el tiempo absoluto de finalización para cada entrada. De esta forma, en cada tic, el núcleo compara el tiempo absoluto actual con el de la primera entrada y lanza el callout cuando los tiempos son iguales.

♦ Ejemplo 6.1:

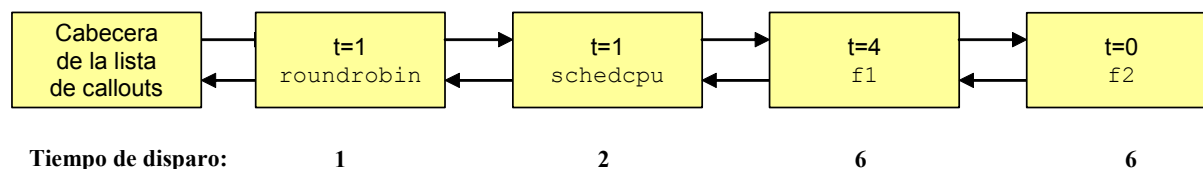
En la Figura 6.1(a) se muestra la lista de callouts en un cierto instante de tiempo. Se observa que dicha lista contiene cuatro entradas asociados a cuatro callouts que han sido ordenados en función de su tiempo de disparo, es decir, el tiempo que resta para que sean invocados.

La primera entrada está asociada al callout para la función `roundrobin` y en ella también se almacena su tiempo de disparo que es 2 tics.

La segunda entrada está asociada al callout para la función `schedcpu`. En su entrada de la lista se almacena el tiempo que resta para ser invocada con respecto a `roundrobin`, en este caso es 1 tic. Su tiempo de disparo es la suma de los tiempos almacenados en esta segunda entrada y en la primera entrada, es decir, $2+1=3$ tics.



(a) Lista de callouts en un cierto instante de tiempo



(b) Lista de callouts un tic más tarde

Figura 6.1: Implementación de la lista de callouts en el UNIX BSD

La tercera entrada está asociada al callout para la función `f1`. En su entrada de la lista se almacena el tiempo que resta para ser invocada con respecto a `schedcpu`, en este caso es 4 tics. Su tiempo de disparo es la suma de los tiempos almacenados en esta segunda entrada y en las dos entradas anteriores, es decir, $2+1+4=7$ tics.

La cuarta entrada está asociada al callout para la función `f2`. En su entrada de la lista se almacena el tiempo que resta para ser invocada con respecto a `f1`, en este caso es 0 tics. Su tiempo de disparo es la suma de los tiempos almacenados en esta segunda entrada y en las tres entradas anteriores, es decir, $2+1+4+0=7$ tics.

Por otra parte, en la Figura 6.1(b) se muestra la lista de callouts un tic más tarde. Se observa como se ha restado 1 tic a la primera entrada de la lista. En consecuencia el tiempo de disparo para los cuatro callouts se ha reducido también en 1 tic.

◆

6.2.3 Alarmas

Un proceso puede solicitar al núcleo que le envíe una señal una vez haya transcurrido un determinado tiempo. A este mecanismo de aviso se le denomina *alarma*. Existen tres tipos de *alarmas*:

- *Alarma de tiempo real*. Tiene asociado un contador o temporizador que se decrementa en tiempo real. Cuando el temporizador llega a 0 el núcleo envía al proceso una señal SIGALRM.

- *Alarma de tiempo virtual.* Tiene asociado un temporizador que se decrementa cuando el proceso se está ejecutando en modo usuario (tiempo virtual). Cuando el temporizador llega a 0 el núcleo envía al proceso una señal SIGVTALRM.
- *Alarma de perfil.* Tiene asociado un temporizador que se decrementa cuando el proceso se está ejecutando tanto en modo usuario como en modo supervisor. Cuando el temporizador llega a 0 el núcleo envía al proceso una señal SIGPROF.

Una elevada resolución de una alarma de tiempo real no implica una alta precisión. Supóngase que un usuario solicita que se le envíe una alarma de tiempo real después de 60 milisegundos. Cuando el tiempo expira, el núcleo envía la señal SIGALRM al proceso. Sin embargo, éste no se percatará de ella y tratará la señal hasta que no sea planificado de nuevo. Esto podría introducir un retardo substancial dependiendo de la prioridad de planificación del proceso receptor y de la carga del sistema. Los temporizadores de alta resolución son útiles cuando son usados para procesos de alta prioridad, que son menos susceptibles de sufrir retardos de planificación.

Por el contrario, la precisión de las alarmas de perfil y de tiempo virtual no sufren del problema descrito para las alarmas de tiempo real, puesto que no utilizan el tiempo real. Sin embargo, su precisión se ve afectada por el hecho de que la rutina de tratamiento de la interrupción del reloj carga todo el tic al proceso actual, incluso aunque éste solamente haya utilizado parte del mismo. De esta forma, el tiempo medido por estas alarmas refleja el número de interrupciones del reloj que han ocurrido mientras el proceso estaba ejecutándose. En general se puede afirmar que si configura un tiempo grande para el disparo de estas alarmas, entonces en promedio se compensa este efecto y el sistema mide bastante bien el tiempo utilizado por el proceso en su ejecución en modo usuario y/o en modo supervisor. En consecuencia la alarma se disparará con bastante precisión. Sin embargo, si el tiempo configurado para el disparo es pequeño, entonces estas alarmas sufren de una imprecisión bastante significativa.

Existen diversas llamadas al sistema que permiten a los usuarios la configuración de alarmas. Así, por ejemplo en el UNIX System V, la llamada al sistema `alarm` permite solicitar una alarma de tiempo real. Mientras que en el UNIX BSD, la llamada al sistema `setitimer` permite al proceso solicitar cualquier tipo de alarma y especificar el intervalo en microsegundos. Internamente, el núcleo convierte este intervalo al número apropiado de tics de CPU, que es la más alta resolución que el núcleo puede suministrar.

6.2.4 Llamadas al sistema asociadas con el tiempo

6.2.4.1 Fijación de la fecha del sistema

La llamada al sistema `stime` permite fijar la fecha y la hora actuales de nuestro sistema. Su sintaxis es:

```
resultado=stime(&valor);
```

donde `valor` es una variable entera que contiene los segundos transcurridos desde las 00:00:00 GMT² del día 1 de enero de 1970. Si la llamada al sistema se ejecuta correctamente `resultado` contendrá el valor 0, en caso contrario contendrá el valor -1.

La fecha del sistema sólo la puede fijar un usuario con los privilegios del superusuario. Por lo tanto, `stime` sólo podrá ser ejecutada por aquellos procesos cuyo `euid` coincida con el del superusuario.

6.2.4.2 Lectura de la fecha del sistema

La llamada al sistema `time` permite leer la fecha y la hora actuales que almacena el sistema. Su sintaxis es:

```
fecha=time(&valor);
```

donde `fecha` contendrá los segundos transcurridos desde las 00:00:00 GMT del día 1 de enero de 1970. El valor que devuelve `time`, también se copia en la variable entera `valor`.

Si la llamada al sistema falla entonces `fecha` tomará el valor `(time_t) (-1)`.

Los saltos de tiempo que se producen a intervalos regulares por necesidades de ajuste del calendario no quedan reflejados en la hora del sistema. Por ejemplo, en el intervalo que va desde 1970 a 1988 se ha producido una variación de 14 segundos, que no han quedado registrados en la hora del sistema

La resolución que ofrece `time` es de segundos, si se requiere realizar una medida más exacta, se puede usar la llamada al sistema `gettimeofday`.

² GMT es el acrónimo inglés de *Greenwich Mean Time*, es decir, el tiempo medido tomando como referencia el meridiano que pasa por la localidad inglesa de Greenwich.

♦ **Ejemplo 6.2:**

```
#include<signal.h>
void despertar(int sig);
main()
{
    long inicio,final;
    int resultado;
[1]  time(&inicio);
[2]  signal(SIGALRM, fun1);
[3]  alarm(10);
[4]  pause();
[5]  time(&final);
[6]  resultado=(final-inicio);
[7]  printf("\nTiempo final= %d (segundos transcurridos desde las
        00:00:00 GMT\n\t\t\t del 1 de enero de 1970)\n",final);
[8]  printf("\n Tiempo en responder= %d (seg)\n",resultado);
}

void fun1(int sig)
{
[9]  printf("\nMensaje 1\n");
};
```

Programa 6.1

Supóngase que el ejecutable que se crea al compilar el Programa 6.1 se llama *alarma*, y que este se invoca desde la línea de comandos (\$) de la siguiente forma:

```
$ alarma
```

Al ejecutarse el proceso asociado a este programa en primer lugar se invoca **[1]** a la llamada al sistema `time` que almacena en la variable `inicio` los segundos transcurridos desde las 00:00:00 GMT del 1 de enero de 1970. A continuación **[2]** se invoca a la llamada al sistema `signal` para asignar a la función `despertar` como manejador de las señales tipo `SIGALRM`. Después **[3]** se invoca a la llamada al sistema `alarm` para solicitar una alarma de tiempo real al cabo de 10 segundos. En **[4]** se invoca a la llamada al sistema `pause` que bloquea la ejecución del proceso hasta que reciba cualquier señal.

Al cabo de 10 segundo se dispara la alarma, el proceso recibe la señal `SIGALRM` y cuando vuelve a modo usuario ejecuta la función `fun1` que es el manejador asociado a este tipo de señales, con lo que se imprime **[9]** en pantalla el mensaje

Mensaje 1

A continuación [5] se realiza la misma acción que en [1] pero usando la variable `final`. Luego [6] se almacena en la variable `resultado` la diferencia entre el contenido de `final` menos el de `inicio`. Finalmente se muestran en pantalla los contenidos de las variables `final` [7] y `resultado` [8] dentro de los siguientes mensajes

```
Tiempo final= 1131386043 (segundos transcurridos desde las 00:00:00 GMT
                        del 1 de enero de 1970)
Tiempo en responder= 10 (seg);
```

y el proceso finaliza.



6.2.4.3 Tiempos de ejecución asociados a un proceso

La llamada al sistema `times` permite conocer el tiempo empleado por un proceso en su ejecución. Su sintaxis es:

```
resultado=times(&tbuffer);
```

La llamada al sistema `times` rellena la estructura `tbuffer` del tipo predefinido `tms` con la información estadística relativa a los tiempos de ejecución empleados por el proceso, desde su inicio hasta el momento de invocar a `times`. El tipo `tms` se define de la siguiente forma:

```
struct tms
{
    clock_t    tms_utime
    clock_t    tms_stime
    clock_t    tms_cutime
    clock_t    tms_cstime
}
```

El tipo `clock_t` se define para contabilizar los tics de reloj. Cada segundo se compone de un total de `CLK_TCK` tics donde `CLK_TCK` es una constante definida en el fichero de cabecera `time.h`. Para calcular el tiempo en segundos que almacena una variable del tipo `clock_t`, hay que dividirla por `CLK_TCK`.

Por lo tanto el significado de los campos de la estructura `tbuffer` es el siguiente:

- `tms_utime`. Es el tiempo de uso de la CPU (en tics) del proceso ejecutándose en modo usuario.

- `tms_stime`. Es el tiempo de uso de la CPU (en tics) del proceso ejecutándose en modo núcleo.
- `tms_cutime`. Es la suma de los campos `tms_utime` y `tms_cutime` para los procesos hijos. Es decir, el tiempo de uso de la CPU (en tics) de los procesos hijos, los hijos de los hijos, etc. ejecutándose en modo usuario.
- `tms_cstime`. Es la suma de los campos `tms_stime` y `tms_cstime` para los procesos hijos. Es decir, el tiempo de uso de la CPU (en tics) de los procesos hijos, los hijos de los hijos, etc. ejecutándose en modo núcleo.

Los valores que aparecen en los campos de la estructura apuntada por `tbuffer` se refieren al proceso que invoca a `times` y a los procesos hijos para los cuales el proceso padre ha ejecutado una llamada al sistema `wait`.

En el cómputo de todos los tiempos no se tiene en cuenta el tiempo dedicado por los procesos del sistema a los procesos del usuario (por ejemplo, el tiempo que emplea el sistema en hacer que un proceso de usuario cambie de contexto). Los tiempos reales son tiempos reales de CPU, por lo que no se contabilizan los periodos en los que el proceso se encontraba en el estado dormido.

Si `times` se ejecuta satisfactoriamente, entonces en `resultado` se almacenará el tiempo real transcurrido (en tics) a partir de un instante pasado arbitrario. Este instante puede ser el momento de arranque del sistema y no cambia de una llamada a otra. Si `times` falla entonces `resultado` contendrá el valor -1.

♦ Ejemplo 6.3:

```
#include <time.h>
#include <sys/times.h>
main()
{
    struct tms pb1,pb2;
    clock_t t1,t2;
    long h=0,k=0,cont=0;
    [1] t1=times(&pb1);
    [2] for(h==1;h<=10000000;h++)
        {
            [3] fd=open("datos.txt",0600);
```

```
[4]         close(fd);

           };

[5]     t2=times(&pb2);
[6]     printf("\n Tiempo real= %g segundos\n", (t2-t1)/CLK_TCK);
[7]     printf("\n Tiempo de uso de la CPU en modo usuario= %g
           segundos\n", (pb2.tms_utime-pb1.tms_utime)/CLK_TCK);
}
```

Programa 6.2

Supóngase que el ejecutable que se crea al compilar el Programa 6.2 se llama `tiempos`, y que este se invoca desde la línea de comandos (\$) de la siguiente forma:

```
$ tiempos
```

Al ejecutarse el proceso asociado a este programa en primer lugar se invoca **[1]** a la llamada al sistema `times` que rellena la estructura `pb1` del tipo `tms` con la información estadística relativa a los tiempos de ejecución empleados por el proceso, desde su inicio hasta el momento de invocar a `times`. Asimismo en la variable `t1` se almacena el tiempo real transcurrido en tics desde que se inicio el sistema.

A continuación **[2]** se ejecuta un bucle `for` 10000000 veces dentro del cual simplemente se invoca a la llamada al sistema `open` para abrir **[3]** el fichero `datos.txt` con permisos de lectura y escritura para todos los usuarios para a continuación cerrarlo mediante el uso de la llamada al sistema `close` **[4]**.

Una vez finalizado el bucle se vuelve a invocar **[5]** a la llamada al sistema `times`. Ahora la estructura que se rellena con información estadística es `pb2`, mientras que el tiempo real transcurrido en tics desde que se inicio el sistema se almacena en la variable `t2`.

A continuación **[6]**, se muestra en pantalla el mensaje

```
Tiempo real= 20.75 segundos
```

Por último **[7]**, se muestra en pantalla el mensaje

```
Tiempo de uso de la CPU en modo usuario= 20.75 segundos
```

y el programa finaliza.



6.3 PLANIFICACIÓN TRADICIONAL EN UNIX

En esta sección se va a describir el diseño y la implementación del planificador utilizado en BSD4.3³. La política de planificación que utiliza este planificador es del tipo *round robin con colas multinivel*. Cada proceso tiene asignada una *prioridad de planificación*⁴ que cambia con el tiempo. Dicha prioridad le hace pertenecer a una de las múltiples colas de prioridad que maneja el planificador.

El planificador siempre selecciona al proceso que encontrándose en el estado *preparado en memoria principal para ser ejecutado* o en el estado *expropiado* tiene la mayor prioridad. En el caso de los procesos de igual prioridad (se encuentran en la misma cola) lo que hace es ceder el uso de la CPU a uno de ellos durante un cuanto, cuando finaliza dicho cuanto le expropia la CPU y se lo cede a otro proceso. El planificador varía dinámicamente la prioridad de los procesos basándose en su tiempo de uso de la CPU. Si un proceso de mayor prioridad alcanza el estado *preparado en memoria principal para ser ejecutado*, el planificador expropia el uso de la CPU al proceso actual incluso aunque éste no haya completado su cuanto.

El núcleo tradicional de UNIX es estrictamente no expropiable. Es decir, si el proceso actual se encuentra en modo núcleo (debido a una llamada al sistema o a una interrupción), no puede ser forzado a ceder la CPU a un proceso de mayor prioridad. Dicho proceso cederá voluntariamente la CPU cuando entre en el estado dormido. En caso contrario sólo se le podrá expropiar la CPU cuando retorne a modo usuario.

6.3.1 Prioridades de planificación de un proceso

La *prioridad de planificación* de un proceso es un valor entre 0 y 127. Numéricamente los valores más bajos corresponden a las prioridades más altas. Las prioridades entre 0 y 49 están reservadas para el núcleo, mientras que los procesos en modo usuario tienen las prioridades entre 50 y 127.

La entrada asociada a un proceso en la tabla de procesos posee los siguientes campos que contienen información relacionada con la prioridad de planificación:

- `p_pri`. Contiene la prioridad de planificación actual.

³ El planificador en SVR3 es prácticamente idéntico, difiere únicamente en algunos aspectos menores, tales como el nombre de algunas funciones y variables.

⁴ Por comodidad, en lo que resta de sección la palabra *prioridad* hará referencia a *prioridad de planificación*, salvo que se indique lo contrario.

- `p_usrpri`. Contiene la prioridad de planificación actual en modo usuario.
- `p_cpu`. Contiene el tiempo transcurrido desde que el proceso utilizó por última vez la CPU, también denominado *uso reciente de la CPU*.
- `p_nice`. Contiene el *factor de amabilidad*, que es controlable por el usuario.

Los campos `p_pri` y `p_usrpri` se utilizan de modo diferente. El planificador consulta `p_pri` para decidir que proceso debe planificar. Cuando un proceso se encuentra en modo usuario, su valor `p_pri` es idéntico a `p_usrpri`. Cuando el proceso despierta después de haber entrado en el estado dormido durante una llamada al sistema, su prioridad es temporalmente aumentada para dar preferencia al procesamiento en modo núcleo. Por este motivo el planificador utiliza `p_usrpri` para salvar la prioridad que debe ser asignada al proceso cuando éste retorne al modo usuario, y `p_pri` para almacenar su prioridad en modo núcleo.

El núcleo asocia una *prioridad de dormir* en función del evento por el que el proceso entró en el estado dormido. Ésta es una prioridad en modo núcleo, y por tanto su valor está comprendido entre 0 y 49. Por ejemplo (ver Figura 4.2), la prioridad de dormir para un proceso esperando por la entrada en un terminal es 28, mientras que para un proceso esperando por una operación de E/S con el disco es 20. Cuando un proceso despierta, el núcleo configura su valor `p_pri` a la prioridad de dormir del evento o recurso. Puesto que las prioridades en modo núcleo son más altas que las prioridades en modo usuario, estos procesos son planificados antes que aquellos que ejecutan código de usuario. Esto permite que las llamadas al sistema se puedan completar apropiadamente, lo que es deseable puesto que los procesos pueden tener bloqueado algún recurso clave del núcleo mientras ejecutan la llamada al sistema.

Cuando un proceso completa la llamada al sistema y va a retornar a modo usuario, su prioridad de planificación es configurada a su prioridad en modo usuario actual, es decir, al valor que se encontraba almacenado en `p_usrpri`. Si esta prioridad es más baja que la de otros procesos planificables, entonces el núcleo realizará un cambio de contexto.

La prioridad de ejecución en modo usuario depende de dos factores: el *factor de amabilidad* (`p_nice`) y el *uso reciente de la CPU* (`p_cpu`).

El *factor de amabilidad* es un número entero entre 0 y 39. Su valor por defecto es 20. Se denomina factor de amabilidad, porque un usuario incrementando este valor está

disminuyendo la prioridad de sus procesos y en consecuencia le está cediendo el turno de uso de CPU a los procesos de otros usuarios. A los procesos en segundo plano el núcleo les asigna de forma automática un factor de amabilidad elevado.

El factor de amabilidad de un proceso también puede ser disminuido y en consecuencia se estaría aumentando su prioridad. Esta acción solamente la puede realizar el superusuario.

La llamada al sistema `nice` permite aumentar o disminuir el factor de amabilidad actual del proceso que la invoca. Por lo tanto un proceso no puede modificar el factor de amabilidad de otro proceso. Su sintaxis es:

```
resultado=nice(incremento);
```

donde `incremento` es una variable entera que puede tomar valores entre -20 y 19. El valor de incremento será sumado al valor del factor de amabilidad actual. Sólo el superusuario puede invocar a `nice` con valores de `incremento` negativos. Si se produce un error durante la ejecución de `nice`, entonces `resultado` contendrá el valor -1.

También es posible modificar el factor de amabilidad de un proceso desde la línea de comandos mediante el uso del comando `nice`.

Los sistemas de tiempo compartido intentan asignar el procesador de tal forma que las aplicaciones en competición reciban aproximadamente la misma cantidad de tiempo de CPU. Esto requiere monitorizar el uso de la CPU de los diferentes procesos y utilizar esa información en las decisiones de planificación. El campo `p_cpu` almacena una medida del uso reciente de la CPU por parte del proceso. Este campo se inicializa a 0 cuando el proceso es creado. En cada tic, la rutina de tratamiento de la interrupción del reloj incrementa `p_cpu` para el proceso actualmente en ejecución, hasta un máximo de 127. Además, cada segundo, el núcleo invoca a una rutina denominada `schedcpu` que reduce el valor de `p_cpu` de un proceso mediante un *factor de disminución* (*decay*). SVR3 utiliza un factor de disminución fijo de 1/2, mientras que BSD4.3 utiliza la siguiente fórmula:

$$\text{decay} = (2 * \text{load_average}) / (2 * \text{load_average} + 1); \quad (1)$$

donde `load_average` es el número medio de procesos preparados para ejecución en el último segundo. Luego al cabo de un segundo el nuevo valor de `p_cpu` vendrá dado por la fórmula:

$$p_cpu = decay * p_cpu; \quad (2)$$

La rutina `schedcpu` también recalcula las prioridades de usuario de todos los procesos usando la fórmula:

$$p_usrpri = PUSER + (p_cpu / 4) + (2 * p_nice); \quad (3)$$

donde `PUSER` es la *prioridad de usuario base*, que vale 50. Este valor es la prioridad más alta que puede tomar un proceso ejecutándose en modo usuario, y es justamente el límite con respecto a las prioridad en modo núcleo.

Como resultado, si un proceso ha acumulado recientemente un gran cantidad de tiempo de CPU, su factor `p_cpu` aumentará. Ello producirá un mayor valor de `p_usrpri`, y por tanto una prioridad de ejecución más baja. Cuanto más tiempo está esperando un proceso en ser planificado, más disminuirá el *factor de disminución* su `p_cpu`, y en consecuencia su prioridad irá aumentando.

Este esquema evita que los procesos de baja prioridad nunca lleguen a ser ejecutados. También favorece a los procesos limitados por E/S (procesos que requieren muchas operaciones de E/S, por ejemplo, las consolas de comandos y los editores de texto) en contraposición a los procesos limitados por la CPU (procesos que requieren mucho uso de la CPU, por ejemplo, los compiladores). Un proceso limitado por E/S, mantiene una alta prioridad ya que su `p_cpu` es pequeño, y recibe tiempo de CPU rápidamente cuando la necesita. Por contra, los procesos limitados por la CPU tienen valores de `p_cpu` altos y por tanto una baja prioridad.

El *factor de uso de la CPU* suministra justicia y paridad en la planificación de los procesos de tiempo compartido. La idea básica es mantener la prioridad de todos estos procesos en un rango aproximadamente igual durante un periodo de tiempo. Los procesos subirán o bajarán dentro de un cierto rango dependiendo de cuanto tiempo de CPU hayan consumido recientemente. Si las prioridades cambian demasiado lentamente, los procesos que comenzaran con una prioridad más baja, permanecerían así durante largos periodos de tiempo, por lo que su ejecución se demorará demasiado.

El efecto del *factor de disminución* es suministrar un promedio ponderado exponencialmente de uso de la CPU a los procesos durante todo su tiempo de vida. La formula usada en el SVR3 conduce a un promedio exponencial simple, que tiene como efecto indeseable la elevación de las prioridades cuando la carga del sistema aumenta. Esto es así porque en un sistema con mucha carga cada proceso recibe poco uso de la

CPU y en consecuencia su valor de uso de la CPU se mantiene bajo, y el factor de disminución lo reduce aún más. Como resultado, el uso de la CPU no tiene mucho impacto en la prioridad, y los procesos que comienzan con una prioridad más baja se quedan sin usar la CPU durante un tiempo desproporcionado.

La aproximación BSD4.3 fuerza al *factor de disminución* a depender de la carga del sistema. Cuando la carga es elevada el factor de disminución es pequeño. Consecuentemente, procesos que reciben ciclos de CPU verán rápidamente disminuida su prioridad.

6.3.2 Implementación del planificador

El planificador mantiene un array denominado `qs` de 32 colas de ejecución (ver Figura 6.2). Cada cola se corresponde a cuatro prioridades adyacentes. Así, la cola 0 es utilizada por las prioridades 0-3, la cola 1 por las prioridades 4-7, etc. Cada cola contiene la cabecera de la lista doblemente enlazada de entradas de la tabla de procesos. La variable global `whichqs` contiene un mapa de bits con un bit asociado a cada cola. El bit está activado si hay al menos un proceso en la cola. Solamente procesos planificables son mantenidos en estas colas del planificador. Esto simplifica la tarea de selección de un proceso para ser ejecutado. El algoritmo del núcleo que implementa el cambio de contexto (`swtch` en BSD4.3), examina `whichqs` para encontrar el índice del primer bit activado. Este índice identifica la cola del planificador que contiene al proceso ejecutable de más elevada prioridad. El algoritmo `swtch` borra al proceso de la cabeza de la cola, y realiza el cambio de contexto.

♦ Ejemplo 6.4:

En la Figura 6.2 se muestra el array `qs` y la variable global `whichqs`. Se observa que la cola 3 (se comienza a contar desde 0) contiene 3 procesos cuyas prioridades se encuentran en el rango 12-15. En consecuencia, en la variable `whichqs` el cuarto bit desde la izquierda, que es el asociado a la cola 3, se encuentra activado.

Asimismo se observa que la cola 5 contiene 2 procesos cuyas prioridades se encuentran en el rango 20-23. Por lo tanto, en la variable `whichqs` el sexto bit desde la izquierda, que es el asociado a la cola 5, se encuentra activado.

Cuando el algoritmo del núcleo que implementa el cambio de contexto (`swtch`), examine `whichqs` comenzando por la izquierda el primer bit que encontrará activado será el asociado a la cola 3. El proceso que será planificado será el que se encuentre en la cabeza de la cola. Así que el algoritmo `swtch` borra al proceso de la cabeza de la cola, y realiza el cambio de contexto.

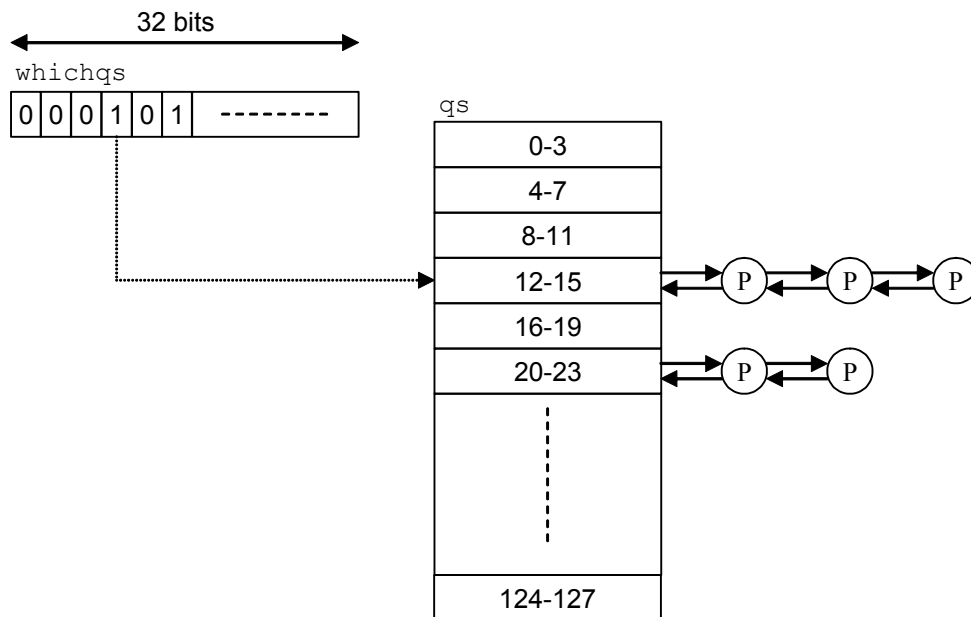


Figura 6.2: Estructuras que usa el planificador en el UNIX BSD4.3

Puesto que tanto BSD4.3 y SVR2 y SVR3 tenían a la arquitectura VAX-11 como máquina objetivo, la implementación del planificador está fuertemente influenciado por esta arquitectura.

6.3.3 Manipulación de las colas de ejecución

El planificador sigue las siguientes reglas para manipular las colas de ejecución:

- El proceso de más alta prioridad siempre se ejecuta, excepto si el proceso actual se está ejecutando en modo núcleo.
- Un proceso tiene asignado un tiempo de ejecución fijo denominado *cuanto* (100 ms en 4.3BSD). Esto solamente afecta a la planificación de los procesos pertenecientes a la misma cola. Cada 100 milisegundos, el núcleo invoca (usando un callout) una rutina denominada `roundrobin` para planificar al siguiente proceso de la misma cola.
- Si un proceso de más alta prioridad fuese puesto en el estado listo para ejecución, éste sería planificado de forma preferente sin esperar por `roundrobin`.

- Si los procesos en el estado preparado en memoria para ser ejecutado o en el estado expropiado pertenecen a una cola de prioridad más baja que el proceso actual, éste continuará ejecutándose incluso aunque su cuanto haya expirado.

La rutina `schedcpu` recalcula la prioridad de cada proceso una vez por segundo. Puesto que la prioridad no puede cambiar mientras el proceso está en la cola de planificados, `schedcpu` borra al proceso de la cola, cambia su prioridad, y lo vuelve a colocar, quizás en una cola de prioridad distinta. La rutina de tratamiento de la interrupción del reloj recalcula la prioridad del proceso actual cada cuatro tics.

El núcleo configura un indicador denominado `runrun`, que indica que un proceso (B) de mayor prioridad que el actual (A) está esperando para ser planificado. Cuando el proceso A retorne a modo usuario, el núcleo comprueba el indicador `runrun`, si está activado, transfiere el control a la rutina de cambio de contexto, para iniciar un cambio de contexto y planificar a B.

♦ Ejemplo 6.5:

Supóngase que en un sistema UNIX el tic de reloj es de 10 ms y que el cuanto es de 100 ms. En consecuencia en un cuanto se producirán 10 tics.

Supóngase también que tres procesos A, B y C han sido creados de forma simultánea con una prioridad inicial `p_usrpri=90`. El factor de amabilidad para todos ellos es `p_nice=20`. La prioridad de usuario base es `PUSER=50`. El tiempo de uso de la CPU (en tics) es `p_cpu=0` para los tres procesos. Se va a utilizar la siguiente notación `p_usrpri(X)` y `p_cpu(X)` denotan la prioridad de usuario y el tiempo de uso de la CPU, respectivamente, para el proceso `X`.

Supóngase además que los procesos durante su ejecución no invocan a ninguna llamada al sistema, y que no existe ningún otro proceso en el sistema en el estado *preparado para ejecución*.

En el modelo de planificador descrito la rutina de tratamiento de la interrupción de reloj (se ejecuta cada tic) recalcula usando la ecuación (3) la prioridad del proceso actual cada 4 tics (es decir, 40 ms). Al finalizar un cuanto se dispara a la rutina `roundrobin` que planifica al siguiente proceso de la misma cola de ejecución. Cada segundo se dispara a la rutina `schedcpu` que reduce el tiempo de uso de la CPU `p_cpu` de todos los procesos planificables mediante un factor de disminución `decay=1/2` usando la ecuación (2) y recalcula la prioridad de usuario de todos los procesos planificables usando la ecuación (3).

Por simplificar la descripción se va a suponer que la rutina del núcleo asociada al tratamiento de la interrupción de reloj, la rutina `roundrobin` y la rutina `schedcpu` se ejecutan de manera prácticamente instantánea.

En el rango de tiempo entre 0 y 100 ms se ejecuta el proceso A. Durante este tiempo la rutina de tratamiento de la interrupción de reloj recalcula la prioridad del proceso actual dos veces en 40 ms y 80 ms usando la ecuación (3). En 40 ms $p_{cpu}(A)=4$ tics, luego

$$p_{usrpri}(A)=PUSER+(p_{cpu}(A)/4)+(2*p_{nice}(A))= 50+(4/4)+(2*20)=91$$

En 80 ms $p_{cpu}(A)=8$ tics, luego

$$p_{usrpri}(A)= 50+(8/4)+(2*20)=92$$

Al finalizar el cuanto en 100 ms se dispara *roundrobin* que planifica al proceso B. Este se ejecuta en el rango entre 100 y 200 ms. La rutina de tratamiento de la interrupción de reloj recalcula la prioridad del proceso actual dos veces en 140 y 180 ms. En 140 ms $p_{cpu}(B)=4$ tics, luego

$$p_{usrpri}(B)= 50+(4/4)+(2*20)=91$$

En 180 ms $p_{cpu}=8$ tics, luego

$$p_{usrpri}(B)= = 50+(8/4)+(2*20)=92$$

Al finalizar el cuanto en 200 ms se dispara *roundrobin* que planifica al proceso C. Este se ejecuta en el rango entre 200 y 300 ms. La rutina de tratamiento de la interrupción de reloj recalcula la prioridad del proceso actual dos veces en 240 y 280 ms. En 240 ms $p_{cpu}(C)=4$ tics, luego

$$p_{usrpri}(C)= 50+(4/4)+(2*20)=91$$

En 280 ms $p_{cpu}=8$ tics, luego

$$p_{usrpri}(C)= 50+(8/4)+(2*20)=92$$

Este esquema de funcionamiento se iría repitiendo hasta llegar a 1s. Así se tiene que:

- En el rango [300, 400] ms se ejecuta el proceso A. Su tiempo de CPU al finalizar el cuanto es $p_{cpu}(A)=16$ y su prioridad de usuario es $p_{usrpri}(A)= 94$.
- En el rango [400, 500] ms se ejecuta el proceso B. Su tiempo de CPU al finalizar el cuanto es $p_{cpu}(B)=16$ y su prioridad de usuario es $p_{usrpri}(B)= 94$.
- En el rango [500, 600] ms se ejecuta el proceso C. Su tiempo de CPU al finalizar el cuanto es $p_{cpu}(C)=16$ y su prioridad de usuario es $p_{usrpri}(C)= 94$.
- En el rango [600, 700] ms se ejecuta el proceso A. Su tiempo de CPU al finalizar el cuanto es $p_{cpu}(A)=24$ y su prioridad de usuario es $p_{usrpri}(A)= 96$.

- En el rango [700, 800] ms se ejecuta el proceso B. Su tiempo de CPU al finalizar el cuanto es $p_cpu(B)=24$ y su prioridad de usuario es $p_usrpri(B)=96$.
- En el rango [800, 900] ms se ejecuta el proceso C. Su tiempo de CPU al finalizar el cuanto es $p_cpu(C)=24$ y su prioridad de usuario es $p_usrpri(C)=96$.
- En el rango [900, 1000] ms se ejecuta el proceso A. Su tiempo de CPU al finalizar el cuanto es $p_cpu(A)=32$ y su prioridad de usuario es $p_usrpri(A)=98$.

Al cabo de 1 s se dispara la rutina `schedcpu` que disminuye el tiempo de uso de CPU de todos los procesos usando la fórmula (2)

$$p_cpu(A) = decay * p_cpu(A) = (1/2) * 32 = 16$$

$$p_cpu(B) = decay * p_cpu(B) = (1/2) * 24 = 12$$

$$p_cpu(C) = decay * p_cpu(C) = (1/2) * 24 = 12$$

Asimismo la rutina `schedcpu` recalcula la prioridad en modo usuario de todos los procesos usando la fórmula (3)

$$p_usrpri(A) = 50 + (16/4) + (2 * 20) = 94$$

$$p_usrpri(B) = 50 + (12/4) + (2 * 20) = 93$$

$$p_usrpri(C) = 50 + (12/4) + (2 * 20) = 93$$

Además quita a los tres procesos de la cola de ejecución en la que habían sido colocados al ser creado, la asociada al rango 88-91, y los coloca en la cola de ejecución asociada al rango de prioridades 92-95.

Si no existe otro proceso más prioritario el próximo proceso en ser planificado será el proceso B.

♦

6.3.4 Análisis

El algoritmo de planificación tradicional es simple pero efectivo. Es adecuado para un sistema de tiempo compartido con una mezcla de trabajos interactivos y batch. El cálculo dinámico de las prioridades previene el abandono de cualquier proceso. Esta implementación favorece a los trabajos limitados por E/S que requieren de forma poco frecuente ciclos de CPU.

El planificador tiene varias limitaciones que lo hacen poco adecuado para su uso en una amplia variedad de aplicaciones comerciales:

- No está bien escalado, si el número de procesos es muy grande resulta poco eficiente para calcular todas las prioridades cada segundo.
- No hay forma de garantizar un determinado tiempo de uso de la CPU a un proceso o a un grupo de procesos en concreto.
- Las aplicaciones tienen poco control sobre sus prioridades. El mecanismo del factor de amabilidad es demasiado simple y resulta inadecuado.
- Puesto que el núcleo no es expropiable, los procesos de mayor prioridad quizás tengan que esperar una cantidad de tiempo significativa incluso después de estar en el estado preparado para ejecución. A este fenómeno se le denomina *inversión de prioridades*.

Los sistemas UNIX modernos son utilizados en una amplia gama de entornos. En particular, hay una fuerte necesidad para que el planificador soporte aplicaciones en tiempo real que requieren un comportamiento más predecible y tiempos de respuesta limitados. Por ello los sistemas UNIX modernos (SVR4, Solaris 2.x, etc) tuvieron que rediseñar por completo el planificador.

6.4 SINCRONIZACIÓN

Varios procesos pueden estar ejecutándose en el núcleo de UNIX al mismo tiempo, quizás incluso se encuentren ejecutando la misma rutina. En un sistema con un único procesador solamente un proceso puede estar ejecutándose en la CPU. Sin embargo el sistema rápidamente conmuta de un proceso a otro, generando la ilusión de que todos ellos se ejecutan concurrentemente. Esta característica se suele denominar *multiprogramación*. Puesto que estos procesos comparten el núcleo, éste debe sincronizar el acceso a sus estructuras de datos para evitar su corrupción.

La primera medida de seguridad que utiliza UNIX es que su núcleo no es expropiable. Es decir, cualquier proceso ejecutándose en modo núcleo continuará ejecutándose, incluso aunque su cuanto haya expirado, hasta que vuelva a modo usuario o entre en el estado dormido en espera de algún recurso que se encuentra ocupado. Esto permite al código del núcleo manipular las estructuras de datos sin necesidad de bloquearlas, sabiendo que ningún otro proceso podrá acceder a ellas hasta que el proceso actual haya terminado de utilizarlas y esté listo para ceder el núcleo en un estado consistente.

La no expropiación del núcleo es una herramienta de sincronización bastante útil para un amplio rango de situaciones. Sin embargo, aunque el proceso actualmente ejecutándose en modo núcleo no pueda ser expropiado si que puede ser interrumpido. Las interrupciones son una parte fundamental de la actividad del sistema y normalmente requieren ser atendidas urgentemente. El manipulador de las interrupciones puede manipular las mismas estructuras de datos con las que el proceso actual estaba trabajando, lo que puede producir una corrupción de los datos. Por lo tanto el núcleo debe sincronizar el acceso a los datos que son utilizados tanto por el código normal del núcleo como por el manipulador de interrupciones. UNIX resuelve este problema suministrando un mecanismo para bloquear (enmascarar) las interrupciones.

Por otra parte, a menudo un proceso desea garantizarse el uso exclusivo de un determinado recurso incluso aunque entre en el estado dormido. Por ejemplo, un proceso puede desear leer un bloque de datos del disco duro desde de un buffer. Para ello en primer lugar se asignará un buffer para almacenar el bloque y después iniciará la operación de E/S con el disco. El proceso deberá esperar hasta que la operación de E/S se complete, lo que significa que mientras tanto deberá ceder el uso de la CPU para que sea ejecutado otro proceso. Si dicho proceso requiere usar el mismo buffer y lo utiliza para algún propósito diferente, el contenido del buffer puede quedar indeterminado o corrupto. Por lo tanto, es necesario disponer de alguna forma de bloquear el recurso mientras un proceso se encuentra en el estado dormido.

UNIX asocia dos indicadores, *bloqueado* y *deseado*, a cada recurso compartido. Cuando un proceso desea acceder a un recurso compartido, como un buffer, primero el núcleo comprueba el indicador *bloqueado*. Si no está activado, lo activa y procede a usar el recurso. Si un segundo proceso intentara acceder al mismo recurso, se encontraría con el indicador *bloqueado* activado y debería entrar en el estado dormido hasta que el recurso quedase disponible. Antes de colocar a dicho proceso en el estado dormido el núcleo activa el indicador *deseado*.

Cuando el primer proceso ha terminado de usar el recurso, el núcleo desactiva el indicador *bloqueado* y comprueba el indicador *deseado*. Si se encuentra activado, eso significará que al menos un proceso se encuentra esperando para usarlo. En ese caso, examina la lista de procesos dormidos y despierta a estos procesos. Cuando uno de ellos sea planificado para ser ejecutado, el núcleo comprobará de nuevo el indicador de *bloqueado* y encontrará que está desactivado, entonces lo activará y procederá a usar el recurso.

Este modelo de sincronización basado en indicadores funciona adecuadamente para máquinas con un solo procesador pero presentan serios problemas de comportamiento en sistemas multiprocesador.

TEMA 7

COMUNICACION ENTRE PROCESOS EN UNIX

7.1 INTRODUCCION

Un entorno de programación complejo utilizada frecuentemente múltiples procesos de forma cooperativa para la realización de diferentes operaciones. Estos procesos deben comunicarse entre si y compartir recursos e información. El núcleo debe disponer de los mecanismos necesarios para implementar esta comunicación. Estos mecanismos son colectivamente denominados como *mecanismos de comunicación entre procesos* o *mecanismos IPC*¹.

Las interacciones entre los procesos persiguen distintos objetivos:

- *Transferencia de datos.* Un proceso puede necesitar transferir datos a otro proceso. La cantidad de datos puede variar desde un byte hasta varios megabytes.
- *Compartir datos.* Múltiples procesos pueden necesitar operar sobre datos compartidos, de tal forma que si un proceso modifica estos datos, los cambios realizados deben ser visibles para el resto de procesos que comparten dichos datos.
- *Notificación de eventos.* Un proceso puede notificar a otro proceso o a un conjunto de procesos que se ha producido algún evento. Por ejemplo, cuando un proceso termina, puede necesitar informar de este hecho a su proceso padre. El receptor puede ser notificado asíncronamente, en cuyo caso su

¹ IPC es el acrónimo que se deriva del término inglés *interprocess communication*

procesamiento normal se verá interrumpido. Alternativamente, el receptor quizás desee esperar por la notificación del evento.

- *Compartir recursos.* Aunque el núcleo suministra sus propias semánticas para la asignación de recursos, éstas pueden no ser adecuadas para todas las aplicaciones. Un conjunto de procesos puede desear definir su propio protocolo de acceso a ciertos recursos. Tales reglas se implementan normalmente mediante un esquema de sincronización y bloqueo.
- *Control de procesos.* Un proceso (controlador), por ejemplo un depurador, necesita disponer de un control total sobre la ejecución de otro proceso (objetivo). El proceso controlador puede interceptar todas las interrupciones software y excepciones generadas por el proceso objetivo.

Este tema se dedica al estudio de los mecanismos IPC en UNIX. En la primera parte del tema se describen un conjunto de mecanismos universales disponibles en todas las versiones de UNIX, como son las señales, las tuberías y el seguimiento de procesos. Por otra parte, la segunda parte del tema se dedica a describir los mecanismos colectivamente denominados como *mecanismos IPC del System V*, es decir, los semáforos, las colas de mensajes y la memoria compartida.

7.2 SERVICIOS IPC UNIVERSALES

Las primeras distribuciones de UNIX únicamente disponían de tres mecanismos que podían ser utilizados para la comunicación entre procesos: las señales, las tuberías y el seguimiento de procesos.

7.2.1 Señales

Las *señales* se utilizan principalmente para notificar a un proceso eventos asíncronos. Originariamente fueron concebidas para el tratamiento de errores, aunque también pueden ser utilizadas como mecanismo IPC. Las versiones modernas de UNIX reconocen más de 31 señales diferentes. La mayoría tienen un significado predefinido, pero existen dos, SIGUSR1 y SIGUSR2, que pueden ser utilizadas por los usuarios según sus necesidades. Un proceso puede enviar una señal a un proceso o a un grupo de procesos usando por ejemplo la llamada al sistema `kill`. Además el núcleo genera señales internamente en respuesta de distintos eventos.

Como mecanismo IPC, las señales poseen varias limitaciones

- Las señales resultan costosas en relación a las tareas que suponen para el sistema. El proceso que envía la señal debe realizar una llamada al sistema; el núcleo debe interrumpir al proceso receptor y manipular la pila de usuario de dicho proceso, para invocar al manipulador de la señal y posteriormente poder retomar la ejecución del proceso interrumpido.
- Tienen un ancho de banda limitado, ya que solamente existen 31 tipos de señales distintas (en SVR4 y BSD4.3).
- Una señal puede transportar una cantidad limitada de información.

En conclusión, las señales son útiles para la notificación de eventos, pero resultan poco útiles como mecanismo IPC.

7.2.2 Tuberías

En su implementación tradicional, una *tubería* es un mecanismo de comunicación unidireccional, que permite la transmisión de un flujo de datos no estructurados de tamaño fijo. Unos procesos (emisores) pueden escribir datos en un extremo de la tubería y otros procesos (receptores) pueden leer estos datos en el otro extremo (ver Figura 7.1). Si bien debe quedar claro que en un cierto instante de tiempo solamente un proceso estará usando la tubería, bien para escribir o bien para leer. Una vez que los datos son leídos por un proceso, estos son borrados de la tubería y en consecuencia ya no pueden ser leídos por otros procesos.

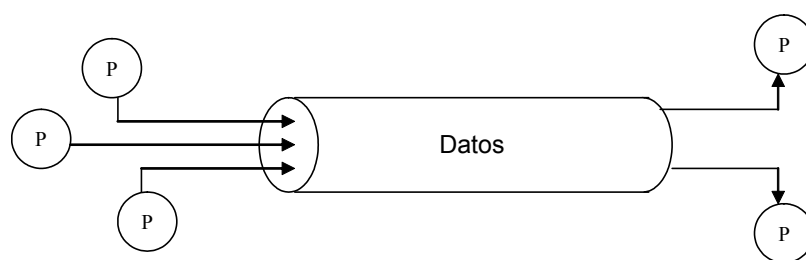


Figura 7.1: Datos fluyendo a través de una tubería

Las tuberías proporcionan un mecanismo de control del flujo de datos bastante simple. Un proceso intentando leer de una tubería vacía se bloqueará hasta que se escriban datos en la tubería. Asimismo, un proceso intentando escribir en una tubería llena se bloqueará hasta que otro proceso lea (y entonces se borren) los datos de la tubería.

Existen dos tipos de tuberías, las *tuberías sin nombre* (llamadas simplemente *tuberías*) y las *tuberías con nombre* o *ficheros FIFO*².

7.2.2.1 Tuberías sin nombre

Las *tuberías sin nombre* se crean invocando a la llamada al sistema `pipe` y solamente pueden ser utilizadas por el proceso que hace la llamada y sus descendientes. La sintaxis de esta llamada es:

```
resultado=pipe(tubería);
```

donde `tubería` es un array entero de dos elementos, mientras que `resultado` es una variable entera. Si la llamada al sistema se ejecuta con éxito en `resultado` se almacenará el valor 0 y en `tubería` se habrán almacenado dos descriptores de ficheros. Para leer de la tubería hay que usar el descriptor almacenado en `tubería[0]`, mientras que para escribir en la tubería hay que usar el descriptor almacenado en `tubería[1]`. En caso de error durante la ejecución de `pipe` en `resultado` se almacenará el valor -1.

Como se describió en la sección 5.2 cuando un proceso invoca a la llamada al sistema `fork` para crear un proceso hijo éste hereda todos los descriptores de ficheros de su progenitor. Esta es la razón por la que un proceso hijo puede también acceder a una tubería creada por su progenitor. Este mismo razonamiento se aplica para los descendientes de éste proceso hijo. De esta forma, en cada tubería pueden escribir y leer varios procesos relacionados genealógicamente. Cada uno de estos procesos puede escribir o/y leer en la tubería.

Normalmente, no obstante, una tubería suele ser compartida entre dos procesos, cada uno poseyendo un extremo. Aplicaciones típicas, como los intérpretes de comandos, manipulan de forma automática los descriptores para que en una tubería solamente pueda escribir un proceso y solamente pueda leer otro proceso (relacionado genealógicamente con el primero), usándola así para transmitir un flujo de datos en una sola dirección.

Como mecanismo IPC, las tuberías proporcionan una forma eficiente de transferir datos de un proceso a otro. Sin embargo poseen algunas limitaciones importantes:

- Una tubería no puede ser utilizada para transmitir datos a múltiples procesos receptores de forma simultánea, ya que al leer los datos de la tubería estos son borrados.

² FIFO es el acrónimo derivado de término inglés “First-In, First-Out”.

- Si existen varios procesos que desean leer en un extremo de la tubería, un proceso que escriba en el otro extremo no puede dirigir los datos a un proceso en concreto. Asimismo, si existen varios procesos que desean escribir en la tubería, no existe forma de determinar cual de ellos envía los datos.
- Si un proceso envía varios mensajes de diferente longitud en una sola operación de escritura en la tubería, el proceso que lee el otro extremo de la tubería no puede determinar cuantos mensajes han sido enviados, o donde termina un mensaje y donde empieza el otro, ya que los datos en la tubería son tratados como un flujo de bytes no estructurados de tamaño fijo.

Existen varias formas de implementar las tuberías. La aproximación tradicional (en SVR2, por ejemplo) es utilizar los mecanismos del sistema de ficheros y asociarla un nodo-i y una entrada en la tabla de ficheros.

La mayoría de las distribuciones basadas en BSD utilizan conectores (sockets) para implementar una tubería. Los *conectores* son un tipo de fichero que se utiliza como canal de comunicación entre procesos. Aunque un conector es tratado sintácticamente como un fichero; semánticamente no lo es. Esto significa que no tiene los problemas de velocidad inherentes al acceso a disco. Los conectores se utilizan sobre todo para la implementación de comunicaciones en red.

SVR4 proporciona tuberías bidireccionales basadas en *streams*. Un *stream* es una ruta de transferencia de datos entre un driver en el espacio del núcleo y un proceso en el espacio de usuario. Un stream posibilita una comunicación *full-duplex*, es decir, permite que un proceso pueda actuar como emisor o receptor en cualquier instante.

◆ Ejemplo 7.1:

El siguiente programa en C ilustra el envío de mensajes entre un proceso emisor y otro receptor a través de una tubería sin nombre.

```
#include <stdio.h>
#define MAX 256
main()
{
    int tuberia[2];
    int pid;
    char mensaje[MAX];
    [1] if (pipe(tuberia)==-1)
    {
```

```
[2]         perror("pipe");
[3]         exit(-1);
    }
[4]     if ((pid=fork())== -1)
    {
        perror("fork");
        exit(-1);
    }
    else if (pid==0)
    {
[5]         while      (read(tuberia[0],      mensaje,      MAX)>0      &&
strcmp(mensaje,"FIN")!=0)
[6]             printf("\nProceso receptor. Mensaje: %s\n", mensaje);
            close(tuberia[0]);
            close(tuberia[1]);
            exit(0);
    }
    else
    {
[7]         while(printf("Proceso emisor. Mensaje: ")!=0...
            && gets(mensaje)!=NULL...
            && write(tuberia[1], mensaje, strlen(mensaje)+1)>0...
            && strcmp(mensaje,"FIN")!=0);
            close(tuberia[0]);
            close(tuberia[1]);
            exit(0);
    }
}
```

Programa 7.1

En primer lugar **[1]** se invoca a la llamada al sistema `pipe` para crear una tubería sin nombre, y se comprueba si se ha ejecutado con éxito. Si es así en `tuberia[0]` se habrá almacenado un descriptor de fichero para poder leer en la tubería, mientras que en `tubería[1]` se habrá almacenado un descriptor de fichero para poder escribir en la tubería, además la llamada devuelve un 0. Si se produce un error durante la ejecución de `pipe` la llamada devuelve un -1, se imprime en pantalla **[2]** `pipe` seguido de “:” y del mensaje asociado al identificador de error contenido en la variable `errno`. Acto seguido se invoca **[3]** a la llamada `exit` para finalizar el programa.

A continuación, se invoca a la llamada **[4]** al sistema `fork` para crear un proceso hijo y se comprueba que se ha ejecutado con éxito.

El proceso hijo (receptor) se va a encargar [5] de leer un mensaje de la tubería y [6] presentarlo en pantalla. El ciclo de lectura y presentación termina al leer el mensaje "FIN".

Por otra parte, el proceso padre (emisor) se va a encargar [7] de leer un mensaje de la entrada estándar y, acto seguido, escribirlo en la tubería para que lo reciba el proceso hijo. El ciclo de lectura de la entrada estándar y escritura en la tubería terminará cuando se introduzca el mensaje "FIN".

En dicho caso, tanto el proceso padre como el hijo procederán a cerrar los descriptores de ficheros asociados a la tubería mediante el uso de la llamada al sistema `close` y ha finalizar su ejecución mediante la invocación de `exit`.



7.2.2.2 Tuberías con nombre o ficheros FIFO

Las *tuberías con nombre o ficheros FIFO* se crean invocando a la llamada al sistema `mknod` y pueden ser utilizadas por cualquier proceso siempre que disponga de los permisos adecuados. La sintaxis de esta llamada es:

```
resultado= mknod(ruta, modo, 0);
```

El parámetro de entrada `ruta` permite especificar el nombre del fichero FIFO. Mientras que `modo` permite especificar el tipo de fichero (`S_IFIFO`) y los usuales permisos de acceso. Si la llamada al sistema se ejecuta con éxito en `resultado` se almacenará el valor 0, en caso contrario se almacenará el valor -1.

◆ Ejemplo 7.2:

La línea de código C:

```
mknod("fifo1", S_IFIFO|0666, 0)
```

invoca a la llamada al sistema `mknod` para crear en el directorio actual un fichero FIFO de nombre `fifo1` con permisos de lectura y escritura para todos los usuarios.



También existe un comando `mknod` que puede usarse desde la línea de ordenes del terminal.

Los ficheros FIFO poseen las siguientes ventajas sobre las tuberías sin nombre:

- Tienen un nombre en el sistema de archivo.
- Pueden ser accedidos por procesos sin ninguna relación familiar.
- Son persistentes, es decir, continúan existiendo hasta que un proceso los desenlaza explícitamente. Por tanto son útiles para mantener datos que deban sobrevivir a los usuarios activos

Asimismo, los ficheros FIFO poseen las siguientes desventajas con respecto a las tuberías sin nombre:

- Deben ser borrados de forma explícita cuando no son usados.
- Son menos seguros que las tuberías, puesto que cualquier proceso con los privilegios adecuados puede acceder a ellos.
- Son difíciles de configurar y consumen más recursos.

7.2.2.3 Lectura y escritura en las tuberías (sin nombre y ficheros FIFO)

La E/S en una tubería es como la E/S en un fichero y de hecho también se realiza usando las llamadas al sistema `read` y `write` sobre los descriptors de la tubería. Un proceso es incapaz de darse cuenta de que el fichero que esta leyendo es en realidad una tubería.

Los procesos emisores añaden datos al final, mientras que los procesos receptores leen datos desde la cabecera. Una vez que un dato ha sido leído, es eliminado de la tubería y no está disponible para otros procesos receptores. El núcleo define un parámetro denominado `PIPE_BUF` (5120 bytes por defecto), que limita la cantidad de datos que una tubería puede mantener. Si un proceso emisor produjese que una tubería rebosara, éste proceso se bloquearía hasta que se habilitase espacio en la tubería mediante las operaciones de lecturas oportunas. Si un proceso intenta escribir más de `PIPE_BUF` bytes en una sola llamada, el núcleo no puede garantizar la atomicidad de la escritura.

El tratamiento de la operación de lectura es ligeramente diferente. Si el tamaño requerido es mayor que la cantidad de datos existentes actualmente en la tubería, el núcleo lee los datos que están disponibles y retorna el número de bytes leídos al proceso solicitante. Si no existe disponible ningún dato, el proceso receptor se bloqueará hasta

que otro proceso escriba en la tubería. La especificación de la opción `O_NDELAY` en el campo `modo` de `mknod` pone a la tubería en modo no bloqueante, es decir, las lecturas y escrituras se completarán sin bloquear, transfiriendo tantos datos como sea posible.

Las tuberías mantiene un contador de los procesos receptores y de los procesos emisores activos. Cuando el último proceso emisor activo cierra la tubería, el núcleo despierta a todos los procesos receptores, para que puedan leer, si lo desean, los datos que quedan en la tubería. Una vez que la tubería está vacía, los procesos receptores obtendrán un valor de retorno de 0 desde la siguiente llamada a `read` y lo interpretarán como el final del fichero. Si el último proceso receptor cierra la tubería, el núcleo envía una señal `SIGPIPE` a los procesos emisores bloqueados. Las siguientes operaciones de escritura devolverán un error `EPIPE`.

7.2.3 Seguimiento de procesos

La llamada al sistema `ptrace` suministra un conjunto de servicios para el seguimiento de procesos. Principalmente es utilizada por programas depuradores. Utilizando `ptrace`, un proceso puede controlar la ejecución de un proceso hijo. Su sintaxis es:

```
ptrace(cmd, id, addr, data);
```

donde `id` es el `pid` del proceso hijo, `addr` se refiere a una posición en el espacio de direcciones del hijo, y la interpretación del argumento `data` depende de `cmd`. El argumento `cmd` permite al padre realizar las siguientes operaciones:

- Lectura o escritura de una palabra en el espacio de direcciones, en el área U o en los registros de propósito general asociados al proceso hijo.
- Interceptar determinadas señales. Cuando una señal interceptada es generada para el hijo, el núcleo suspenderá al hijo y notificará al padre el evento.
- Configura puntos de chequeo en el espacio de direcciones del hijo.
- Reanudar la ejecución de un hijo suspendido o parado.
- Reanudar la ejecución del hijo pero solo durante una instrucción, ejecutada la cual volverá a suspenderse el hijo
- Terminar al proceso hijo.

Típicamente un proceso padre crea un hijo, y éste invoca a la llamada `ptrace` para permitir al padre controlarle. El padre entonces utiliza la llamada al sistema `wait` para esperar por un evento que cambie el estado del proceso hijo. Cuando el evento ocurre, el núcleo despierta al padre. El valor de retorno de `wait` indica que el hijo se ha parado y suministra información sobre el evento que ha causado esta parada. El padre entonces controla al hijo mediante las operaciones que se hayan especificado en `ptrace`.

Aunque `ptrace` ha permitido el desarrollo de muchos depuradores, tiene varios inconvenientes y limitaciones:

- Un proceso solo puede controlar la ejecución de su hijo. Si éste hijo crea otro proceso, el depurador no puede controlar la ejecución de este nuevo proceso o sus descendientes.
- `ptrace` es extremadamente ineficiente, requiere de varios cambios de contexto para transferir una sola palabra desde el hijo al padre. Estos cambios de contexto son necesarios porque el depurador no tiene acceso directo al espacio de direcciones del hijo.
- Un depurador no puede seguir a un proceso que ya se está ejecutando, puesto que el hijo primero necesita llamar a `ptrace` para informar al núcleo de que desea ser seguido.

Durante mucho tiempo, `ptrace` era la única herramienta para depurar programas. Los sistemas UNIX modernos tales como SVR4 o Solaris suministran servicios de depuración más eficientes.

7.3 MECANISMOS IPC DEL SYSTEM V

7.3.1 Consideraciones generales

Los mecanismos IPC descritos en la sección anterior no satisfacían las necesidades de muchas aplicaciones. Un gran avance llegó con el UNIX System V, que suministraba tres nuevos mecanismos: *semáforos*, *colas de mensajes* y *memoria compartida*, que se conocen de forma colectiva como *mecanismos IPC del System V*. Posteriormente estos mecanismos fueron implementados por la mayoría de las distribuciones de UNIX, incluso las BSD.

7.3.1.1 Características comunes de los mecanismos IPC del System V

Los *mecanismos IPC del System V* están implementados en el sistema como una unidad y comparten características comunes, entre las que se encuentran:

- 1) Cada tipo de mecanismo IPC tiene asignada una *tabla en el espacio de memoria del núcleo* de tamaño fijo. Por lo tanto en el núcleo existen tres tablas relacionadas con los mecanismos IPC: una para semáforos, otra para mensajes y una tercera para la memoria compartida.
- 2) Cada tabla asignada a un tipo de mecanismo IPC posee un número de entradas configurable. Cada entrada contiene información relativa a una instancia de dicho mecanismo IPC o canal IPC.
- 3) Cada entrada de la tabla tiene asignada una *llave numérica*, que permite controlar el acceso a dicha instancia del mecanismo IPC.
- 4) Cada entrada de la tabla asociada a un tipo de mecanismo IPC tiene asignado un índice I_T para su localización dentro de la tabla.
- 5) Cada entrada de la tabla asociada a un tipo de mecanismo IPC tiene almacenada una estructura `ipc_perm` que presenta la siguiente definición:

```
struct ipc_perm
{
    ushort uid;      → Identificador de usuario del proceso propietario
                     del recurso.
    ushort gid;      → Identificador de grupo del proceso propietario del
                     recurso.
    ushort cuid;     → Identificador de usuario del proceso creador del
                     recurso
    ushort cgid;     → Identificador de grupo del proceso creador el
                     recurso.
    ushort mode;     → Modo de acceso (permisos de lectura, escritura y
                     ejecución para el usuario, el grupo y otros usuarios)
    ushort seq;      → Número de secuencia. Es un contador que lo
                     mantiene el núcleo y que se incrementa siempre que
                     se cierra una instancia o canal de un mecanismo
                     IPC. Este contador es necesario para identificar los
                     canales abiertos e impedir que mediante una
                     elección aleatoria del identificador de canal, un
                     proceso pueda adquirirlo.
```

```
key_t key;    → Llave de acceso
}
```

- 6) Cada entrada de la tabla asociada a un tipo de mecanismo IPC, además de la estructura `ipc_perm`, tiene almacenada también otras informaciones como por ejemplo el *pid* del último proceso que ha utilizado la entrada y la fecha de la última actualización o acceso.
- 7) Cada instancia de un mecanismo IPC tiene asignado un descriptor numérico N_{IPC} elegido por el núcleo, que la referencia de forma única y que será utilizado para localizar la instancia rápidamente cuando se realicen operaciones sobre ella.
- 8) Cada tipo de mecanismo IPC dispone de una llamada al sistema tipo `get` [`shmget` (memoria compartida), `semget` (semáforos) y `msgget` (colas de mensajes)] que permite crear una nueva instancia de un determinado tipo de mecanismo IPC o acceder a alguna ya existente.
- 9) Cada tipo de mecanismo IPC dispone de una llamada al sistema tipo `ctl` [`shmctl` (memoria compartida), `semctl` (semáforos) y `msgctl` (colas de mensajes)] que permite acceder a la información administrativa y de control de una instancia de un mecanismo IPC.

7.3.1.2 Asignación de un índice I_T a una instancia N_{IPC}

El núcleo calcula el descriptor numérico N_{IPC} que asigna a una instancia de un mecanismo IPC usando la siguiente fórmula:

$$N_{IPC} = seq * N_T + I_T \quad (1)$$

donde *seq* es el número de secuencia de la instancia, N_T es el tamaño de la tabla asociada al mecanismo IPC, e I_T es el índice de la instancia en la tabla. Esto asegura que un nuevo N_{IPC} es generado si una entrada de la tabla de un cierto mecanismo IPC es reutilizada, puesto que *seq* es incrementado en una unidad. Asimismo se evita que los procesos accedan a una instancia usando un descriptor viejo.

El usuario pasa el N_{IPC} como un argumento de las siguientes llamadas al sistema asociadas con la instancia del mecanismo IPC. El núcleo traduce el N_{IPC} a la posición de la instancia en la tabla usando la fórmula:

$$I_T = N_{IPC} \bmod(N_T) = N_{IPC} \% N_T \quad (2)$$

♦ **Ejemplo 7.3:** La tabla asociada a un determinado tipo de mecanismo IPC posee $N_T=100$ entradas. Calcular I_T de en los siguientes casos: a) $N_{IPC}=5$. b) $N_{IPC}=30$. c) $N_{IPC}=101$. d) $N_{IPC}=303$.

Aplicando la fórmula (2) se obtiene:

a) $I_T = 5 \bmod(100) = 5 \% 100 = 5$

b) $I_T = 30 \bmod(100) = 30 \% 100 = 30$

c) $I_T = 101 \bmod(100) = 101 \% 100 = 1$

d) $I_T = 303 \bmod(100) = 303 \% 100 = 3$

♦

♦ **Ejemplo 7.4:** La tabla asociada a un determinado tipo de mecanismo IPC posee $N_T=100$ entradas. Calcular los descriptores posibles N_{IPC} de la entrada $I_T=1$ si el número de secuencia puede tomar como máximo el valor 3.

Los posibles valores N_{IPC} de la entrada $I_T=1$ se obtendrán usando la fórmula (1) para los valores $seq=0, 1, 2$ y 3 .

$seq=0 \quad N_{IPC} = 0*100+1 = 1$

$seq=1 \quad N_{IPC} = 1*100+1 = 101$

$seq=2 \quad N_{IPC} = 2*100+1 = 201$

$seq=3 \quad N_{IPC} = 3*100+1 = 301$

Supóngase que el descriptor asociado a una instancia de un mecanismo IPC es $N_{IPC}=201$. En un determinado instante dicha instancia es eliminada de la tabla. Cuando se vuelva a utilizar dicha entrada de la tabla, es decir, cuando se cree otra nueva instancia de un mecanismo IPC, el núcleo le asignará $N_{IPC}=301$. Aquellos procesos que intenten acceder con $N_{IPC}=201$ recibirán una señal de error ya que no es una entrada válida. Los descriptores N_{IPC} son reciclados por el núcleo transcurrido un cierto intervalo de tiempo.

♦

7.3.1.3 Creación de llaves

Cada entrada de una tabla de un determinado tipo de mecanismo IPC tiene una *llave numérica*, que permite controlar el acceso a dicha instancia del mecanismo IPC. La llamada al sistema `ftok` permite a un usuario crear una llave. Su sintaxis es la siguiente:

```
resultado=ftok(ruta, letra);
```

Esta llamada tiene dos parámetros de entrada, `ruta` que es la ruta de acceso de un fichero que debe existir dentro del sistema de archivos y `letra` que es un carácter. Si la llamada se ejecuta con éxito en `resultado`, que es una variable del tipo predefinido `key_t`, se almacenará una llave. En caso de error en `resultado` se almacenará el valor `key_t-1`.

En general `ftok` produce una llave de 32 bits combinando el parámetro `letra` con el número del nodo-i del fichero del parámetro `ruta` y con el número de dispositivo del sistema de archivos al que pertenece este fichero.

◆ Ejemplo 7.5:

Las siguientes líneas de código C permiten crear una llave asociada al fichero "archivo1" y al carácter 'A':

```
#include <sys/types.h>
#include <sys/ipc.h>
...
key_t llave;
...
if((llave=ftok("archivo1",'A'))==(key_t)-1)
{
    /* Tratamiento del error al crear una llave*/
}
```

◆

7.3.1.4 Algunos comentarios sobre las llamadas al sistema tipo `get`

Un proceso adquiere una instancia de un mecanismo IPC haciendo una llamada al sistema del tipo `get`, pasándole una llave, ciertos indicadores y otros argumentos que dependen de cada mecanismo. Los indicadores permitidos son `IPC_CREAT` y `IPC_EXCL`. Su significado es el siguiente:

- `IPC_CREAT` pide al núcleo que cree la instancia si ésta no existe ya.
- `IPC_EXCL` es utilizado junto con `IPC_CREAT` y pide al núcleo que devuelva un error si la instancia ya existía.

Si no se especifica ningún indicador, el núcleo busca una instancia ya existente con la misma llave. Si la encuentra, y el proceso invocador tiene permiso de acceso, el núcleo devuelve el descriptor numérico N_{IPC} de la instancia. En caso contrario devuelve el valor -1.

Si la llave toma el valor especial `IPC_PRIVATE`, el núcleo crea una nueva instancia. En este caso la instancia no podrá ser accedida a través de posteriores llamadas tipo `get`. Por lo tanto el proceso que invoca a la llamada al sistema con este argumento tiene propiedad exclusiva sobre la instancia. Eso si, el propietario puede compartir el recurso con sus hijos, que lo heredan cuando se realiza la llamada al sistema `fork`.

7.3.1.5 Algunos comentarios sobre las llamadas al sistema tipo `ctl`

Todos los tipos de mecanismos IPC poseen una llamada al sistema de control del tipo `ctl` que implementa diversos comandos. Estos comandos incluyen `IPC_STAT` y `IPC_SET` para obtener y configurar información del estado de un recurso, y `IPC_RMID` para eliminar un recurso. Los semáforos disponen de comandos adicionales para obtener y configurar los valores de un determinado semáforo perteneciente a un cierto conjunto.

Cada recurso IPC debe ser explícitamente eliminado mediante el uso del comando `IPC_RMID`. En caso contrario, el núcleo considera que se encuentra activo incluso aunque todos los procesos que lo estaban utilizando hayan terminado. Por lo tanto, un recurso IPC puede perdurar y ser utilizado más allá del tiempo de vida de los procesos que lo han estado utilizando. Esta propiedad puede ser bastante útil. Por ejemplo, un proceso puede escribir datos en una región de memoria compartida o un mensaje en una cola y después finalizar. Más tarde, otro proceso puede recuperar estos datos.

Únicamente el creador, el propietario actual o el superusuario pueden usar el comando `IPC_RMID`. La eliminación de un recurso afecta a todos los procesos que actualmente acceden a él, y el núcleo debe asegurarse de que todos estos procesos tratan este evento adecuadamente.

7.3.2 Semáforos

Los *semáforos* son objetos que pueden tomar valores enteros que soportan dos operaciones atómicas: $P()$ y $V()$ ³. La operación $P()$ decrementa en una unidad el valor del semáforo y bloquea al proceso que solicita la operación si su nuevo valor es menor que cero. La operación $V()$ incrementa en una unidad el valor del semáforo; si el valor resultante es mayor o igual a cero, $V()$ despierta a los procesos que estuvieran esperando por este evento.

Los semáforos pueden ser utilizados para implementar varios protocolos de sincronización. Por ejemplo, considérese el problema de administrar un cierto recurso limitado, es decir, un recurso con un número fijo de instancias. Los procesos intentan adquirir una instancia del recurso, y lo liberan cuando han terminado de utilizarlo. Este recurso puede ser representado por un semáforo que es inicializado al número de instancias. La operación $P()$ es utilizada al intentar adquirir el recurso, decrementará el semáforo cada vez que tenga éxito. Cuando el semáforo alcanza el valor cero significará que no existen instancias libres, por lo que cualquier otra operación $P()$ bloqueará al proceso. Liberar un recurso resulta en una operación $V()$, que incrementa el valor del semáforo, produciendo que los procesos bloqueados despierten

En el espacio de memoria del núcleo existe una *tabla de semáforos* con información de todos los semáforos existentes en el sistema. Cada entrada de esta tabla se denota por un identificador numérico *semid* que hace referencia a un conjunto de semáforos. Además cada entrada se implementa mediante la estructura `semid_ds` cuya definición es:

```
struct semid_ds {  
    struct ipc_perm sem_perm;    → Estructura que contiene los permisos de acceso  
    struct sem *sem_base;       → Puntero al primer semáforo del conjunto  
    ushort sem_nsems;           → Número de semáforos en el conjunto  
    time_t sem_otime;           → Fecha de la última operación  
    time_t sem_ctime;           → Fecha del último cambio mediante semctl  
}
```

³ Los nombres de las operaciones $P()$ y $V()$ derivan de las palabras holandesas *Proberen* (comprobar) y *Verhogen* (incrementar) originariamente establecidas por E.W. Dijkstra en 1965.

Por otra parte, para cada semáforo perteneciente a un conjunto el núcleo guarda su valor y la información de sincronización en una estructura `sem`, cuya definición se muestra a continuación:

```
struct sem {

    ushort semval;    → Valor actual del semáforo

    pid_t sempid;     → pid del proceso que ha solicitado la última operación, llamando a
                        semop.

    ushort semncnt;   → Número de procesos esperando a que semval se incremente ( >0 ).

    ushort semzcnt;   → Número de procesos esperando a que semval valga cero.

};
```

♦ Ejemplo 7.6:

En la Figura 7.2 se representan las estructuras de datos del núcleo necesarias para el manejo de semáforos. A modo de ejemplo se han supuesto dos entradas activas en la tabla de semáforos, es decir, hay dos conjuntos de semáforos `semid=0` y `semid=1`. La entrada `semid=0` contiene información de un conjunto con 4 semáforos, mientras que la entrada `semid=1` contiene información de un conjunto con 2 semáforos.

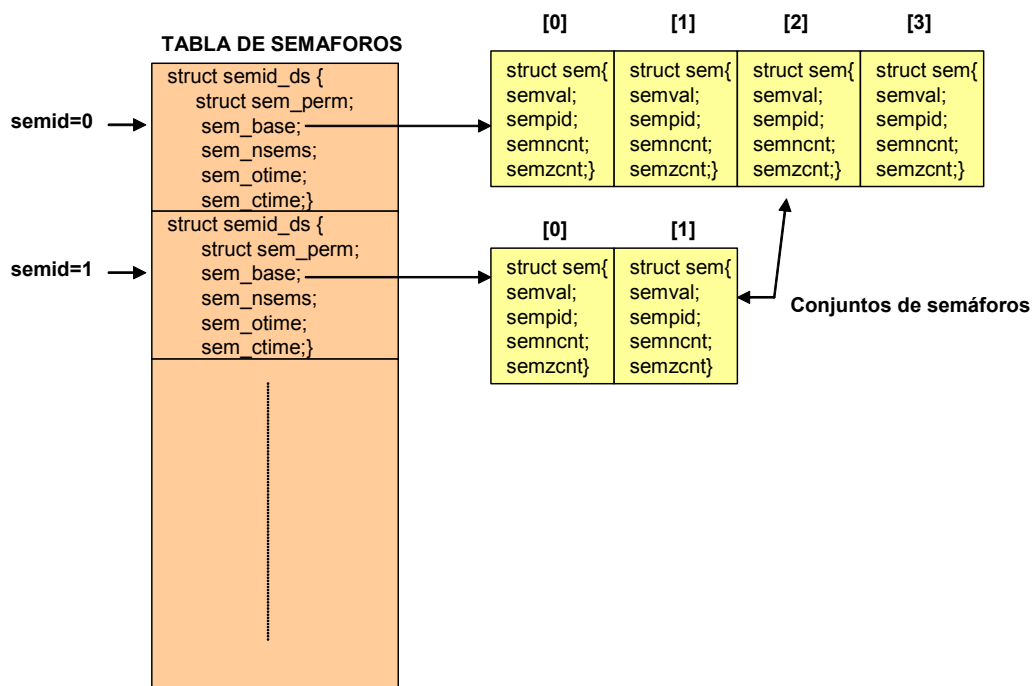


Figura 7.2: Estructura de datos del núcleo necesarias para el manejo de semáforos

Obsérvese como en cada entrada de la tabla de semáforos hay almacenada una estructura `semid_ds` que entre otras informaciones (`sem_perm`, `sem_nsems`, `sem_otime`, `sem_ctime`) posee un puntero `sem_base` que apunta al primer semáforo de cada conjunto.

Por otra parte cada semáforo de un conjunto viene definido por una estructura `sem` que contiene la siguiente información: `semval`, `sempid`, `semncnt` y `semzcnt`.

◆

7.3.2.1 Creación u obtención de un conjunto de semáforos

La llamada al sistema `semget` crea u obtiene un array o conjunto de semáforos. Su sintaxis es:

```
semid=semget(key,count,flags);
```

donde `key` es una llave numérica del tipo predefinido `key_t` o bien la constante `IPC_PRIVATE`, `count` es el número entero de semáforos del conjunto o array asociados a `key` y `flags` es una máscara de indicadores (máscara de bits). Estos indicadores permiten especificar, de forma similar a como se hace para los ficheros, los permisos de acceso al conjunto de semáforos. Asimismo en `flags` también se pueden introducir los indicadores `IPC_CREAT` e `IPC_EXCL`.

Si la llamada al sistema `semget` se ejecuta con éxito entonces en `semid` se almacenará el identificador entero de un array o conjunto de `count` semáforos asociados a la llave `key`. Si no existe un conjunto de semáforos asociado a la llave la orden fallará y en `semid` se almacenará el valor `-1` a menos que se haya realizado con el indicador `IPC_CREAT` de `flags` activo, lo que fuerza a crear un nuevo conjunto de semáforos. También se crea un nuevo conjunto de semáforos si el parámetro `key` se configura al valor `IPC_PRIVATE`.

◆ Ejemplo 7.7:

Las siguientes líneas de código C muestran como crear un nuevo conjunto de cinco semáforos, asociado a la llave creada a partir del fichero “ayudante” y la clave ‘J’. Este conjunto de semáforos se va a crear con permisos de lectura y modificación para el usuario.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
...
```

```

int semid;
key_t llave;
...
llave=ftok("ayudante",'J');
if(llave==(key_t)-1)
{
/*Se ha producido un error al crear la llave.
Código de tratamiento del error*/
}

/*Creación del conjunto de semáforos*/
semid=semget(llave, 5, IPC_CREAT| 0600);
if (semid==-1)
{
/*Error al crear el conjunto de semáforos.
Código de tratamiento del error*/
}

```

◆

7.3.2.2 Realización de operaciones con los elementos de un conjunto de semáforos

La llamada al sistema `semop` es utilizada para realizar operaciones sobre los elementos de un determinado conjunto de semáforos. Su sintaxis es:

```
resultado=semop(semid, sops, nsops);
```

donde `semid` es un identificador de un array o conjunto concreto de semáforos, `sops` es un puntero a un array de estructuras del tipo `sembuf` que indican las operaciones que se van a llevar a cabo sobre los semáforos, y `nsops` es el número total de elementos que tiene el array de operaciones, es decir, el número total de operaciones.

En general, el núcleo lee el array de operaciones `sops` del espacio de direcciones del usuario y verifica que los números de los semáforos son legales y que el proceso tiene los permisos necesarios para leer o cambiar los valores de los semáforos. Si no cuenta con los permisos adecuados la llamada `semop` falla y en `resultado` se almacena el valor -1.

La definición de la estructura del tipo `sembuf` utilizada es:

```
struct sembuf{
    unsigned short sem_num;
    short sem_op;
    short sem_flg;
}
```

El significado de cada uno de los elementos de una estructura `sembuf` es el siguiente:

- `sem_num` identifica a uno de los semáforos del conjunto `semid`. Su valor está comprendido entre 0 y N-1, donde N es el número total de semáforos en el conjunto.
- `sem_op` especifica la acción a realizar en el semáforo elegido. Los valores de `sem_op` se interpretan de la siguiente manera:
 - `sem_op > 0`. Añadir `sem_op` al valor actual del semáforo. Los procesos que estaban durmiendo en espera de que el valor fuese incrementado serán despertados.
 - `sem_op = 0`. Bloquear el proceso hasta que el valor del semáforo sea cero.
 - `sem_op < 0`. Bloquear el proceso hasta que el valor del semáforo sea mayor o igual que el valor absoluto de `sem_op`, a continuación restar `sem_op` de dicho valor. Si el valor del semáforo ya es superior al valor absoluto de `sem_op`, el proceso que invoca esta llamada al sistema no se bloqueará.
- `sem_flg`, permite suministrar dos indicadores a la llamada. El indicador `IPC_NOWAIT` pide al núcleo que devuelva un error en vez de bloquear al proceso. Asimismo, puede ocurrir un interbloqueo si un proceso que retiene un semáforo termina prematuramente sin liberarlo. Otros procesos esperando para adquirir dicho semáforo puede quedar para siempre bloqueados en la operación `P()`. Para evitar este problema, es posible pasar a `semop` el indicador `SEM_UNDO` para que el núcleo recuerde la operación y automáticamente la deshaga si el proceso termina.

◆ Ejemplo 7.8:

Las siguientes líneas de código C muestran como realizar una operación `P()` y otra `V()` sobre los semáforos 2 y 4 respectivamente del conjunto de semáforos `semid` que agrupa un total de 5 semáforos.

```
struct sembuf operaciones[5];
...
operaciones[0].sem_num=2; /*Semáforo número 2*/
operaciones[0].sem_op=-1; /*Operación P*/
operaciones[0].sem_flg=0;
operaciones[1].sem_num=4; /*Semáforo número 4*/
operaciones[1].sem_op=1; /*Operación V*/
operaciones[1].sem_flg=0;

semop(semid,operaciones,2);
...
```

◆

Finalmente comentar que el núcleo mantiene una lista para cada proceso que ha solicitado una operación sobre un semáforo con el indicador `SEM_UNDO`. Esta lista contiene un registro por cada operación que debe ser deshecha. Cuando un proceso termina, el núcleo chequea si tiene una lista de estas características, si la tiene el núcleo recorre la lista reconstruyendo todas las operaciones realizadas con anterioridad.

7.3.2.3 Acceso a la información administrativa y de control de un conjunto de semáforos

La llamada al sistema `semctl` permite acceder a la información administrativa y de control que posee el núcleo sobre un cierto conjunto de semáforos. Su declaración es:

```
resultado=semctl(semid, semnum, cmd, arg);
```

donde `semid` es el identificador de un array o conjunto de semáforos, `semnum` es el identificador de un semáforo concreto dentro del array, `cmd` es un número entero o una constante simbólica (ver Tabla 7.1) que especifica la operación a efectuar que indica la operación que va a realizar la llamada al sistema `semctl`, y `arg` es una unión del tipo `semun`, que se define de la siguiente forma:

```

union semun
{
    int val;                → usado con SETVAL
    struct semid_ds *buf;   → usado por IPC_STAT y por IPC_SET
    unsigned short* array; → usado por GETALL y SETALL.
}arg;

```

Si la llamada `semctl` tiene éxito, en `resultado` se almacenará un número entero cuyo valor depende del comando `cmd`. Si falla en `resultado` se almacenará el valor `-1`.

Comando	Significado
GETVAL	Se usa para leer el valor de un semáforo. Este número se almacena en <code>resultado</code> .
SETVAL	Permite inicializar un semáforo a un valor determinado que se especifica en <code>arg</code> .
GETPID	Se usa para leer el <i>pid</i> del último proceso que actuó sobre el semáforo. Este número se almacena en <code>resultado</code> .
GETNCNT	Permite leer el número de procesos que hay esperando a que se incremente el valor del semáforo. Este número se almacena en <code>resultado</code> .
GETZCNT	Permite leer el número de procesos que hay esperando a que el semáforo tome el valor cero. Este número se almacena en <code>resultado</code> .
GETALL	Permite leer el valor de todos los semáforos asociados a un identificador <code>semid</code> . Estos valores se almacenan en <code>arg</code> .
SETALL	Sirve para inicializar el valor de todos los semáforos asociados a un identificador <code>semid</code> . Los valores de inicialización deben estar en <code>arg</code> .
IPC_STAT, IPC_SET	Permiten leer y modificar la información administrativa asociada al identificador <code>semid</code> .
IPC_RMID	Indica al núcleo que debe borrar el conjunto de semáforos agrupados bajo el identificador <code>semid</code> . La operación no tendrá efecto mientras haya algún proceso que esté usando los semáforos.

Tabla 7.1: Valores posibles del parámetro `cmd` de la llamada `semctl`

♦ Ejemplo 7.9:

Las siguientes líneas de código C muestran como crear un nuevo conjunto de cinco semáforos, asociado a la llave creada a partir del fichero “ayudante” y la clave ‘J’. Este conjunto de semáforos se va a crear con permisos de lectura y modificación para el usuario. Además se inicializan los dos primeros semáforos con el valor 3 y los tres últimos con el valor 2. Finalmente se pregunta por el valor del semáforo número 2.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
...

int semid, valor;
ushort sem_conjunto[5];
...
/*Creación del conjunto de semáforos*/
semid=semget(ftok("ayudante",'J'), 5, IPC_CREAT | 0600);
if (semid== -1)
{
/*Código de tratamiento del error*/
/*Inicialización de los semáforos*/
sem_conjunto[0]=3;
sem_conjunto[1]=3;
sem_conjunto[2]=2;
sem_conjunto[3]=2;
sem_conjunto[4]=2;
semctl(semid,0,SETALL,sem_conjunto);
...
/* Pregunta por el valor del semáforo número 2*/
valor=semctl(semid,2,GETVAL,0);
```

◆

7.3.3 Colas de mensajes

Una cola de mensajes es una estructura de datos gestionada por el núcleo, en ella van a poder escribir varios procesos. Los mecanismos de sincronización para que no se produzca colisiones en el uso de la cola de mensajes son responsabilidad del núcleo. Los datos que se escriben en la cola deben tener formato de mensaje y son tratados como un todo indivisible, es decir, el proceso extrae o coloca la información en una única operación.

El mecanismo de comunicación de las colas de mensajes corresponde a la implementación del concepto de *buzón*, que permite la comunicación indirecta entre procesos. Un proceso tiene la posibilidad de depositar mensajes o extraerlos del buzón.

Cada mensaje esta tipificado, y cada proceso extraerá de una cola de mensajes aquellos que quiera extraer.

En la implementación del UNIX System V todos los mensajes son almacenados en el espacio del núcleo y tienen asociado un identificador de cola de mensaje, denominado `msqid`. Los procesos pueden leer y escribir mensajes de cualquier cola.

7.3.3.1 Estructuras de datos asociadas a los mensajes

De forma general un *mensaje* se implementa mediante una estructura que consta de dos campos: *el tipo del mensaje y el texto o cuerpo del mensaje*. El *tipo del mensaje* es un entero positivo que permite identificar al mensaje de acuerdo con una tipificación previamente establecida por el programador. Por su parte, *el texto del mensaje* es un array de caracteres que contiene el mensaje propiamente dicho.

El núcleo mantiene básicamente tres tipos de estructuras de datos para implementar las colas de mensajes: la tabla de colas de mensajes, la lista enlazada de cabeceras de mensajes asociadas a una cola y a un área de datos.

El núcleo posee una *tabla de colas de mensajes*, cada entrada en dicha tabla está asignada a una única cola de mensajes que viene identificada por un descriptor numérico `msqid`. Además, cada entrada contiene una estructura del tipo `msqid_ds`:

```
struct msqid_ds {  
  
    struct ipc_perm msg_perm;    → Estructura de los derechos de acceso.  
  
    struct msg *msg_first;       → Puntero al primer mensaje.  
  
    struct msg *msg_last;       → Puntero al último mensaje.  
  
    ushort msg_cbytes;          → Número total de bytes en la cola.  
  
    ushort msg_qbytes;          → Número máximo de bytes.  
  
    ushort msg_qnum;            → Número de mensajes en la cola.  
  
    ushort msg_lspid;           → pid del último proceso emisor.  
  
    ushort msg_lrpid;           → pid del último proceso receptor.  
  
    time_t msg_stime;           → Fecha del último envío de mensaje.  
  
    time_t msg_rtime;           → Fecha de la última recepción del mensaje.  
  
    time_t msg_ctime;           → Fecha del último cambio por msgctl.  
  
};
```

A su vez cada cola de mensajes `msgqid` tiene asociada una *lista enlazada de cabeceras de mensajes* pertenecientes a dicha cola. Cada cabecera viene descrita por una estructura `msg`, que presenta la siguiente definición:

```
struct msg{

    struct msg *msg_sig; →Puntero al mensaje siguiente.

    long msg_type;      →Tipo de mensaje.

    short msg_ts;       →Tamaño del texto del mensaje.

    char *msg_spot;     →Dirección del texto de mensaje en el área de datos del núcleo

}
```

Finalmente, el texto o cuerpo de cada mensaje perteneciente a una cola de mensajes se encuentra almacenado en un *área de datos* dentro del segmento del núcleo en memoria principal.

♦ **Ejemplo 7.10:**

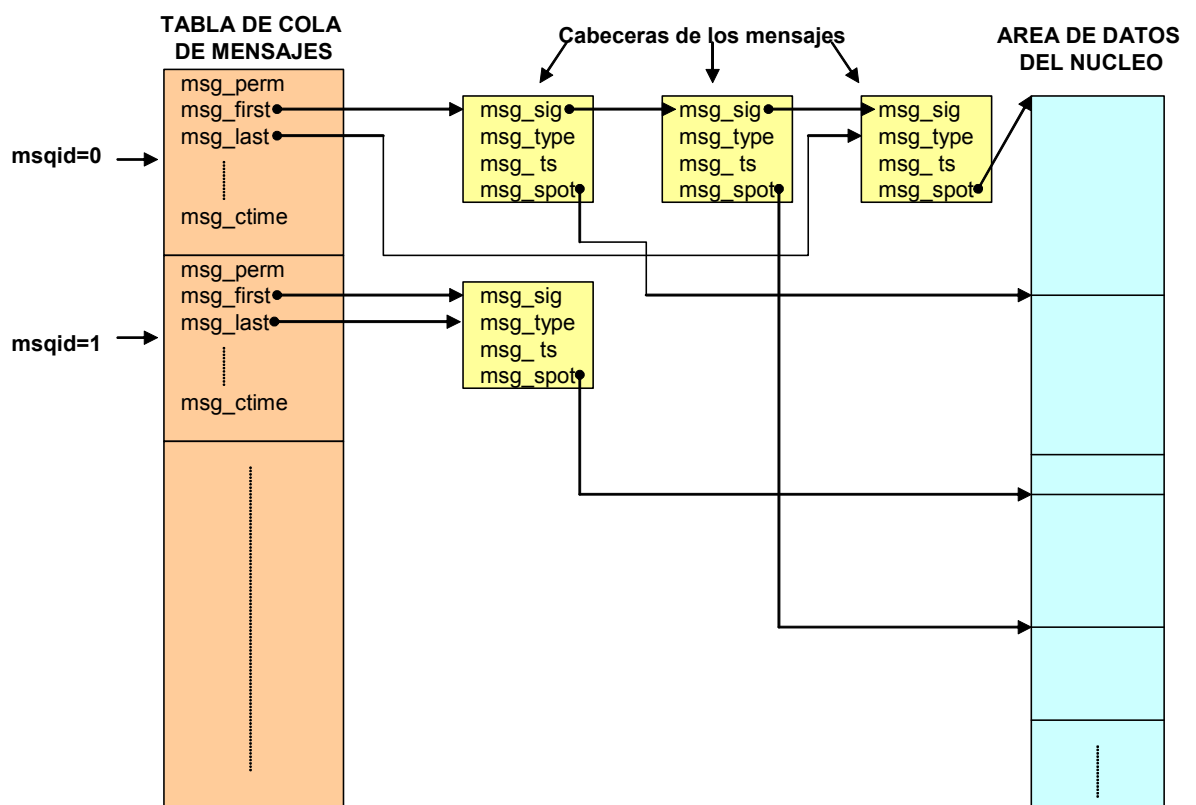


Figura 7.3: Estructuras de datos utilizadas en la implementación del mecanismo IPC de cola de mensajes.

En la Figura 7.3 se muestran las estructuras de datos del núcleo utilizadas en la implementación del mecanismo IPC de cola de mensajes. Se observa como la tabla de cola de mensajes tiene dos entradas activas (`msqid=0` y `msqid=1`). La cola `msqid=0` posee una lista enlazada de tres cabeceras de mensajes, mientras que la cola `msqid=1` posee una lista enlazada con una única cabecera de mensajes.

Cada entrada de la tabla contiene una estructura `msqid_ds` que aporta entre otras informaciones un puntero (`msg_first`) que apunta a la cabecera del primer mensaje de la cola y otro puntero (`msg_last`) que apunta a la cabecera del último mensaje de la cola. Además la cabecera de cada mensaje en una cola contiene un puntero (`msg_sig`) que apunta a la cabecera del siguiente mensaje en la cola. Se observa también como la cabecera de un mensaje contiene además el tipo de mensaje (`msg_type`), el tamaño del texto del mensaje (`msg_ts`) y la dirección del área de datos del núcleo donde se encuentra el texto de dicho mensaje (`msg_spot`).

◆

7.3.3.2 Creación u obtención de una cola de mensajes

La llamada al sistema `msgget` crea una cola de mensajes o bien permite acceder a una cola ya existente usando una clave dada. Su sintaxis es:

```
msqid=msgget(key, flags)
```

donde `key` es la clave de la cola de mensaje y `flags` es una máscara de indicadores (similar a la descritas para los semáforos). Si la llamada al sistema `msgget` se ejecuta con éxito entonces en `msqid` se almacenará el identificador entero de una cola de mensajes asociada a la llave `key`. En caso contrario en `msqid` se almacenará el valor `-1`.

◆ Ejemplo 7.11:

Las siguientes líneas de código C muestran como crear una cola de mensajes, asociada a la llave creada a partir del fichero "ayudante" y la clave `'J'`. Esta cola se va a crear con permisos de lectura y modificación para el usuario.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
int msqid;
key_t llave;
...
llave=ftok("ayudante", 'J');
```

```
msqid=msgget(llave, IPC_CREAT | 0600);  
if (msqid==-1)  
{  
    /* Error en la creación de la cola de mensajes.  
    Tratamiento del error*/  
}
```

◆

7.3.3.3 Envío de mensajes

La llamada al sistema `msgsnd` permite a un proceso enviar un mensaje desde su espacio de direcciones a una determinada cola de mensajes. Su sintaxis es:

```
resultado=msgsnd(msqid, &buffer, msgsz, msgflags);
```

donde `msqid` es un identificador de una cola de mensajes, `buffer` es la variable del espacio de direcciones del usuario que contiene el mensaje que se desea enviar, `msgsz` es la longitud del texto del mensaje en bytes, y `msgflags` es una máscara de indicadores que permite especificar el comportamiento del proceso emisor en caso de que no pueda enviarse el mensaje debido a una saturación del mecanismo de colas. Si la llamada al sistema tiene éxito en `resultado` se almacenará el valor 0, si falla se almacenará el valor -1.

Por defecto la llamada al sistema `msgsnd` es bloqueante, es decir, el proceso que la invoca pasará al estado dormido interrumpible por señales sino se puede escribir en la cola de mensajes, y se le despertará cuando se pueda escribir. También se le despertaría si la cola de mensajes fuese borrada, o recibiese una señal que no ignora. Es posible hacer que esta llamada sea no bloqueante para ello, hay que colocar el indicador `IPC_NOWAIT` en la máscara `msgflags` de `msgsnd`. En dicho caso sino se puede escribir en la cola la llamada devolverá el valor -1 y asignará a la variable `errno` el valor `EAGAIN`.

Cuando se realiza la llamada al sistema `msgsnd` el núcleo realiza la siguiente secuencia de acciones:

- 1) Comprueba que el proceso emisor tiene permiso de escritura para la cola `msqid`.
- 2) Comprueba que la longitud del mensaje no excede los límites del sistema y que no contiene demasiados bytes.

- 3) Comprueba que el tipo de mensaje es un entero positivo.
- 4) Si todas las comprobaciones anteriores son superadas con éxito, asigna espacio para el mensaje en el área de datos del núcleo y copia los datos desde el espacio de direcciones del usuario al espacio de direcciones del núcleo
- 5) Asigna una cabecera de mensaje y la coloca al final de la lista enlazada de cabeceras de mensajes de la cola de mensajes `msqid`.
- 6) Salva el tipo de mensaje y su tamaño en la cabecera del mensaje.
- 7) Configura la cabecera del mensaje para que apunte al texto de mensaje en el área de datos del núcleo.
- 8) Actualiza varios campos de tipo estadístico en la entrada de la tabla de colas asignada a la cola `msqid`.
- 9) El núcleo despierta a los procesos que estaban dormidos esperando por la llegada de un mensaje en dicha cola.
- 10) Si el número de bytes en la cola excede el límite de la cola, el proceso emisor dormirá hasta que otros mensajes sean eliminados de la cola.
- 11) Si el proceso estableció en su llamada a `msgsnd` (indicador `IPC_NOWAIT` del campo `msgflag`) que no desea esperar, entonces la llamada devolverá el valor `-1`.

7.3.3.4 Recepción de mensajes

La llamada al sistema `msgrcv` permite que un proceso pueda extraer un mensaje de una determinada cola de mensajes. Su sintaxis es:

```
resultado=msgrcv(msqid, &buffer, msgsz, msgtipo, msgflags);
```

donde `msqid` es un identificador de una cola de mensajes, `buffer` es la variable del espacio de direcciones del usuario donde se va almacenar el mensaje, `msgsz` es la longitud del texto del mensaje en bytes, `msgtipo` indica el tipo del mensaje que se desea extraer y `msgflags` es una máscara de indicadores que permite especificar el comportamiento del proceso receptor en caso de que no pueda extraerse ningún mensaje del tipo especificado. Si la llamada al sistema tiene éxito en `resultado` se almacenará

el número de bytes del mensaje recibido (este número no incluye los bytes asociados al tipo de mensaje). En caso de error en `resultado` se almacenará el valor -1.

El argumento `msgtipo` puede tomar los siguientes valores:

- `msgtipo = 0`. Se extrae el primer mensaje que haya en la cola independientemente de su tipo. Corresponde al mensaje más viejo.
- `msgtipo > 0`. Se extrae el primer mensaje del tipo `msgtype` que haya en la cola.
- `msgtipo < 0`. Se extrae el primer mensaje que cumpla que su tipo es menor o igual al valor absoluto de `msgtipo` y a la vez sea el más pequeño de los que hay.

♦ Ejemplo 7.12:

Supóngase que se tiene una cola que contiene tres mensajes cuyos tipos son 3, 1 y 2, respectivamente, y un usuario solicita un mensaje con `msgtipo=-2`, ¿Que tipo de mensaje extrae el núcleo?

Solución:

Se extrae el primer mensaje que cumpla que su tipo es menor o igual al valor absoluto de `msgtipo` y a la vez sea el más pequeño de los que hay. Es decir:

$$\text{Tipo del mensaje devuelto} = \min \{3, 1, 2\} \leq |-2|$$

$$\text{Tipo del mensaje devuelto} = 1$$

♦

Por defecto la llamada al sistema `msgrcv` es bloqueante, es decir, el proceso receptor pasará al estado dormido interrumpible por señales sino se no puede extraer ningún mensaje del tipo especificado, y se le despertará cuando se pueda extraer. También se le despertaría si la cola de mensajes fuese borrada, o recibiese una señal que no ignora. Es posible hacer que esta llamada sea no bloqueante para ello, hay que colocar el indicador `IPC_NOWAIT` en la máscara `msgflags` de `msgrcv`. En dicho caso sino se puede escribir en la cola la llamada devolverá el valor -1 y asignará a la variable `errno` el valor `ENOMSG`.

Por otra parte, si se intenta extraer un mensaje de longitud mayor al tamaño especificado por el argumento `msgsz` de `msgrcv` se producirá un error, a menos que en

el campo `msgflags` se coloque el indicador `MSG_NOERROR`. En este caso se extraerá únicamente los `msgsz` primeros bytes del mensaje.

◆ **Ejemplo 7.13:**

Las siguientes líneas de código C muestran como enviar y recibir un mensaje del tipo 2, que se compone de una cadena de 50 caracteres:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
...
key_t llave;
int msqid;
struct
{
    long tipo;
    char cadena[50];
} mensaje;
int longitud=sizeof(mensaje)-sizeof(mensaje.tipo);
...
llave=ftok("ayudante",'J');
msqid=msgget(llave, IPC_CREAT | 0600);

...
...
/*Envío del mensaje*/
mensaje.tipo=2;
strcpy(mensaje.cadena,"MI PRIMER MENSAJE");
if(msgsnd(msqid, &mensaje,longitud,0)==-1)
{
    /*Error durante el envío del mensaje.
    Tratamiento del error.*/
}
...
/*Recepción del mensaje*/
mensaje.tipo=2;
if(msgrcv(msqid, &mensaje,longitud,2,0)==-1)
{
```

```
    /*Error durante la recepción del mensaje.  
    Tratamiento del error.*/  
}
```

◆

7.3.3.5 Acceso a la información administrativa y de control de una cola de mensajes

La llamada al sistema `msgctl` permite leer y modificar la información estadística y de control de una cola de mensajes, su declaración es:

```
resultado=msgctl(msqid, cmd, &buffer);
```

donde `msqid` es el identificador de la cola, `cmd` es un número entero o una constante simbólica que especifica la operación a efectuar, y `buffer` es una estructura del tipo predefinido `msqid_ds` que contiene los argumentos de la operación. Si la llamada `msgctl` tiene éxito, en `resultado` se almacenará un número entero cuyo valor depende del comando `cmd`. Si falla en `resultado` se almacenará el valor `-1`.

Las operaciones que se pueden especificar con el argumento `cmd` de `msgctl` son:

- **IPC_RMID.** Borra del sistema la cola de mensajes identificada por `msqid`. Si la cola está siendo usada por otros procesos, la eliminación de la cola no se hace efectiva hasta que todos los procesos terminan de utilizarla.
- **IPC_STAT.** Lee el estado de la estructura `msg_perm` asociada a la entrada `msqid` de la tabla de colas y lo almacena en `buff`
- **IPC_SET.** Modifica el valor de los campos de la estructura `msg_perm` asociada a la entrada `msqid` de la tabla de colas. Los nuevos valores para estos campos los toma de `buff`.

En la estructura `msg_perm` los campos modificables por el usuario son: `msg_perm.uid`, `msg_perm.gid` y `msg_perm.mode`. Mientras que, el superusuario puede modificar el campo `msg_qbytes`. Los demás campos o no son modificables o son manipulados por el sistema directamente.

◆ Ejemplo 7.14:

La llamada al sistema

```
msgctl(msqid, IPC_RMID, 0);
```

borra la cola de mensajes con identificador `msqid`.

◆

7.3.3.6 Discusión

Las colas de mensajes suministran servicios similares a las tuberías. Sin embargo las colas de mensajes son más versátiles y no poseen las limitaciones de las tuberías. Las colas de mensajes transmiten datos como mensajes discretos, a diferencia de las tuberías que transmiten datos como un flujo de bytes sin formato. Esto permite un mejor procesamiento de los datos. El campo *tipo de mensaje* de los mensajes permite asociar prioridades a los mensajes, lo que posibilita a un proceso receptor el poder comprobar antes los mensajes más urgentes. Asimismo en escenarios donde una cola de mensajes es compartida por múltiples procesos, el campo *tipo de mensaje* puede ser utilizado para designar un receptor.

Las colas de mensajes son útiles para transferir pequeñas cantidades de datos. Sin embargo si hay que transferir grandes cantidades de datos el rendimiento del sistema se deteriora. Esto es debido a que la transferencia de un mensaje requiere de dos operaciones de copia de datos en memoria: la primera del espacio de direcciones del proceso emisor a un buffer interno del núcleo, y la segunda de dicho buffer al espacio de direcciones del proceso receptor.

Otra limitación de las colas de mensajes es que no pueden especificar un determinado receptor. Cualquier proceso con los permisos apropiados puede recuperar mensajes de la cola. Aunque, como se mencionó con anterioridad, procesos cooperantes pueden acordar un protocolo para especificar receptores. Finalmente, otra limitación de las colas de mensajes es que no suministran un mecanismo de difusión, es decir, un proceso no puede enviar un único mensaje a varios receptores.

Debido a las limitaciones de las colas de mensajes, la mayoría de las aplicaciones de los sistemas UNIX más modernos encuentran en el uso de los *streams* un mecanismo más potente para implementar el paso de mensajes.

7.3.4 Memoria Compartida

La forma más rápida de comunicar dos procesos es hacer que compartan una zona de memoria. Para enviar datos de un proceso a otro, el proceso emisor solamente tiene que escribir en memoria y automáticamente esos datos estarán disponibles para que los lea otro proceso.

Es conocido que la memoria convencional que puede direccionar un proceso a través de su espacio de direcciones virtuales es un espacio local a dicho proceso y cualquier intento de direccionar esa memoria desde otro proceso va a provocar una violación de segmento.

El sistema UNIX System V soluciona este problema permitiendo crear regiones de memoria virtual que pueden ser direccionadas por varios procesos simultáneamente.

7.3.4.1 Estructuras de datos utilizadas para compartir memoria

El núcleo posee una *tabla de memoria compartida*, cada entrada en dicha tabla está asignada a una región de memoria compartida que viene identificada por un descriptor numérico `shmid`. Además, cada entrada contiene una estructura del tipo `shmid_ds`, que se define de la siguiente forma:

```
struct shmid_ds{  
    struct ipc_perm shm_perm;    → Estructura que mantiene los permisos  
    int shm_segsz;               → Tamaño del segmento  
    ushort shm_lpid;             → pid del proceso que realizó la ultima operación sobre  
                                la región de memoria compartida  
    ushort shm_cpid;             → pid del proceso creador  
    ushort shm_nattch;           → Número de procesos unidos a la región de memoria  
                                compartida  
    time_t shm_atime;            → Fecha de la última conexión  
    time_t shm_dtime;            → Fecha de la última desconexión  
    time_t shm_ctime;            → Fecha de la última operación shmctl  
};
```

7.3.4.2 Creación u obtención de una región de memoria compartida

Para crear un segmento de memoria compartida o acceder a uno que ya existe, se utiliza la llamada al sistema `shmget`, cuya sintaxis es:

```
shmid=shmget(key, size, flags);
```

donde `key` es la clave de acceso a un segmento de memoria compartida, `size` especifica el tamaño en bytes del segmento de memoria solicitado y `flags` es una máscara de indicadores (similar a la descrita para los semáforos). Si la llamada al sistema `shmget` se ejecuta con éxito entonces en `shmid` se almacenará el identificador entero de la zona de memoria compartida asociada a la llave `key`. En caso contrario en `shmid` se almacenará el valor -1.

El identificador devuelto por `shmget` es heredado por los procesos descendientes del actual.

Cuando un proceso realiza la llamada al sistema `shmget` el núcleo realiza las siguientes acciones:

- 1) Busca en la *tabla de memoria compartida* la región asociada con el parámetro `key`, si encuentra dicha región y el proceso tiene los permisos de acceso correctos entonces devuelve el descriptor `shmid`.
- 2) Si no encuentra la región asociada con el parámetro `key` y el usuario ha configurado el indicador `IPC_CREAT` de `flags` para crear una nueva región entonces:
 - 2.1) Comprueba que el tamaño especificado `size` se encuentra entre los límites mínimo y máximo permitidos
 - 2.2) Asigna una región mediante el uso del algoritmo `allocreg()`.
 - 2.3) Salva los permisos, tamaño y un puntero a la *tabla de regiones* dentro de la estructura `shmid_ds` asociada a la entrada `shmid` de la *tabla de memoria compartida*.

2.4) Activa un bit en la entrada de la *tabla de regiones* asignada a la región de memoria compartida `shmid` para identificarla como una región de memoria compartida.

2.5) También en la entrada de la *tabla de regiones* asignada a la región de memoria compartida `shmid` el núcleo activa un bit para indicar que dicha región no debe ser liberada cuando el último proceso que la comparta termine. Por lo tanto, los datos en una región de memoria compartida permanecerán intactos incluso aunque ningún proceso comparta ya dicha región.

◆ Ejemplo 7.15

Las siguientes líneas de código C muestran como crear una zona de memoria compartida de tamaño 4096 bytes, sólo el usuario va a tener permisos de lectura y escritura.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
...
int shmid;
...
shmid=shmget(IPC_PRIVATE,4096,IPC_CREAT | 0600);

if (shmid==-1)
{
/* Error en la creación de la memoria compartida.
   Tratamiento del error.*/
}
```

◆

7.3.4.3 Ligar una región de memoria compartida al espacio de direcciones virtuales de un proceso

Antes de que un proceso pueda usar la región de memoria compartida `shmid`, es necesario asignarle un espacio de direcciones virtuales de dicho proceso. Esto es lo que se conoce como *unirse o enlazarse al segmento de memoria compartida*.

La llamada `shmat` asigna un espacio de direcciones virtuales al segmento de memoria cuyo identificador `shmid` ha sido dado por `shmget`. Por lo tanto `shmat` enlaza

una región de memoria compartida de la tabla de regiones con el espacio de direcciones de un proceso (ver Figura 7.4)

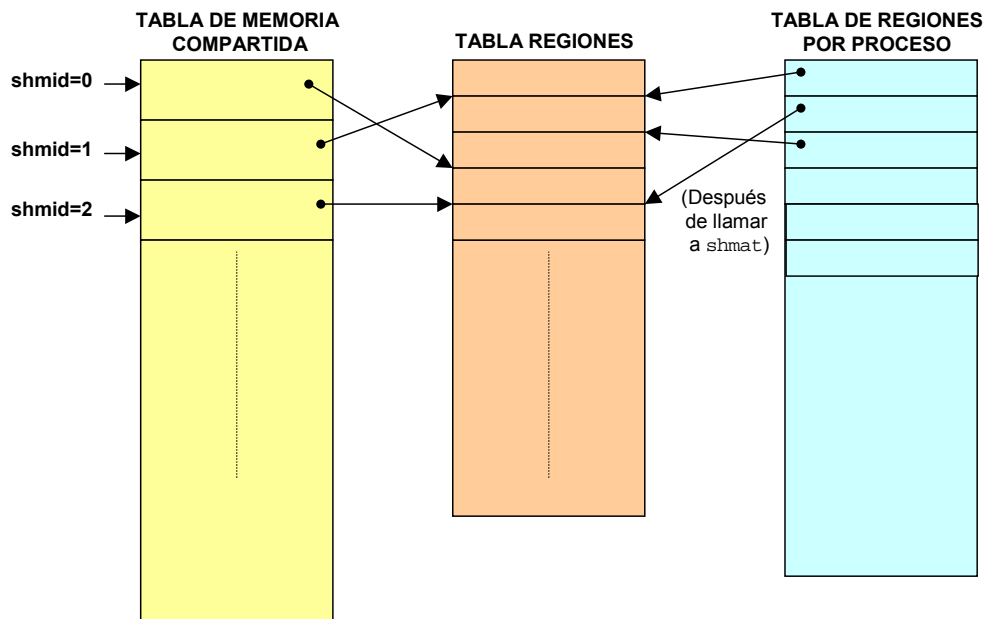


Figura 7.4: Estructura de datos para el mecanismo IPC de memoria compartida, una vez realizada la llamada al sistema `shmat`.

La llamada al sistema `shmat` tiene la siguiente sintaxis:

```
resultado=shmat(shmid,shmdir,shmflags);
```

donde `shmid` es un identificador de una región de memoria compartida, `shmdir` es la dirección virtual del proceso donde se desea que empiece la región de memoria compartida, `shmflags`, es una máscara de bits que indica la forma de acceso a la memoria. Si el bit `SHM_RDONLY` está activo, la memoria será accesible para leer, pero no para escribir. Por defecto un segmento de memoria se comparte para lectura y escritura. Si la llamada al sistema `shmat` tiene éxito en `resultado` se almacena la dirección a la que está unido el segmento de memoria compartida `shmid`. En caso contrario en `resultado` se almacena el valor -1.

Las reglas que utiliza `shmat` para determinar la dirección son:

- Si `shmdir = 0`, el sistema selecciona la dirección.
- Si `shmdir ≠ 0`, el valor de la dirección devuelto depende si se especificó o no el bit `SHM_RND` del parámetro `shmflags`. Si se especificó el segmento de

memoria es enlazada en la dirección especificada por el parámetro `shmdir` redondeada por la constante `SHMLBA` (SHare Memory Lower Boundary Address). En caso contrario el segmento de memoria es enlazado en la dirección especificada por el parámetro `shmdir`

Obviamente para conseguir portabilidad lo mejor es dejar que el núcleo asigne la dirección (`shmdir=0`)

En el momento que una región de memoria compartida `shmid` se une a un proceso, está pasa a formar del espacio de direcciones virtuales de dicho proceso, siendo por tanto accesible de la misma forma (mediante el uso de punteros) que las restantes direcciones virtuales. Luego no es necesario invocar a ninguna llamada al sistema especial para acceder a los datos almacenados en un segmento de memoria compartida.

7.3.4.4 Desligar una región de memoria compartida del espacio de direcciones virtuales de un proceso

Cuando un proceso ha terminado de usar un segmento de memoria compartida `shmid` entonces debe desenlazarse o desunirse de él, para conseguirlo utiliza la llamada al sistema `shmdt`. Su sintaxis es:

```
resultado=shmdt(shmdir);
```

donde `shmdir` es la dirección virtual del segmento de memoria compartida que se quiere separar del proceso. Si la llamada tiene éxito en `resultado` se almacena el valor 0. En caso contrario se almacena el valor -1.

7.3.4.5 Acceso a la información administrativa y de control de una región de memoria compartida

La llamada al sistema `shmctl` permite realizar operaciones de control sobre una zona de memoria compartida creada previamente por `shmget`. Su sintaxis es:

```
resultado=shmctl(shmid,cmd,&buffer);
```

donde `shmid` es el identificador de una región de memoria compartida, `cmd` es un número entero o una constante simbólica (ver Tabla 7.2) que especifica la operación a efectuar, y `buffer` es una estructura del tipo predefinido `shmid_ds` que contiene los argumentos de la operación. Si la llamada `shmctl` tiene éxito, en `resultado` se almacenará un número entero cuyo valor depende del comando `cmd`. Si falla en `resultado` se almacenará el valor -1.

Valores	Significado
IPC_STAT	Lee el estado de la estructura de control <code>shm_perm</code> de la memoria compartida y lo devuelve en la zona apuntada por <code>buffer</code> .
IPC_SET	Inicializa alguno de los campos de la estructura de control de la memoria compartida <code>shm_perm</code> . El nuevo valor para estos campos los toma de la estructura apuntada por <code>buffer</code> .
IPC_RMID	Borra del sistema la región de memoria compartida identificada por <code>shmid</code> . Si existen varios procesos compartiendo la zona de memoria el borrado no se realiza hasta que todos los procesos liberen la memoria.
SHM_LOCK	Bloquea el segmento identificado por <code>shmid</code> . Esto implica que no se puede intercambiar a memoria secundaria. Solo se permite esta operación si el identificador de usuario efectivo es igual al del superusuario.
SHM_UNLOCK	Desbloquea el segmento de memoria compartida <code>shmid</code> , permitiendo el intercambio con memoria secundaria. Solo se permite esta operación si el identificador de usuario efectivo es igual al del superusuario

Tabla 7.2: Valores posibles del parámetro `cmd` de la llamada `shmctl`

◆ **Ejemplo 7.16:**

Las siguientes líneas de código C muestran como crear una zona de memoria compartida en la que se va almacenar un array unidimensional de 20 números reales. Tras manipular dicho array, la zona de memoria compartida es eliminada.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define MAX 20
int shmid, i;
float *array;
key_t llave;
...
/* Creación de una llave.*/
llave=ftok("prueba",'K');

/* Petición de una zona de memoria compartida */
shmid=shmget(llave,MAX*sizeof(float),IPC_CREAT | 0600);

/* Unión de la zona de memoria compartida a nuestro espacio de direcciones virtuales. */
array=shmat(shmid,0,0);
```

```
/* Manipulación de la zona de memoria compartida */
for (i=0; i<MAX; i++){
    array[i]=i*i;
}
...
/* Separación de la zona de memoria compartida de nuestro espacio de
direcciones virtuales. */
shmdt(array);

/* Borrado de la zona de memoria compartida */
shmctl(shmid, IPC_RMID, 0);
```



TEMA 8

SISTEMAS DE ARCHIVOS EN UNIX

8.1 INTRODUCCION

Un sistema de ficheros permite realizar una abstracción de los dispositivos físicos de almacenamiento de la información para que sean tratados a nivel lógico, como una estructura de más alto nivel y más sencilla que la estructura de su arquitectura hardware particular.

Todas las versiones del UNIX System V, así como las versiones anteriores a BSD4.2 disponían de un único sistema de ficheros, ahora conocido como *sistema de ficheros System V (s5fs)*¹. Con la distribución BSD4.2 se introdujo un nuevo sistema de ficheros denominado *sistema de ficheros rápido (FFS)*², que suministraba mejores prestaciones y mayor funcionalidad que *s5fs*. Desde entonces, FFS fue ganando una amplia aceptación, de hecho fue incluido en distribuciones no BSD como SVR4.

Tanto *s5fs* como *FFS* resultaban adecuados para aplicaciones generales de tiempo compartido. Sin embargo, resultaban inadecuados para las necesidades de otros tipos de aplicaciones. Por ello fue necesario crear nuevos sistemas de ficheros que mejoraran el FFS y atendieran las necesidades de ciertas aplicaciones específicas. La mayoría de estos sistemas de ficheros usan técnicas sofisticadas que suministran un mejor comportamiento, y una mayor seguridad y disponibilidad.

Los sistemas de ficheros anteriormente comentados son *locales* puesto que almacenan y administran sus datos en dispositivos directamente conectados al sistema. La proliferación de redes de computadoras condujo a un incremento de la necesidad de poder compartir ficheros entre computadoras. Los sistemas de ficheros *distribuidos* permiten a un usuario acceder a ficheros que residen en máquinas remotas. Ejemplos de

¹ El acrónimo *s5fs* deriva del término inglés *System V file system*.

² El acrónimo *FFS* deriva del término inglés *Fast File System*.

sistemas de ficheros distribuidos son: NFS (Sun Microsystems's Network File System), RFS (AT&T Remote File Sharing) y AFS (Andrew File System).

Asimismo, surgió una creciente necesidad de que UNIX pudiera soportar sistemas de ficheros de otros sistemas operativos tales como MS-DOS. Esto permitiría a un sistema UNIX ejecutándose en una máquina poder acceder a ficheros en particiones MS-DOS de la misma máquina.

Puesto que el subsistema de archivos de UNIX solo podía soportar un único tipo de sistema de archivos, se hacía necesario disponer de un interfaz en el subsistema de archivos de UNIX que permitiera soportar múltiples tipos de sistema de ficheros: UNIX y no-UNIX, locales y distribuidos. Este objetivo se consiguió con el *interfaz nodo-v/sfv* desarrollado por Sun Microsystems que introdujo los conceptos de *nodo virtual (nodo-v)* y *sistema de ficheros virtual (sfv)*.

En este tema, antes de describir el interfaz nodo-v/sfv implementado en SVR4, se incluyen unas nociones básicas acerca de los ficheros especiales, el montaje de sistemas de ficheros, los enlaces simbólicos, y la *caché de buffers de bloques*. Finalmente, una vez descrito el interfaz nodo-v/sfv del SVR4, se estudia el sistema de ficheros *s5fs* debido a su importancia histórica y a su sencillo diseño. El estudio del *s5fs* también se realiza desde la perspectiva del SVR4.

8.2 FICHEROS ESPECIALES

Los *ficheros especiales* o *ficheros de dispositivos* permiten a los procesos comunicarse con los dispositivos periféricos. Los dispositivos periféricos pueden ser de dos tipos: *dispositivos modo bloque (discos, CD-ROM,...)* y *dispositivos modo carácter* (terminales, impresoras, ratón...). La principal diferencia entre ambos tipos de dispositivos es que para realizar la transferencia de datos con los dispositivos modo bloque el núcleo utiliza un área de almacenamiento en la memoria principal denominada *caché de buffers de bloques*. Usualmente, en el directorio `/dev` se suelen almacenar todos los ficheros de dispositivos.

El sistema también puede soportar dispositivos software (o pseudodispositivos) que no tienen asociados un dispositivo físico. Por ejemplo, si una parte de la memoria del sistema se gestiona como un dispositivo, los procesos que quieran acceder a esa zona de memoria tendrán que usar las mismas llamadas al sistema que existen para el manejo de ficheros ordinarios, pero sobre el fichero de dispositivo `/dev/mem` (fichero de

dispositivo genérico para acceder a memoria). En esta situación la memoria es tratada como un periférico más.

Una de las características distintivas del sistema de ficheros de UNIX es la generalización del concepto de *fichero* para incluir todo tipo de objetos relativos a E/S, tales como directorios, enlaces simbólicos, dispositivos hardware, pseudodispositivos, y abstracciones de comunicación como las tuberías o los conectores. Cada uno de ellos es accedido a través de descriptores de ficheros, y el mismo conjunto de llamadas al sistema que operan sobre los ficheros ordinarios también manipulan estos objetos de E/S. Por ejemplo, un usuario puede enviar datos a una impresora en línea simplemente abriendo el fichero especial asociado con ella y escribiéndolo.

Sin embargo, algunos objetos de E/S no soportan todas las operaciones de ficheros. Por ejemplo, los terminales y las impresoras, no tienen noción de acceso aleatorio o búsquedas. Las aplicaciones a menudo necesitan verificar (típicamente a través de la llamada al sistema `fstat`) a que tipo de fichero están accediendo.

Los ficheros de dispositivos, al igual que el resto de ficheros, tienen asociado un nodo-i. En el caso de los ficheros ordinarios o los directorios, este nodo-i contiene, entre otras informaciones, donde se encuentran los bloques de datos del fichero. Pero en el caso de los ficheros de dispositivo no hay datos a los que referenciar. En su lugar, el nodo-i contiene dos números conocidos como *número principal* (major number) y *número secundario* (minor number). El *número principal* indica el tipo de dispositivo de que se trata (disco, cinta, terminal, etc). El *número secundario* indica el número de unidad dentro del dispositivo, es decir, la instancia específica del dispositivo.

Por ejemplo, todos los discos duros pueden tener un número principal igual a 5, y cada disco duro existente tendrá un número secundario diferente. Por otra parte, los dispositivos de modo bloque y los dispositivos de modo carácter tienen conjuntos independientes de números principales. Así un número principal igual a 5 para dispositivos en modo bloque puede referirse a una unidad de disco, mientras que para dispositivos de modo carácter puede referirse a una impresora en línea.

El núcleo mantiene dos tablas, *la tabla de conmutación de dispositivos modo bloque* y *la tabla de conmutación de dispositivos modo carácter*. Cada entrada de una de estas tablas contiene una estructura cuyos miembros son unos punteros a una colección de rutinas que permiten manejar a un dispositivo. Esta colección de rutinas constituyen realmente el *driver* o *manejador* del dispositivo.

Cuando un usuario invoca a una llamada al sistema para realizar una operación de E/S (supóngase que se realiza una llamada `read`) sobre un fichero especial, el núcleo realiza las siguientes acciones:

- 1) Usa el descriptor del fichero para localizar el objeto de fichero abierto.
- 2) Comprueba en el objeto de fichero abierto que el fichero ha sido abierto en un modo tal que permite ser leído.
- 3) Obtiene en el objeto de fichero abierto el puntero al nodo-i en memoria (nodo-im) para esta entrada. Un nodo-im es una estructura de datos del núcleo que duplica la información almacenada en un nodo-i de un disco y que además contiene ciertas informaciones adicionales asociada al fichero activo en memoria.
- 4) Bloquea el nodo-im para asegurarse temporalmente la exclusividad de acceso al fichero.
- 5) Comprueba el campo *modo* del nodo-im para encontrar el tipo del fichero. Supóngase que es un fichero de dispositivo de modo carácter.
- 6) Utiliza el *número principal* y el número secundario (almacenados en el nodo-im) como índice en la *tabla de conmutación de dispositivos modo carácter* para localizar la estructura que contiene los punteros a las rutinas que constituyen el manejador del dispositivo. Supóngase que dicha estructura se denomina `cdevsw` y que su definición es:

```
struct cdevsw{  
    int (*d_open)();  
    int (*d_close)();  
    int (*d_read)();  
    int (*d_write)();  
    ...  
}cdevsw[];
```

Los campos de la estructura `cdevsw` tales como `d_read` definen un interfaz abstracto. Cada dispositivo lo implementa a través de funciones específicas, por ejemplo, `lpread()` para una impresora en línea o `ttread()` para un terminal.

- 1) De `cdevsw`, obtiene el puntero a la rutina que implementa la operación de lectura (`d_read`) para este dispositivo.
- 2) Invoca a la rutina `d_read` para realizar la operación de lectura sobre el dispositivo.
- 3) Desbloquea el nodo-im y devuelve el resultado al usuario.

Se observa que muchos de estos pasos son independientes del dispositivo. Los pasos 1 al 4 y el paso 9 se pueden aplicar tanto a los ficheros ordinarios como a los ficheros de dispositivos. Por lo tanto este conjunto de pasos son independientes del tipo de fichero. Los pasos 5 al 7 representan el interfaz entre el núcleo y los dispositivos, que se encuentra encapsulado en la estructura almacenada en una entrada de la *tabla de conmutación de dispositivos modo carácter* o de la *tabla de conmutación de dispositivos modo bloque*. Todo el procesamiento dependiente del dispositivo está localizado en el paso 8.

8.3 MONTAJE DE SISTEMAS DE FICHEROS

8.3.1 Consideraciones generales

Un *disco físico* es un dispositivo periférico para el almacenamiento permanente de datos. Se trata de un dispositivo modo bloque, que contiene un array de bloques de tamaño fijo. Cada uno de estos bloques posee un número identificativo denominado *número de bloque físico*.

Un *disco lógico* es una abstracción de almacenamiento que el núcleo ve como una secuencia lineal de bloques de tamaño fijo accesibles aleatoriamente. Cada bloque de un disco lógico tiene asignado un número identificativo denominado *número de bloque lógico*. El *driver o manejador del disco* entre otras tareas se encarga de traducir los números de bloques lógicos a números de bloques físicos.

En el caso más simple, un disco lógico se corresponde con un disco físico entero. Sin embargo, es usual dividir un disco físico en varias *particiones* físicas contiguas, cada una asociada a un disco lógico. Las *particiones* son, por lo tanto, divisiones del disco independientes unas de las otras y es responsabilidad del administrador del sistema decidir que va a contener cada una de ellas. Se denomina *partición activa* a aquella partición en la que se busca el sistema operativo en el momento del arranque de la máquina. Naturalmente, para que una partición sea autoarrancable, debe tener un sector de arranque y el archivo o archivos de arranque del sistema.

La existencia de diferentes particiones en un disco posibilita el que en un mismo disco físico coexistan varios sistemas operativos sin que interfieran unos con otros. Esto se consigue dedicando una partición de disco a cada uno de los sistemas.

Cada distribución de UNIX tiene una aplicación que permite crear la tabla de particiones, por ejemplo en algunas versiones de UNIX esta aplicación se denomina `fdisk`. Este programa presenta un menú que permite definir el tamaño dedicado a cada partición, visualizar el total de particiones que se han definido, definir una partición como partición activa, etc.

Una vez establecidas las particiones del disco, se pueden instalar sistemas de ficheros sobre ellas. Ciertas utilidades de usuario como `newfs` o `mkfs` permiten crear un sistema de ficheros UNIX en un disco físico. Solamente dispositivos de modo bloque pueden alojar un sistema de ficheros UNIX. Cada sistema de ficheros se encuentra contenido por completo en un único disco lógico, y un disco lógico puede contener un único sistema de ficheros. Algunos discos lógicos en lugar de contener un sistemas de ficheros son usados por el subsistema de memoria como un *área de intercambio*, para el almacenamiento temporal de procesos completos o de páginas de procesos.

En los sistemas UNIX más antiguos cada partición física estaba asociada a un disco lógico. Por ello, la palabra partición es a menudo utilizada para describir el almacenamiento físico de un sistema de ficheros. Los sistemas UNIX más modernos soportan otras configuraciones de almacenamiento. Por ejemplo, varios discos físicos pueden ser combinados en un único disco lógico o volumen, soportando así ficheros de tamaño más grande que el tamaño de un único disco.

Aunque la jerarquía de ficheros de UNIX parece monolítica, se puede tener varios subárboles independientes, cada uno de los cuales puede contener un sistema de ficheros completo. Un sistema de ficheros se configura para ser el *sistema de ficheros raíz*, y para que su directorio raíz sea el *directorio raíz del sistema*. Los otros sistemas de ficheros son adjuntados a la nueva estructura montando cada nuevo sistema de ficheros dentro de un directorio del árbol ya existente, al que se le denominará *directorio de montaje* o *punto de montaje*.

Una vez montado, el directorio raíz del sistema de ficheros montado cubre u oculta el directorio en el cual es montado, y cualquier acceso al directorio de montaje es traducido a un acceso al directorio raíz del sistema de ficheros montado. Obviamente, el sistema de ficheros montado permanece visible hasta que es desmontado.

♦ Ejemplo 8.1:

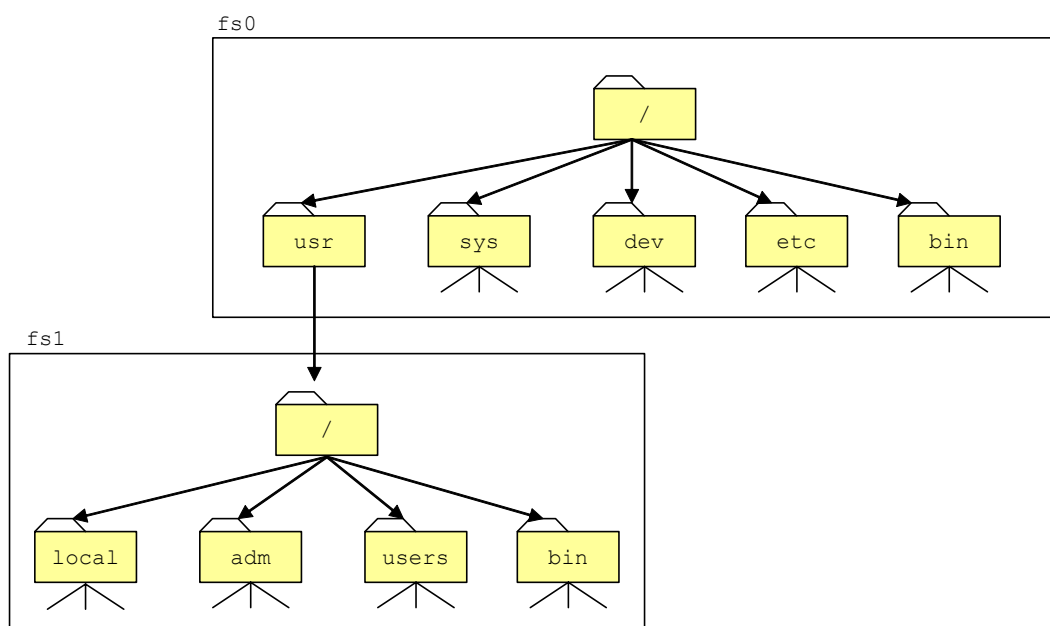


Figura 8.1: Montaje de un sistema de ficheros en otro

La Figura 8.1 muestra un árbol de ficheros compuesto de dos sistemas de ficheros. En este ejemplo, `fs0` está instalado como el sistema de ficheros raíz de la máquina, y el sistema de ficheros `fs1` está montado en el directorio `/usr` de `fs0`. A este directorio se le denomina *directorio de montaje* o *punto de montaje*, cualquier intento de acceder a `/usr` resulta en un acceso al directorio raíz del sistema `fs1` montado en él.

Si el directorio `/usr` de `fs0` contiene cualquier fichero, estos son ocultos cuando `fs1` es montado en él, y quizás ya no son accesibles para el usuario. Cuando `fs1` es desmontado, estos ficheros se hacen visibles y son accesibles de nuevo.

♦

La noción de *sistemas de ficheros montados* permite ocultar al usuario los detalles de la organización del almacenamiento. El espacio de nombres de ficheros es homogéneo, y el usuario no necesita especificar la unidad de disco como parte del nombre del fichero (como si es necesario especificar en otros sistemas operativos). Asimismo, cada sistema de ficheros montado pueden ser considerado de forma individual para realizar copias de seguridad o realizar tareas de compactación o reparación. Además, el administrador del sistema puede variar independientemente las protecciones de cada sistema de ficheros montado.

Normalmente, los sistemas de ficheros que se utilizan no se están cambiando frecuentemente, por eso el núcleo utiliza una *tabla de montaje* para identificar a los sistemas de ficheros que debe montar al arrancar la máquina, y desmontar al apagarla. Dicha tabla suele ubicarse en el fichero `/etc/mtab`. En este fichero aparecen varias líneas, cada línea da información sobre un sistema de ficheros montado.

♦ **Ejemplo 8.2:**

En el caso de un sistema Linux la tabla de montaje se suele ubicar en `/etc/fstab`. A continuación se muestra, a modo de ejemplo, un posible contenido de este archivo:

# device	directory	type	options
/dev/hda1	/	ext2	defaults
/dev/hda2	/usr	ext2	defaults
/dev/hda3	none	swap	sw
/dev/sda1	/dos	msdos	defaults
/proc	/proc	proc	none

La primera línea es una línea de comentarios para especificar el significado de cada columna: dispositivo que se monta (*device*), directorio de montaje (*directory*), tipo de sistema de ficheros montado (*type*) y opciones de montaje (*options*).

Así, la segunda línea indica que la primera partición del disco duro (`/dev/hda1`) tiene como punto de montaje el directorio raíz (`/`), el tipo de sistema de ficheros montado es `ext2` (Segundo sistema de archivos extendido (*ext2fs*)), uno de los estándar de Linux. Las opciones de montaje han sido las establecidas por defecto (*default*), entre las que se encuentran:

- El sistema de archivos se monta con permisos de lectura/escritura.
- El sistema de archivos se considera como un dispositivo modo bloque.
- Todas las E/S de archivo deberían hacerse asíncronamente.
- Se permite la ejecución de ficheros ejecutables.
- Se interpretan los bits `S_ISUID` y `S_ISGID` de los archivos.
- Los usuarios normales no pueden montar el sistema de archivos.

La tercera línea indica que la segunda partición del disco duro (`/dev/hda2`) tiene como punto de montaje el directorio `/usr`, el tipo de sistema de ficheros montado es `ext2`. Las opciones de montaje han sido las establecidas por defecto.

La cuarta línea indica que la tercera partición del disco duro (`/dev/hda3`) se utiliza como área de intercambio. Su punto de montaje se especifica como `none` porque no se desea que aparezca en el árbol de directorios. Las áreas de intercambio se montan con la opción `sw` y con el tipo `swap`.

La quinta línea indica que la primera partición de un disco duro SCSI (`/dev/sda1`) tiene como punto de montaje el directorio `/dos`, el tipo de sistema de ficheros montado es `msdos`, es decir, MS-DOS. Las opciones de montaje han sido las establecidas por defecto.

Finalmente, la última línea está asociada a `/proc` que es un sistema de ficheros especial que suministra un interfaz elegante y potente con el espacio de direcciones de cualquier proceso. Fue inicialmente diseñado en SVR4 como una utilidad para soportar a los procesos depuradores, y sustituir a `ptrace`, pero ha ido evolucionando hasta convertirse en un interfaz general al modelo de procesos. Permite a un usuario leer y modificar el espacio de direcciones de otro proceso y realizar varias tareas de control sobre él, utilizando el interfaz de sistema de ficheros y las llamadas al sistema estándar. Cada proceso es representado como un subdirectorio de `/proc`, y el nombre de este subdirectorio es el `pid` del proceso. A su vez, cada subdirectorio contiene diferentes ficheros y subdirectorios con información de control sobre el proceso. Estos subdirectorios y ficheros no ocupan espacio en ninguna partición física de disco.



Los sistemas de ficheros montados imponen algunas restricciones en la jerarquía de ficheros. Así un fichero perteneciente a un cierto sistema de ficheros puede aumentar su tamaño en función del espacio libre que exista en dicho sistema de ficheros. Asimismo el cambio de nombre y las operaciones sobre los enlaces duros de un fichero están también limitadas al sistema de ficheros al que pertenece. Además cada sistema de ficheros debe residir en un único *disco lógico* y está limitado por el tamaño de este disco.

8.3.2 Llamadas al sistema y comandos asociados al montaje de sistema de ficheros

La llamada al sistema `mount` permite montar un sistema de ficheros desde un programa. Su sintaxis, en los sistemas UNIX clásicos, es:

```
resultado = mount(dispositivo, dir, flags);
```

donde `dispositivo` es la ruta de acceso del fichero del dispositivo del disco donde se encuentra el sistema de ficheros que se va a montar, `dir` es la ruta de acceso del directorio sobre el que se va a montar el sistema de ficheros, y `flags` es una máscara de bits que permite especificar diferentes opciones. En concreto el bit menos significativo de `flags` se utiliza para revisar los accesos de escritura sobre el sistema de ficheros. Si vale 1, la escritura estará prohibida, por lo que sólo se podrán hacer accesos de lectura; en caso contrario, la escritura estará permitida, pero de acuerdo a los permisos individuales de cada fichero.

Si la llamada se ejecuta con éxito en `resultado` se almacena el valor 0. En caso contrario, se almacena el valor -1.

La implementación del interfaz nodo-v/sfv tuvo que modificar la llamada al sistema `mount` para soportar la existencia de múltiples tipos de sistemas de ficheros. Así, su sintaxis en el SVR4 es:

```
resultado = mount(dispositivo, dir, flags, tipo, dataptr, datalon);
```

donde `tipo` es una array de caracteres que especifica el tipo del sistema de ficheros, `dataptr` es un puntero a argumentos adicionales dependientes del sistema de ficheros, y `datalon` es el tamaño total de estos parámetros extra.

Cuando un sistema de ficheros deja de ser utilizado, puede ser desmontado. La llamada para llevar a cabo esta acción es `umount`, su sintaxis es:

```
resultado = umount(dispositivo);
```

donde `dispositivo` es la ruta de acceso del fichero del dispositivo que da acceso al sistema de ficheros que se desea desmontar.

Las llamadas `mount` y `umount` no actualizan el fichero `/etc/mtab`, que contiene la tabla de montaje. Por lo tanto si se decide montar un sistema de ficheros desde un programa, habrá que actualizar también desde dicho programa el fichero `/etc/mtab`.

Por otra parte, también es posible montar un sistema de ficheros desde la línea de ordenes usando el comando `mount`, cuya sintaxis más usual es:

```
mount <dispositivo> <dir>
```

donde `<dispositivo>` es la ruta de acceso del fichero del dispositivo del disco donde se encuentra el sistema de ficheros que se va a montar y `<dir>` es la ruta de acceso del directorio sobre el que se va a montar el sistema de ficheros.

Asimismo para desmontar un sistema de archivos se puede usar el comando `umount`, cuya sintaxis más usual es:

```
umount <dispositivo>
```

En principio, los comandos `mount` y `umount` sólo pueden ser utilizados por el superusuario; aunque cualquier usuario puede invocar la orden `mount` sin argumentos, que muestra por pantalla el contenido del fichero `/etc/mtab`.

♦ Ejemplo 8.3:

La llamada al sistema

```
mount("/dev/hda2","/usr",0);
```

monta la partición 2 del disco duro sobre el directorio `/usr`, el sistema se monta en modo lectura/escritura.

La llamada al sistema

```
umount("/dev/hda2");
```

desmonta la partición 2 del disco duro.

La orden

```
# mount /dev/fd0 /mnt/floppy
```

monta en el directorio o punto de montaje `/mnt/floppy` el sistema de archivos asociado al dispositivo físico `/dev/fd0`, es decir, a la disquetera de discos 3.5".

La orden

```
# umount /dev/fd0
```

desmonta el sistema de archivos asociado al dispositivo físico `/dev/fd0`.

Finalmente, la orden:

```
# mount /dev/hdc /mnt/cdrom
```

monta en el directorio o punto de montaje `/mnt/cdrom` el sistema de archivos asociado al dispositivo físico `/dev/hdc`, es decir, al CD-ROM.

♦

8.4 ENLACES SIMBOLICOS

En UNIX un mismo fichero puede tener diferentes nombres, cada uno de estos nombres constituye un *enlace duro* al fichero. Un enlace duro apunta al nodo-i del fichero. Aunque extremadamente útiles los enlaces duros poseen ciertas limitaciones. Es

imposible crear enlaces duros a través de distintos sistemas de ficheros. Además los enlaces duros solamente pueden apuntar a ficheros, para prevenir la aparición de ciclos en el árbol de directorios no es posible crear enlaces duros a un directorio. Asimismo, los enlaces duros también presentan algunos problemas de control.

♦ **Ejemplo 8.4:**

Un ejemplo de problema de control de los enlaces duros se pone de manifiesto en el siguiente escenario: supóngase que el usuario *X* posee un fichero llamado `/usr/X/fichero1`. Otro usuario *Y* puede crear un enlace duro a este fichero y llamarlo `/usr/Y/link1` (ver Figura 8.2). Para hacer esto, *Y* sólo necesita tener permiso de ejecución para los directorios de la ruta de acceso y permiso de escritura para el directorio `/usr/Y/`. Posteriormente, el usuario *X* puede desenlazar `fichero1` y creer que el fichero ha sido borrado (típicamente, los usuarios no comprueban los contadores de enlaces de sus ficheros). El fichero, sin embargo, continua existiendo debido al otro enlace.

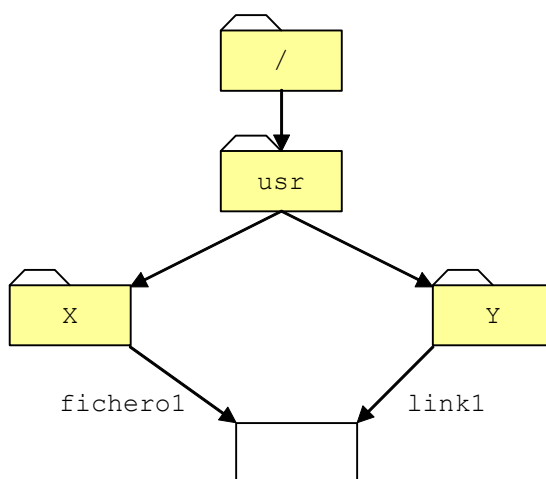


Figura 8.2: Enlaces duros (fichero1 y link1) de un fichero.

Por supuesto, `/usr/Y/link1` es todavía propiedad del usuario *X*, aunque el enlace fuese creado por *Y*. Si *X* ha protegido contra escritura al fichero, entonces *Y* no podrá modificarlo. Sin embargo, *X* puede no desear que el fichero continúe existiendo. En sistemas que imponen cuotas de uso del disco duro, el espacio ocupado por el fichero continuará siendo cargado a *X*. Además, no hay forma de que *X* pueda descubrir la localización del enlace, en particular si *Y* tiene protegido contra escritura el directorio `/usr/Y` (o si *X* ya no conoce el número de nodo-*i* del fichero).

♦

BSD4.2 introdujo los *enlaces simbólicos* para solventar muchas de las limitaciones de los enlaces duros. Estos fueron pronto adoptados por la mayoría de las distribuciones

y SVR4 lo incorporó dentro de `s5fs`. Un *enlace simbólico* es un fichero especial que apunta a otro fichero (el fichero al que se enlaza). El atributo tipo de fichero lo identifica como un enlace simbólico. La porción de datos del fichero contiene la ruta (absoluta o relativa del fichero al que se enlaza). Muchos sistemas permiten que pequeñas rutas sean almacenadas en el nodo-i del enlace simbólico.

Cuando el núcleo encuentra un enlace simbólico durante el análisis de una ruta de acceso a un fichero, reemplaza el nombre del enlace por su contenido, y continua con el análisis de la ruta.

Por convenio la máscara de modo simbólica de cualquier enlace simbólico siempre es `lrwxrwxrwx`. Sin embargo, esta máscara simplemente es una notación, no tiene el significado usual ya que no se usa para determinar los permisos de acceso al enlace simbólico; estos son determinados por la máscara de modo del fichero apuntado por el enlace. Asimismo el uso del comando `chmod` sobre un enlace simbólico en realidad estaría especificando los permisos del fichero al que apunta el enlace.

Los enlaces simbólicos son muy útiles porque no tienen las limitaciones asociadas a los enlaces estrictos. Como un enlace simbólico no apunta a un nodo-i, es posible crear enlaces simbólicos a través de distintos sistemas de archivos. Además, los enlaces simbólicos pueden apuntar a cualquier tipo de archivo, incluso a archivos inexistentes.

Por otro lado, los enlaces simbólicos también presentan algunos inconvenientes. En primer lugar ocupan espacio en disco, dado que alojan sus nodos-i y sus bloques de datos. Asimismo la existencia de enlaces simbólicos en una ruta de acceso ralentiza su análisis.

El comando `ln` permite crear tanto enlaces duros como enlaces simbólicos a un fichero. Para crear un enlace duro su sintaxis es:

```
ln <fichero> <enlace>
```

donde `<fichero>` es la ruta de acceso al fichero al que se desea crear el enlace y `<enlace>` es el nombre que se desea dar al enlace. Si se desea crear un enlace simbólico, la sintaxis del comando es:

```
ln -s <fichero> <enlace>
```

◆ Ejemplo 8.5:

Supóngase que en el directorio de trabajo actual se tiene el fichero `prueba`. La orden

```
$ ls -i prueba
```

mostrará en la pantalla el mensaje

```
12500 prueba
```

Donde 12500 es el número de nodo-i asignado al fichero `prueba`. Para crear un enlace duro denominado `enlace` al fichero `prueba` se debe usar la orden

```
$ ln prueba enlace
```

La orden

```
$ ls -i prueba enlace
```

mostrará el siguiente mensaje en la pantalla

```
12500 enlace    12500 prueba
```

Es decir, cuando se accede a `prueba` o a `enlace` se accede al nodo-i número 12500, cuyo contador de referencias o enlaces duros contendrá el valor 2. Por tanto si se hacen cambios en `prueba`, estos cambios también serán efectuados en `enlace`. A todos los efectos, `prueba` y `enlace` son el mismo fichero. El valor de este contador se puede visualizar usando la orden

```
$ ls -l prueba enlace
```

que mostraría el siguiente mensaje en la pantalla

```
-rw-r--r--  2 ALUMNO    users 2 512 Aug  15 15:31  enlace
-rw-r--r--  2 ALUMNO    users 2 512 Aug  15 15:30  prueba
```

El número 2 después de la máscara de modo simbólica es precisamente el contador de enlaces duros (`enlace` y `prueba`) del nodo-i 12500.

◆

◆ Ejemplo 8.6:

Supóngase que en el directorio de trabajo actual se tiene el fichero `prueba2`. La orden

```
$ ln -s prueba2 enlace2
```

crea el enlace simbólico `enlace2` apuntando al fichero `prueba2`. Si se escribe la orden

```
$ ls -i prueba2 enlace2
```

muestra en la pantalla el mensaje

```
13500 enlace2      23500 prueba2
```

lo que pone de manifiesto que los dos ficheros tienen nodos-i diferentes.

Asimismo, la orden

```
$ ls -l prueba2 enlace2
```

muestra en la pantalla el mensaje

```
lrwxrwxrwx  1 ALUMNO      users 3    4 Aug   15 26:51 enlace2 -> prueba2
-rw-r--r--  1 ALUMNO      users 2 124 Aug   15 26:50 prueba2
```

cuya primera línea indica que `enlace2` es un enlace simbólico apuntando a `prueba2`.

◆

8.5 LA CACHÉ DE BUFFERS DE BLOQUES

Las operaciones de E/S en disco son una de las principales causas de los cuellos de botella en cualquier sistema. El tiempo requerido para leer un bloque de 512 bytes de un disco es del orden de unos pocos milisegundos. El tiempo para copiar la misma cantidad de datos de una posición a otra de memoria principal es del orden de unos pocos microsegundos. Los dos difieren en un factor de 1000. Si cada operación de E/S requiriera un acceso a disco, el sistema sería muy lento. Por lo tanto, es necesario minimizar las operaciones de E/S en disco. UNIX consiguió este objetivo implementando, via software, una memoria caché en un área de memoria principal para almacenar los bloques de disco accedidos recientemente en el sistema de ficheros. A esta caché se le denomina *caché de buffers de bloques*.

Los sistemas UNIX tradicionales usaban esta caché de buffers únicamente para almacenar bloques de disco. Los sistemas UNIX modernos tales como SVR4 y Sun OS (versión 4 o superiores) integran la *caché de buffers* con el sistema de paginación. En esta sección se describe la *caché de buffers* de los sistemas UNIX tradicionales tales como SVR3 o anteriores.

El tamaño de la *cache de buffers* es típicamente el 10% de la memoria principal. La *cache de buffers* está compuesta de buffers de datos usualmente de tamaño fijo (suficientemente grande para contener un bloque de disco).

Un buffer consta de dos partes: la *cabecera del buffer* y la *copia del bloque de disco* que almacena. La *cabecera del buffer* almacena información que permite identificar al buffer para poder realizar tareas de sincronización y administración de la caché.

La caché mantiene un conjunto de *colas de dispersión o colas hash* basadas en el número de dispositivo y en el número de bloque. El núcleo enlaza los buffers ubicados en una cierta cola de dispersión como una lista circular doblemente enlazada. Cada cola posee un buffer mudo a modo de cabecera para marcar el principio y el final de la lista. El número de buffers en una cola de dispersión varía durante el tiempo de vida del sistema. El núcleo debe usar una función de dispersión que distribuya los buffers uniformemente entre todas las colas de dispersión, además esta función debe ser sencilla para que no se vea afectado el rendimiento del sistema. Los administradores del sistema configuran el número de colas de dispersión de la caché de buffers cuando generan el sistema operativo.

♦ **Ejemplo 8.7:**

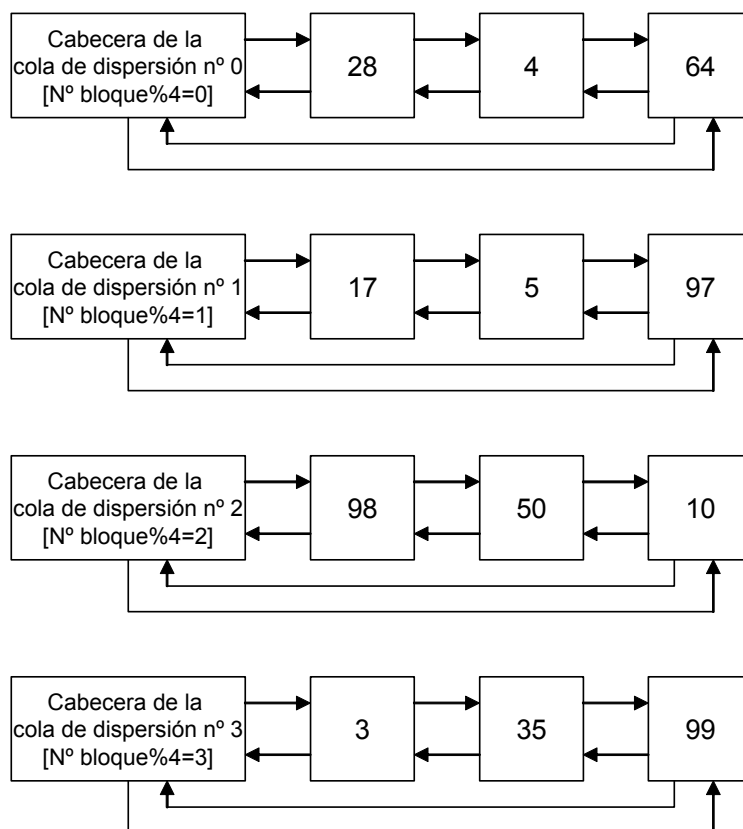


Figura 8.3: Caché de buffers

En la Figura 8.3 se representa un esquema de una caché de buffers de bloques. Esta caché consta de cuatro colas de dispersión y actualmente cada cola contiene tres buffers. La cola hash

nº 0 contiene los buffers marcados con los números 28, 4 y 64, que son los números de los bloques de disco que contienen, obsérvese que todos estos números cumplen la regla

$$N^{\circ} \text{ bloque} \% 4 = 0$$

es decir, al dividir el número de bloque por 4 su resto es 0. Se observa que las otras colas siguen reglas similares para los buffers que contienen.



El *almacenamiento de apoyo* de una caché es la posición permanente de los datos, cuyas copias son almacenadas en la caché. Una caché puede administrar datos de diferentes almacenamientos de apoyo. Para la caché de buffer de bloques, el almacenamiento de apoyo es el sistema de ficheros en disco. Si la máquina está conectada en red, el almacenamiento de apoyo incluye a los ficheros en los nodos remotos.

Generalmente, una caché puede soportar dos políticas de escritura: *inmediata* y *post-escritura*. La política de escritura inmediata consiste en que cuando hay que realizar una operación de escritura ésta se realiza tanto en la copia de los datos almacenados en la caché como en los datos originales situados en el almacenamiento de apoyo. De esta forma los datos en el almacenamiento de apoyo están siempre actualizados (excepto quizás por la última operación de escritura). Además no hay problemas de pérdida de datos o corrupción del sistema de ficheros en caso de que el sistema se cuelgue. También, la administración de la caché es más simple.

Todas estas ventajas, convierten a la política de escritura inmediata en una buena opción para cachés implementadas por hardware. Sin embargo, esta política no es apropiada para la caché de buffers de bloques, puesto que el rendimiento del sistema se ve seriamente afectado. Se estima que en la operación normal de un sistema cerca de un tercio de las operaciones de E/S son operaciones de escritura, y muchas de ellas son transitorias. Por ejemplo, la sobreescritura de un dato o el borrado del contenido de un fichero. Esto causaría muchas escrituras innecesarias, ralentizando al sistema tremendamente.

Por esta razón, la caché de buffer de UNIX utiliza una política de escritura del tipo *post-escritura*. Es decir, los bloques modificados son simplemente marcados como “sucios”, y son escritos al disco cuando los buffers que los contienen son seleccionados para ubicar otros bloques al no existir buffers libres en la caché. Esto permite a UNIX eliminar muchas de las escrituras y también reorganizar las escrituras de forma que se

optimice el rendimiento del disco. Retrasar las escrituras, sin embargo, supone un riesgo potencial de corrupción del sistema de ficheros en caso de que la máquina se cuelgue.

8.5.1 Funcionamiento básico

Cuando un proceso debe leer o escribir un bloque, el núcleo primero busca el bloque en la caché de buffers. Intenta localizar un buffer que tenga la combinación adecuada de número de dispositivo y número de bloque. Si no lo localiza, significará que el bloque no está en la caché. En dicho caso debe ser leído del disco (excepto cuando el bloque entero debe ser sobrescrito). Para ello, el núcleo escoge un buffer de la caché para almacenar dicho bloque e inicia una operación de lectura en disco.

Si el bloque es modificado por un proceso, el núcleo aplica las modificaciones a la copia almacenada en la caché de buffers y lo marca como “sucio” activando un indicador en la cabecera del buffer. Cuando un buffer que contiene un bloque “sucio” es seleccionado para ubicar a otro bloque al no existir buffers libres en la caché, se copia el contenido de dicho buffer en el disco antes de traer al otro bloque. Con ello se mantiene el contenido del disco actualizado.

Cuando un proceso obtiene un buffer lo bloquea para que no pueda ser utilizado por otros procesos. Esto sucede antes de iniciar la operación de E/S al disco o cuando el proceso desea leer o escribir en dicho buffer. Si un buffer ya está bloqueado por un proceso (A), y otro proceso (B) intenta acceder a él, el proceso B pasará al estado dormido hasta que el buffer sea desbloqueado por A. Puesto que el manipulador de las interrupciones del disco puede también intentar acceder al buffer, el núcleo desactiva las interrupciones del disco mientras está intentando adquirir el buffer.

El núcleo mantiene una *lista de buffers libres* usando una estrategia del tipo LRU³. Se trata de una lista circular doblemente enlazada que posee un buffer mudo a modo de cabecera para marcar el principio y el final de la lista. El buffer más cercano a la cabecera es el buffer libre usado menos recientemente, mientras que el buffer situado al final de la lista es el buffer libre usado más recientemente. Cuando el núcleo necesita un buffer libre, toma al buffer más cercano a la cabecera. Pero también puede tomar cualquier otro buffer de la lista si contiene el bloque que busca. En ambos casos, el núcleo borra de la lista de buffers libres el buffer que toma.

Cuando un buffer es liberado el núcleo lo coloca al final de la *lista de buffers libres*, puesto que en ese momento se trata del buffer usado más recientemente. Conforme el

núcleo va extrayendo buffers de la lista, dicho buffer avanzará hacia la cabecera de la lista. Inicialmente, cuando el sistema es arrancado, todos los buffers son colocados en la lista de buffers libres.

♦ **Ejemplo 8.8:**

En la Figura 8.4 se representa un esquema de una caché de buffers de bloques resaltando la lista de buffers libres. Mientras que en la Figura 8.5 se representa esta lista aparte. Se observa que forman parte de esta lista los buffers marcados con 3, 98, 5, 28 y 64. En esta lista el buffer libre usado menos recientemente es el más cercano a la cabecera, es decir, el 3. Asimismo el buffer libre usado más recientemente es el situado al final, es decir, el 64.

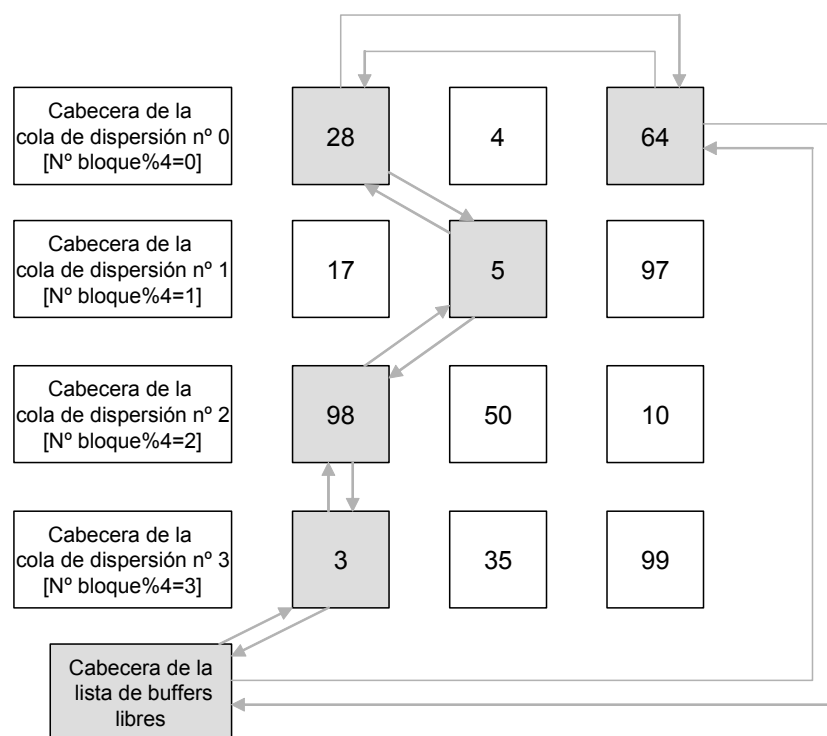


Figura 8.4: Implementación de la lista de buffers libres dentro de la caché de buffers

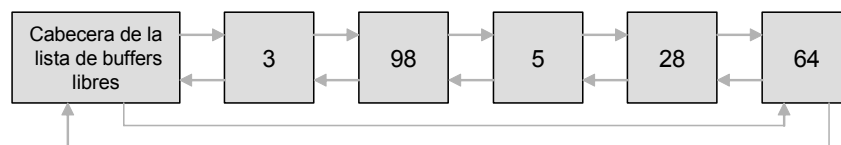


Figura 8.5: Detalle de la lista de buffers libres

♦

³ LRU es el acrónimo del término inglés “Least Recently Used”, que significa “usado menos recientemente”.

Existen dos excepciones en la gestión de la lista de buffers libres descrita. La primera involucra a los buffers que se han vuelto no válidos debido a un error de E/S o porque los bloques que almacenan pertenecen a un fichero que ha sido borrado o truncado. Tales buffers serán situados inmediatamente a la cabeza de la cola, puesto que está garantizado que no volverán a ser accedidos nuevamente.

La segunda excepción involucra a los buffers “sucios” que alcanzan la cabeza de la lista, en dicho instante son eliminados de la lista y colocados en la cola de escritura del manejador del disco. Cuando la escritura se completa, el buffer es marcado como “limpio” y puede ser retornado a la lista de buffers libres. Puesto que ya había alcanzado la cabeza de la lista sin ser accedido de nuevo, es colocado en la cabeza de la lista en vez de al final.

8.5.2 Cabeceras de los buffers

Cada buffer consta de dos partes una cabecera y el bloque de datos que almacena. El núcleo utiliza la *cabecera* para: identificar y localizar al buffer, sincronizar el acceso al mismo, y administrar del comportamiento de la caché. La cabecera de un buffer también se utiliza para pasar parámetros al manejador o driver del disco. Cuando el núcleo desea leer o escribir el buffer en el disco, carga los parámetros de la operación de E/S en la cabecera y pasa esta cabecera al driver del disco. La cabecera contiene toda la información requerida por la operación de disco.

Campos	Descripción
<code>int b_flags</code>	Indicadores del estado del buffer
<code>struct buf *b_forw, *b_back</code>	Punteros para mantener el buffer en la cola hash
<code>struct buf *av_forw, *av_back</code>	Punteros para mantener el buffer en la lista de buffers libres
<code>cadrr_t b_addr</code>	Puntero al bloque de datos del buffer
<code>dev_t b_edev</code>	Número de dispositivo
<code>daddr_t b_blkno</code>	Número de bloque en el dispositivo
<code>int b_error</code>	Estado de error E/S
<code>unsigned b_resid</code>	Número de bytes que restan por transferir

Tabla 8.1: Campos de la estructura *buf*

Formalmente, la cabecera de un buffer está implementada mediante una estructura *buf* cuyos campos más importantes se listan en la Tabla 8.1. El campo `b_flags` es un mapa de bits de varios indicadores. Por ejemplo, el núcleo usa los indicadores `B_BUSY` (bloqueado) y `B_WANTED` (deseado) para sincronizar el acceso al buffer, el indicador

B_DELWRI para marcar un buffer como “sucio” y el indicador B_AGE para marcar a un buffer que es un buen candidato para ser reutilizado. Asimismo el driver del disco también usa algunos indicadores, como por ejemplo: B_READ, B_WRITE, B_ASYNC, B_DONE y B_ERROR.

8.5.3 Ventajas

Usar la caché de buffers reduce el tráfico con el disco y elimina las operaciones de E/S al disco innecesarias. Asimismo la caché de buffers sincroniza el acceso a los bloques del disco mediante los indicadores B_BUSY (bloqueado) y B_WANTED (deseado). Si dos procesos intentan acceder al mismo bloque, solo uno será capaz de bloquearlo. Asimismo, la caché de buffer ofrece un interfaz modular entre el driver del disco y el resto del núcleo. Ninguna otra parte del núcleo puede acceder al driver del disco, y el interfaz entero está encapsulado en los campos de la cabecera del buffer.

8.5.4 Inconvenientes

A pesar de sus muchas ventajas, existen algunos inconvenientes importantes en la caché de buffers. En primer lugar, la política de escritura de la caché que es del tipo post-escritura implica que los datos se pueden perder si el sistema se cuelga. Esto podría dejar al disco en un estado inconsistente. En segundo lugar, aunque reducir el acceso al disco mejora el rendimiento del sistema, los datos deben ser copiados dos veces, primero del disco al buffer, y después del buffer al espacio de direcciones del usuario. La segunda copia es varios ordenes de magnitud más rápida que la primera, y normalmente el ahorro de accesos a disco compensa de sobra la copia adicional memoria-memoria que debe realizarse. Esto puede llegar a ser, sin embargo, un factor importante, cuando se lee o se escribe secuencialmente un fichero grande hasta el final y después no se vuelve a acceder a él de nuevo. De hecho, tal operación crea un problema adicional que es *la sustitución de todo el contenido de la caché*. Puesto que todos los bloques del fichero son leídos en un periodo de tiempo muy pequeño, consume todos los buffers en la caché, borrando todos los datos que se encontraban allí almacenados. Esto produce un gran número de fallos en la caché durante un rato, relantizando al sistema hasta que la caché se llena de nuevo con un conjunto de bloques más útiles. Este problema puede ser evitado si el usuario puede predecirlo. El sistema de ficheros Veritas (VxFS), por ejemplo, permitía al usuario suministrar consejos sobre como un fichero debía ser accedido. Usando esta característica, un usuario podía desactivar la carga en la caché de buffers de ficheros grandes y pedirle al sistema de ficheros que transfiriera los datos directamente del disco al espacio de usuario.

8.6 EL INTERFASE NODO-V/SFV

Sun Microsystems introdujo *el interfaz nodo-v/sfv* (nodo virtual/sistema de ficheros virtual) para suministrar un marco de trabajo en el núcleo que permitiera el acceso y la manipulación de diferentes tipos de sistemas de ficheros. El *interfaz nodo-v/sfv* se basa en dos conceptos: *nodo virtual* y *sistema de ficheros virtual*). Desde su aparición ha ido ganando una amplia aceptación, SVR4 fue la primera distribución del UNIX System V que incluyó este interfaz.

El *interfaz nodo-v/sfv* permite al sistema UNIX:

- Soportar diferentes tipos de sistemas de ficheros locales simultáneamente, tanto UNIX (*s5fs* o *ufs*⁴) y no-UNIX (DOS, A/UX, etc).
- Soportar sistema de ficheros distribuidos. Un sistema de ficheros en una máquina remota puede ser accedido de igual forma que un sistema de ficheros local.
- Presentar al usuario una imagen homogénea (árbol) del sistema de ficheros.
- Poder añadir al núcleo nuevos sistemas de ficheros de una forma modular.

En definitiva, el interfaz nodo-v/sfv es una capa de código del subsistema de ficheros del núcleo (ver Figura 8.6) que se encarga de traducir cualquier llamada al sistema u operación del núcleo sobre un fichero (o sobre un sistema de ficheros) a la función adecuada según el tipo de sistema de ficheros.

Por ejemplo, cuando un proceso realiza una llamada al sistema `read` sobre un fichero, el núcleo en primer lugar invoca a una función contenida en el interfaz nodo-v/sfv asociada a esta llamada al sistema que realiza un primer conjunto de operaciones independientemente del sistema de ficheros al que pertenezca el fichero. A continuación, esta función invoca a otra función cuya implementación depende del sistema de ficheros al que pertenece el fichero que se encarga de realizar la operación de lectura sobre el fichero.

⁴ *ufs* es el acrónimo inglés de *UNIX file system* que es el nombre que recibió el sistema de ficheros FFS cuando se integró en el sistema el interfaz nodo-v/sfv.

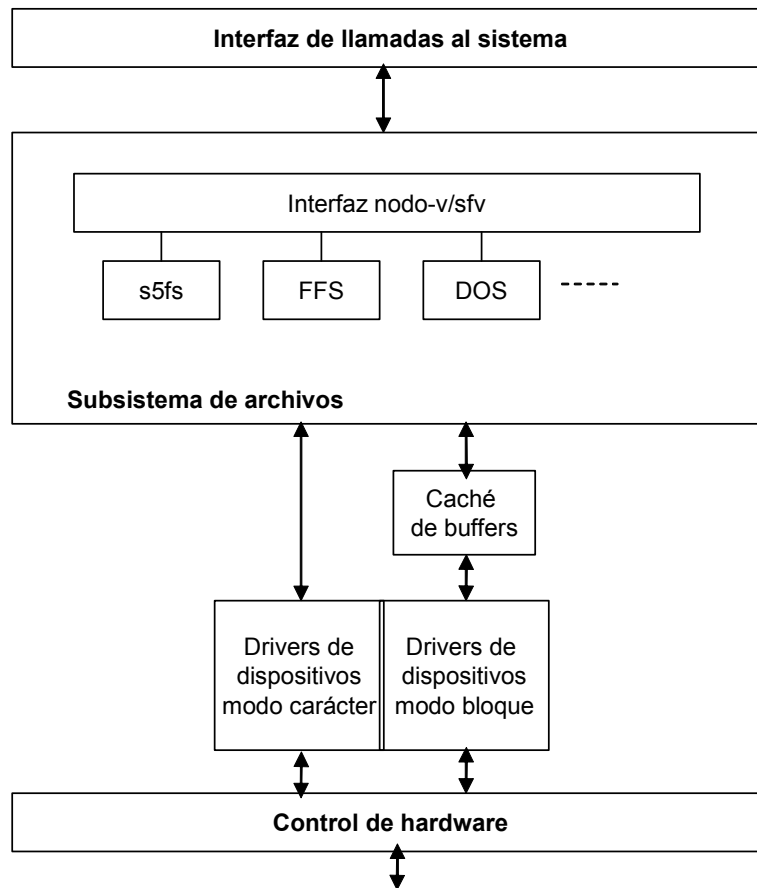


Figura 8.6: Ubicación del interfaz nodo-v/sfv dentro del núcleo

8.6.1 Una breve introducción a la programación orientada a objetos

El interfaz nodo-v/sfv fue diseñado usando conceptos de *programación orientada a objetos*. Estos conceptos han sido ampliados a otras áreas del núcleo de UNIX, tales como la administración de memoria, la comunicación basada en mensajes, y la planificación de procesos. Por lo tanto se hace necesario revisar brevemente los fundamentos de la programación orientada a objetos en la forma en que se aplica al núcleo de UNIX. Aunque tales técnicas se desarrollan de forma natural mediante lenguajes orientados a objetos tales como C++, los programadores de UNIX han preferido implementarlos en el lenguaje C para ser consistentes con el resto del núcleo de UNIX.

La aproximación orientada a objetos está basada en la noción de clases y objetos. Una *clase* es un tipo de dato complejo, que consta de campos de datos miembros y un conjunto de funciones miembros. Un *objeto* es una instancia de una clase. Las funciones miembros de una clase operan sobre los objetos individuales de la clase. Cada miembro

(campo de datos o función) de una clase puede ser o *pública* o *privada*. Sólo los miembros públicos son visibles externamente a los usuarios de la clase. Los datos y funciones privados pueden ser únicamente accedidos internamente por las otras funciones de la clase.

Para una clase cualquiera dada, podemos generar una o más clases derivadas, llamadas *subclases* (ver Figura 8.7). Una subclase puede ser en si misma una base para clases adicionales derivadas, estableciéndose por tanto una jerarquía de clases. Una subclase hereda todos los atributos (datos y funciones) de la clase base. También puede añadir sus propios datos y funciones. Además puede borrar algunas de las funciones de la clase base y suministrar su propia implementación de éstas.

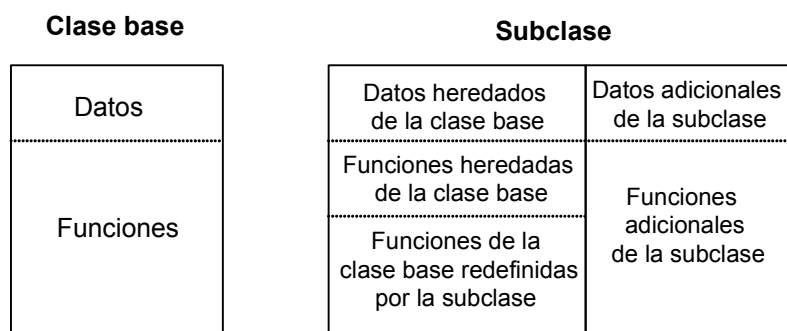


Figura 8.7: Relación entre una clase base y su subclase

Puesto que una subclase contiene todos los atributos de la clase base, un objeto del tipo subclase es también un objeto de la clase base. Por ejemplo, la clase *directorio* puede ser una clase derivada de la clase base *fichero*. Esto significa que cada directorio es también un fichero. Por lo tanto, un puntero a un objeto directorio es también un puntero a un objeto fichero. Los atributos añadidos por la clase derivada no son visibles por la clase base. Por tanto un puntero a un objeto base no puede ser usado para acceder a los datos y las funciones de la clase derivada.

Frecuentemente, se usa una clase base simplemente para representar una abstracción y definir un interfaz, con clases derivadas suministrando implementaciones específicas de las funciones base. Así la clase *fichero* puede definir una función llamada `create()`, pero cuando un usuario llama a esta función para un fichero arbitrario, se invoca a una rutina diferentes dependiendo de si el fichero es un fichero regular, un directorio, un enlace simbólico, un fichero de dispositivo, etc. De hecho, se puede no tener una implementación genérica de `create()` que cree un fichero arbitrario. A tal función se le denomina una *función virtual pura*.

Los lenguajes orientados a objetos suministran estos servicios. En C++, por ejemplo, es posible definir una *clase base abstracta* como aquella que contiene al menos una *función virtual pura*. Puesto que la clase base no tiene ninguna implementación para esta función, no puede ser instanciada. Puede solamente ser usada para derivar subclases, que suministran implementaciones específicas para las funciones virtuales. Todos los objetos son instancias de una subclase u otra, pero el usuario puede manipularlos usando un puntero a la clase base, sin conocer a que subclase pertenece. Cuando una función virtual es invocada por tal objeto, la implementación automáticamente determina a que función específica debe llamar, dependiendo del subtipo actual del objeto.

8.6.2 Perspectiva general del interfaz node-v/sfv

La abstracción *nodo virtual* (*nodo-v*) representa a un fichero en el núcleo de UNIX. Por su parte, la abstracción *sistema de ficheros virtual* (*sfv*) representa a un sistema de ficheros. Ambas son consideradas como clases bases abstractas, a partir de las cuales se pueden derivar subclases que suministran implementaciones específicas para los diferentes tipos de sistemas de ficheros tales como *s5fs*, *ufs*, FAT (el sistema de ficheros de MS-DOS),...

En C, una clase base es implementada como una estructura, más un conjunto de funciones del núcleo globales (y macros) que definen las funciones no virtuales públicas. La clase base contiene un puntero a otra estructura que consiste de un conjunto de punteros a funciones, uno por cada función virtual.

8.6.2.1 La clase *nodo-v*

La Figura 8.8 muestra la clase *nodo-v* en SVR4. Los campos datos en la base *nodo-v* contienen información que no dependen del tipo de sistema de ficheros. Las funciones miembros pueden ser divididas en dos categorías. La primera es un conjunto de funciones virtuales que define el interfaz dependiente del sistema de ficheros. Cada sistema de ficheros diferente debe suministrar su propia implementación de estas funciones. La segunda es un conjunto de *rutinas de utilidad y macros* que pueden ser usadas por otros subsistemas del núcleo para manipular los ficheros. Estas funciones a su vez llaman a rutinas dependientes del sistema de ficheros para realizar tareas de bajo nivel.

La base *nodo-v* tiene dos campos que permiten implementar subclases. El primero es `v_data`, que es un puntero (de tipo `caddr_t`) a una estructura de datos privada que mantiene datos del sistema de ficheros específico del *nodo-v*. Para *s5fs* y *ufs*, esta

estructura es simplemente la tradicional estructura `inode` (nodo-i) (para `s5fs` y `ufs`, respectivamente). NFS utiliza una estructura `rnode`, etc. Puesto que esta estructura es accedida indirectamente a través de `v_data`, es opaca a la clase base `vnode`, y sus campos son únicamente visibles a las funciones internas al sistema de ficheros específico.

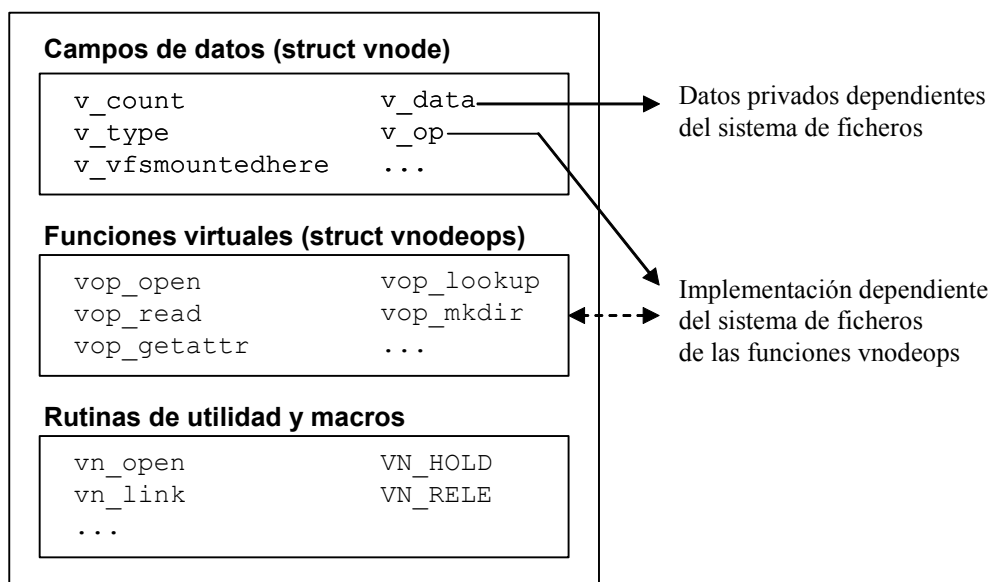


Figura 8.8: La abstracción nodo-v.

El campo `v_op` apunta a la estructura `vnodeops`, que consta de un conjunto de punteros a las funciones que implementan el interfase virtual del nodo-v. Tanto el campo `v_data` como el campo `v_op` son configurados cuando el nodo-v es inicializado, típicamente durante una llamada al sistema `open` o `creat`. Cuando el código independiente del sistema de ficheros llama a una función virtual para un nodo-v arbitrario, el núcleo usando el puntero `v_op` llama a la función correspondiente de la implementación del sistema de ficheros adecuada. Por ejemplo, la operación `VOP_CLOSE` permite al proceso invocador cerrar al fichero asociado con el nodo-v, que es accedida mediante una macro. Una vez que los nodos-v han sido apropiadamente inicializados, esta macro asegura que invocando a la operación `VOP_CLOSE` se llamaría a la rutina `ufs_close` para un fichero `ufs`, a la rutina `nfs_close` para un fichero NFS, etc.

De forma general un objeto nodo-v se implementa mediante la siguiente estructura de datos⁵:

```
struct vnode {
    u_short v_flags;           /*V_ROOT, etc*/
    u_short v_count;          /*Contador de referencias*/
    struct vfs *vfsmountedhere; /*Para puntos de montaje*/
    struct vnodeops *v_op;     /*Vector de operaciones sobre el
                               nodo-v*/
    struct vfs *vfsp;          /*Sistema de ficheros al que
                               pertenece*/
    struct stdata *v_stream;   /*Puntero al stream asociado si
                               existe alguno*/
    struct page *v_page;       /*Lista de páginas residente*/
    enum vtype v_type;         /*Tipo de fichero*/
    dev_t v_rdev;              /*Identificador del dispositivo
                               para ficheros de dispositivos*/
    caddr_t v_data;           /*Puntero a una estructura de
                               datos privada*/
    ...
};
```

8.6.2.2 La clase sfv

De forma similar, la clase base sfv (ver Figura 8.9) tiene dos campos: `vfs_data` y `vfs_op` permiten enlazar a las subclases y por tanto suministrar el acceso en tiempo de ejecución a las funciones y datos dependientes del sistema de ficheros.

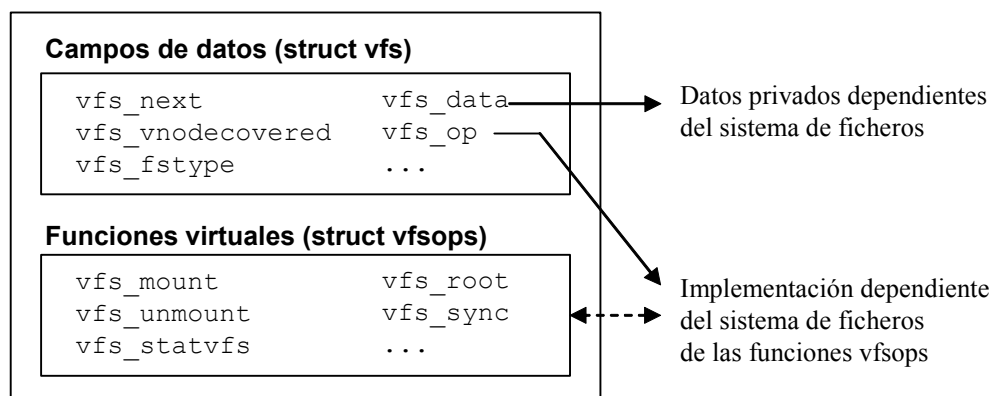


Figura 8.9: La abstracción sfv

⁵ Con el fin de ahorrar espacio las definiciones de estructuras que se presentan en esta sección no están completas, este hecho se pone de manifiesto escribiendo “...” antes del final de la definición.

El objeto *sfv* (`struct vfs`) representa a un sistema de ficheros. El núcleo asocia un objeto *sfv* para cada sistema de ficheros activo. Está descrito por la siguiente estructura de datos:

```
struct vfs    {
    struct vfs *vfs_next;           /*Siguiente VFS en la
                                   lista*/
    struct vfsops *vfs_op;          /*Vector de operaciones*/
    struct vnode *vfs_vnodecovered; /*Nodo-v de montaje*/
    int vfsfstype;                  /*Indice del tipo de sistema
                                   de ficheros*/
    cadrr_t vfs_data;               /*Datos privados*/
    dev_t vfs_dev;                  /*identificador
                                   del dispositivo*/
    ...
};
```

◆ Ejemplo 8.9:

La Figura 8.10 muestra la relación entre los objetos nodo-v y *sfv* en un sistema que contiene dos sistemas de ficheros. El segundo sistema de ficheros está montado en el directorio `/usr` del sistema de ficheros raíz. La variable global `rootvfs` apunta a la cabecera de la lista enlazada de todos los objetos *sfv*, que es la estructura *vfs* asociada al sistema de ficheros raíz. El campo `vfs_vnodecovered` apunta al nodo-v en que está montado el sistema de ficheros.

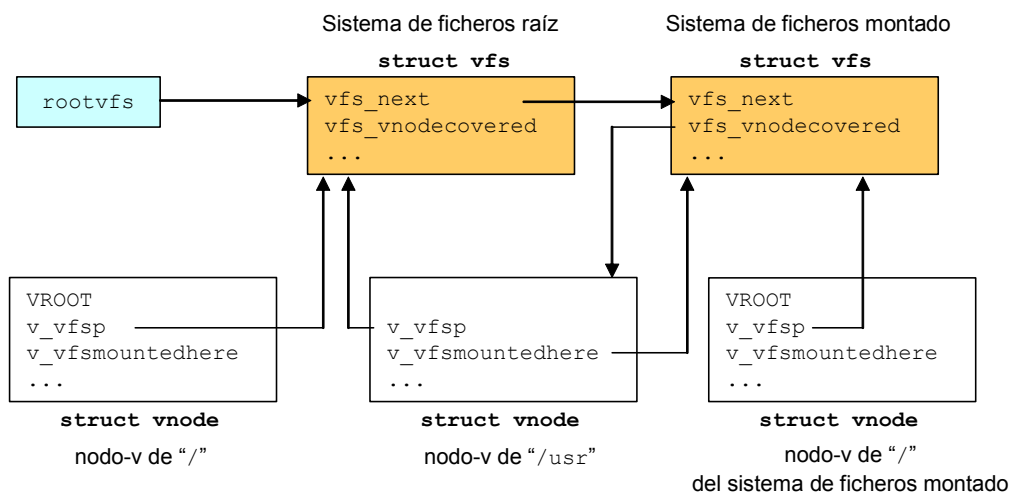


Figura 8.10: Relaciones entre los objetos nodo-v y *sfv*

El campo `v_vfsp` de cada nodo-v apunta al *sfv* al que pertenece. Los nodos-v raíces de cada sistema de ficheros tienen el indicador `VROOT` activado. Si un nodo-v es un punto de montaje, su

campo `v_vfsmountedhere` apunta al objeto `sfv` del sistema de ficheros montado sobre él. Obsérvese que el sistema de ficheros raíz no está montado en ninguna parte.



8.6.3 Nodos virtuales y ficheros abiertos

El *nodo virtual* (nodo-v) es la abstracción fundamental que representa a un fichero activo en el núcleo. Define el interfaz al fichero y canaliza todas las operaciones sobre el fichero a las funciones específicas del sistema de ficheros apropiado. Hay dos formas mediante las cuales el núcleo accede a un nodo-v. La primera es mediante las llamadas al sistema asociadas a E/S, que localizan el nodo-v a través de su descriptor de fichero, como se describirá en esta sección. La segunda, es mediante las rutinas de análisis de rutas de acceso, que utilizan las estructuras de datos dependientes del sistema de ficheros para localizar el nodo-v.

Un proceso debe abrir un fichero antes de leer o escribir en él. La llamada al sistema `open` devuelve un descriptor de fichero al proceso invocador. Este descriptor, que es típicamente un entero pequeño, actúa como un manejador para el fichero y representa una sesión independiente, o flujo, para el fichero. El proceso debe pasar el descriptor a las llamadas al sistema `write` o `read` que realice.

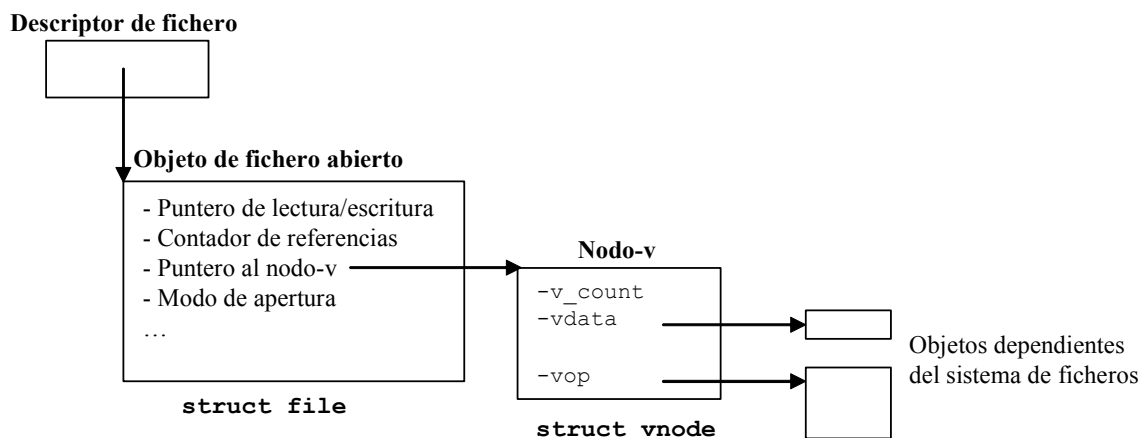


Figura 8.11: Estructuras independientes del sistema de ficheros

El *descriptor del fichero* es un objeto para cada proceso, contiene un puntero (ver Figura 8.11) a un objeto de fichero abierto (`struct file`). Asimismo contiene un conjunto de indicadores por descriptor. Entre los indicadores implementados se encuentran `FCLOSEEXEC`, que pide al núcleo que cierre el descriptor cuando el proceso invoca a la llamada al sistema `exec`, y `U_FDLCK` que se utiliza para bloquear el fichero.

El *objeto de fichero abierto* almacena la información necesaria que permite administrar una sesión con el fichero. Si varios usuarios tienen el fichero abierto (o el mismo usuario lo ha abierto varias veces), cada uno tiene su propio objeto de fichero abierto. Sus campos incluyen:

- *Puntero de lectura/escritura* desde el origen del fichero, para indicar donde debe comenzar la siguiente operación de lectura o escritura sobre el fichero.
- *Contador de referencias* que indica el número de descriptores de ficheros que apuntan a él. Normalmente es 1, pero podría ser mayor si los descriptores son clonados mediante `dup` o `fork`.
- *Puntero al nodo-v del fichero*.
- *Modo de apertura del fichero*. El núcleo comprueba este modo en cada operación de E/S. Por tanto si un usuario ha abierto un fichero de sólo lectura, el no podrá escribir en el fichero usando este descriptor incluso aunque tenga los privilegios necesarios.

Los sistemas UNIX tradicionales usan una *tabla de descriptores de ficheros* de tamaño fijo y estática que se aloja en el área U. El descriptor devuelto al usuario es un índice dentro de esta tabla. El tamaño de la tabla (típicamente 64 elementos) limita el número de ficheros que el usuario puede tener abiertos al mismo tiempo. En los sistemas UNIX modernos, la tabla de descriptores puede tener un tamaño mucho mayor.

Algunas implementaciones, tales como SVR4 o SunOS, alojan los descriptores en tablas de (normalmente) 32 entradas y guardan estas tablas en listas enlazadas, con la primera tabla alojada en el área U del proceso. De este modo, en vez de simplemente usar el descriptor como un índice en una única tabla, el núcleo primero tiene que localizar la tabla apropiada y después acceder a la entrada adecuada dentro de dicha tabla. Este esquema elimina las restricciones sobre el número de ficheros que un proceso puede tener abierto, pero complica la complejidad del código y el rendimiento del sistema.

Algunas nuevas distribuciones basadas en SVR4 alojan la tabla de descriptores dinámicamente y la extienden cuando es necesario llamando a la rutina `kmem_realloc()`, que o extiende la tabla en el mismo lugar o la copia en una nueva localización donde su espacio haya aumentado.

8.6.4 El contador de referencias del nodo-v

El campo `v_count` del nodo-v mantiene un contador de referencias que determina cuanto tiempo el nodo-v debe permanecer en el núcleo. Un nodo-v es alojado y asignado a un fichero cuando el fichero es accedido por primera vez. Por lo tanto, otros objetos pueden mantener punteros, o referencias, a este nodo-v y esperar para acceder al nodo-v usando el puntero. Esto significa que si esta referencia existe, el núcleo debe retener el nodo-v y no reasignarlo a otro fichero.

Este contador de referencias es una de las propiedades genéricas de un nodo-v y es manipulado por el código independiente del sistema de ficheros. Dos macros, `VN_HOLD` y `VN_RELE`, incrementan y decrementan el contador de referencias, respectivamente. Cuando el contador de referencias alcanza el valor 0, el fichero está inactivo y el nodo-v puede ser liberado o reasignado.

Es importante distinguir entre referencia y bloqueo. Bloquear un objeto impide que otros procesos accedan a él de una cierta forma, dependiendo de si el bloqueo es exclusivo o de lectura/escritura. Mantener una referencia a un objeto simplemente asegura la persistencia del objeto. El código independiente del sistema de ficheros bloquea un nodo-v durante periodos de tiempo cortos, típicamente durante la duración de una única operación sobre un nodo-v. Una referencia es típicamente mantenida durante un tiempo largo, no solamente a través de múltiples operaciones con el nodo-v sino también a través de múltiples llamadas al sistema. Algunas de las operaciones que requieren la referencia de un nodo-v son:

- La apertura de un fichero requiere la adquisición de una referencia, en consecuencia el contador de referencias del nodo-v se incrementa. Por el contrario cerrar el fichero libera la referencia, es decir, se decrementa el contador de referencias del nodo-v.
- Un proceso siempre mantiene una referencia a su directorio de trabajo actual. Cuando el proceso cambia de directorio de trabajo, adquiere una referencia al nuevo directorio y libera la referencia al directorio viejo.
- Cuando un nuevo sistema de ficheros es montado, adquiere una referencia al directorio de punto de montaje. Desmontar el sistema de ficheros libera dicha referencia.
- La rutina de análisis de rutas de acceso adquiere una referencia en cada directorio intermedio que se encuentra en su búsqueda. Mantiene la referencia

mientras busca el directorio y la libera después de adquirir una referencia al siguiente componente de la ruta.

El contador de referencias asegura la persistencia del nodo-v y también del fichero que subyace. Cuando un proceso borra un fichero que otro proceso (o quizás el mismo proceso) había abierto, el fichero no se borra físicamente. La entrada del directorio de dicho fichero es eliminada así que nadie más puede abrirlo. El fichero en si mismo continua existiendo puesto que el nodo-v tiene un contador de referencias distinto de cero. Los procesos que actualmente tenían el fichero abierto pueden continuar accediendo a él hasta que lo cierren. Cuando la última referencia sea liberada, el código independiente del sistema de ficheros invocará la operación `VOP_INACTIVE` para completar el borrado del fichero. Para un fichero *ufs* o *s5fs*, por ejemplo, el nodo-i y los bloques de datos serán liberados en este momento.

8.6.5 Objetos dependientes del sistema de ficheros

8.6.5.1 Partes dependientes del sistema de ficheros de un objeto nodo-v

El nodo-v es un objeto abstracto que no puede existir por si solo sino que debe ser instanciado en el contexto de un fichero específico. Los campos `v_op` y `v_data` del nodo-v enlazan a la parte dependiente del sistema de ficheros. `v_data` apunta a una estructura de datos privada que mantiene información dependiente del sistema de ficheros. La estructura de datos depende del sistema de ficheros al que pertenece el fichero, por ejemplo para ficheros *s5fs* y *ufs* se utiliza la estructura que define su nodo-i.

`v_data` es un puntero opaco, lo que significa que el código independiente del sistema de ficheros no puede directamente acceder al objeto dependiente del sistema de ficheros. El código dependiente del sistema de ficheros, sin embargo, si que puede acceder a los objetos nodo-v base. Se necesita, por lo tanto, una forma de localizar al nodo-v a través del objeto de datos privado. Puesto que los dos objetos son siempre asignados conjuntamente, es eficiente combinarlos en uno solo. De esta forma en las implementaciones estándar de la referencia de la capa del nodo-v, el nodo-v es simplemente una parte del objeto dependiente del sistema de ficheros.

Por otra parte, el interfaz del nodo-v define un conjunto de operaciones sobre un fichero genérico. El código independiente del sistema de ficheros manipula el fichero usando estas operaciones únicamente. Este código no puede acceder a los objetos dependientes del sistema de ficheros directamente. La estructura `vnodeops`, que implementa esta interfaz se define de la siguiente forma:

```

struct vnodeops{
    int (*vop_open) ();
    int (*vop_close) ();
    int (*vop_read) ();
    int (*vop_write) ();
    int (*vop_create) ();
    int (*vop_remove) ();
    int (*vop_link) ();
    int (*vop_mkdir) ();
    int (*vop_rmdir) ();
    int (*vop_lookup) ();
    int (*vop_inactive) ();
    int (*vop_rwlock) ();
    int (*vop_rwunlock) ();
    int (*vop_getpage) ();
    ...
};

```

Cada sistema de ficheros implementa este interfaz de una forma distinta suministrando su propio conjunto de funciones. Por ejemplo, *ufs* implementa la operación VOP_READ leyendo el fichero del disco local mientras que NFS envía una petición a un servidor remoto para obtener el dato. Por lo tanto cada sistema de ficheros suministra una instancia de la estructura `vnodeops`, por ejemplo, *ufs* define el objeto:

```

struct vnodeops  ufs_vnodeops = {
    ufs_open,
    ufs_close,
    ...
};

```

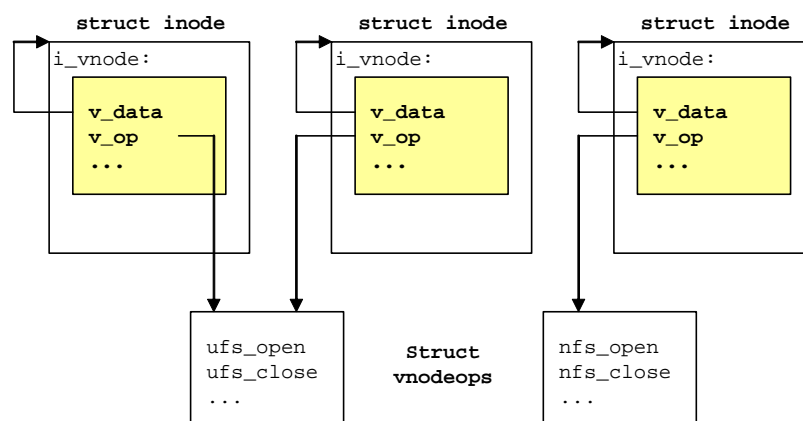


Figura 8.12: Objetos de un nodo-v dependientes del sistema de ficheros

El campo `v_op` del nodo-`v` apunta a la estructura `vnops` para el tipo de sistema de ficheros asociado. Como se muestra en la Figura 8.12, todos los ficheros del mismo tipo de sistema de ficheros comparten una misma instancia de esta estructura y acceden al mismo conjunto de funciones.

8.6.5.2 Partes dependientes del sistema de ficheros de un objeto `sfs`

Como el nodo-`v`, el objeto `sfs` tiene punteros a sus datos privados y a su vector de operaciones. El campo `vfs_data` es un puntero opaco, que apunta a una estructura de datos por cada sistema de ficheros. A diferencia de los nodos-`v`, el objeto `sfs` y su estructura de datos privada normalmente se asignan por separado. El campo `vfs_op` apunta a una estructura `vfops`, que se define de la siguiente forma:

```
struct vnops {
    int (*vfs_mount)();
    int (*vop_unmount)();
    int (*vop_root)();
    int (*vop_statvfs)();
    int (*vop_sync)();
    ...
};
```

Cada tipo de sistema de ficheros suministra su propia implementación de estas operaciones. Por lo tanto existe una instancia de la estructura `vfops` por cada tipo de sistema de ficheros: `ufs_vfops` para `ufs`, `nfs_vfops` para NFS, etc. La Figura 8.13 muestra las estructuras de datos de la capa `sfs` para un sistema que contiene dos sistemas de ficheros del tipo `ufs` y un sistema de ficheros del tipo NFS.

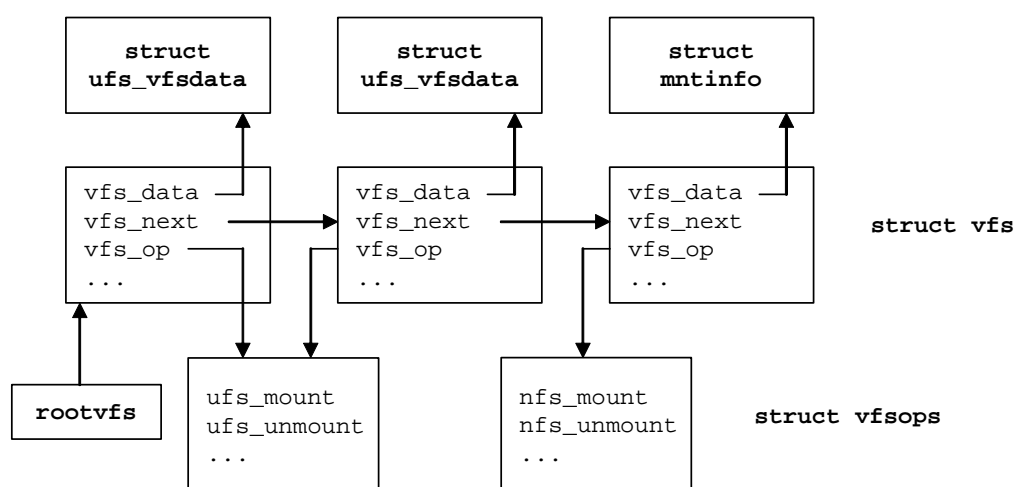


Figura 8.13: Estructuras de datos de la capa `sfs`

8.6.6 Montaje de un sistema de ficheros

8.6.6.1 El conmutador del sistema de ficheros virtual

EL SVR4 mantiene una tabla global denominada *conmutador del sistema de ficheros* virtual que contiene una entrada por cada tipo de sistema de ficheros existente en el sistema. En cada entrada se almacena una estructura `vfssw`, cuya definición es:

```
struct vfssw {
    char *vsw_name;           /* Tipo del sistema de ficheros */
    int (*vsw_init)();        /* Dirección de la rutina de
                               inicialización */
    struct vfsops *vsw_vfsops; /* Vector de operaciones para
                               este sistema de ficheros*/
} vfssw[];
```

El núcleo usa esta tabla para poder encaminar hacia las implementaciones específicas de cada sistema de ficheros las operaciones sobre los objetos nodo-v y sfv.

8.6.6.2 Implementación de mount

La llamada al sistema `mount` obtiene el nodo-v del directorio punto de montaje llamando a la rutina `lookupn()`. Esta rutina comprueba que el nodo-v representa a un directorio y que no existe ningún otro sistema de ficheros montado en sobre él. Después busca la tabla `vfssw[]` para encontrar la entrada que se ajusta al nombre dado por el argumento `tipo`.

Una vez localizada la entrada en esta tabla, el núcleo invoca a su operación `vsw_init`, que llama a una rutina de inicialización específica del sistema de ficheros que asigna las estructuras de datos y los recursos necesarios para operar con el sistema de ficheros. A continuación, el núcleo asigna una nueva estructura `vfs` y la inicializa de la siguiente forma:

1. Añade la estructura a la lista enlazada encabezada por `rootvfs`.
2. Configura el campo `vfs_ops` para apuntar al vector `vfsops` especificado en la entrada de la tabla de conmutación.
3. Configura el campo `vfs_vnodecovered` para apuntar al nodo-v del directorio punto de montaje.

Después el núcleo almacena un puntero a la estructura `vfs` en el campo `v_vfsmountedhere` del nodo-v del directorio cubierto. Finalmente invoca a la operación `VFS_MOUNT` del `sfv` para realizar el procesamiento dependiente del sistema de ficheros de la llamada `mount`.

8.6.6.3 Procesamiento VFS_MOUNT

Cada sistema de ficheros suministra su propia función para implementar la operación `VFS_MOUNT`. Esta función debe realizar las siguientes operaciones:

1. Verificar permisos para la operación.
2. Asignar e inicializar el objeto de datos privados del sistema de ficheros.
3. Almacenar un puntero a este objeto en el campo `vfs_data` del objeto `sfv`.
4. Acceder al directorio raíz del sistema de ficheros e inicializar su nodo-v en memoria. La única forma de que el núcleo acceda a la raíz de un sistema de ficheros montado es mediante la operación `VFS_ROOT`. La parte de `sfv` dependiente del sistema de ficheros debe mantener la información necesaria para localizar el directorio raíz.

Típicamente, los sistemas de ficheros locales pueden implementar `VFS_MOUNT` leyéndola en los metadatos del sistema de ficheros (como por ejemplo el superbloque en el `s5fs`) desde el disco, mientras que los sistemas de ficheros distribuidos pueden enviar una petición de montaje remoto al servidor.

8.6.7 Operaciones sobre ficheros

8.6.7.1 Análisis de rutas de acceso

La función independiente del sistema de ficheros `lookuppn()` traduce una ruta de acceso y devuelve un puntero al nodo-v del fichero deseado. También establece una referencia sobre este nodo-v. El punto de comienzo del análisis de la ruta de acceso depende de si ésta es relativa o absoluta. Para rutas de acceso relativas, `lookuppn()` comienza en el directorio de trabajo actual, obteniendo el puntero a su nodo-v del área `U`. Para rutas absolutas, comienza en el directorio raíz, cuyo puntero a su nodo-v se encuentra en la variable global `rootdir`.

`lookuppn()` incrementa el contador de referencias del nodo-v del directorio de comienzo de la búsqueda, y después ejecuta un bucle para ir analizando de uno en uno

cada componente de la ruta de acceso. En cada iteración del lazo debe realizar las siguientes tareas:

1. Asegurarse de que el nodo-v es un directorio (excepto si se ha alcanzado el último componente de la ruta). El campo `v_type` en el nodo-v contiene esta información.
2. Si el componente es “.” y el directorio actual es el raíz del sistema, se pasa a analizar el siguiente componente de la ruta. El directorio raíz del sistema actúa como su propio directorio padre.
3. Si el componente es “.” y el directorio actual es el directorio raíz de un sistema de ficheros montado, accede al directorio punto de montaje. Si un directorio es raíz de un sistema de ficheros entonces tendrá su indicador VROOT activado. El campo `v_vfsp` apunta a la estructura `vfs` para dicho sistema de ficheros, que contiene un puntero al punto de montaje en el campo `vfs_vnodecovered`.
4. Invocar a la operación VOP_LOOKUP sobre este nodo-v, que realiza una llamada a la función de búsqueda específica del sistema de ficheros al que pertenezca (`s5lookup()` para *s5fs*, `ufs_lookup()` para *ufs*, etc). Esta función busca el componente de la ruta dentro del directorio, y si lo encuentra devuelve un puntero al nodo-v de dicho fichero (alojándolo en el núcleo si no estaba ya alojado allí). También establece una referencia sobre este nodo-v.
5. Si el componente no fue encontrado, comprueba si se trataba del último componente de la ruta. Si es así, finaliza con éxito devolviendo un puntero al directorio pero sin eliminar la referencia que había creado. En caso contrario devuelve el error ENOENT.
6. Si el nuevo componente es un punto de montaje (para ello comprueba que el valor almacenado en `v_vfsmountedhere` es distinto del valor nulo) sigue el puntero al objeto `sfv` del sistema de ficheros montado e invoca su operación `vfs_root` para obtener el nodo-v del directorio raíz de este sistema de ficheros.
7. Si el nuevo componente es un enlace simbólico (`v_type==VLNK`), se invoca a su operación VOP_SYMLINK para traducir el enlace simbólico. Se adjunta el resto de la ruta de acceso a los contenidos del enlace y se reinicia la iteración.

Si el enlace contiene una ruta de acceso absoluta, la búsqueda debe retomarse desde el directorio raíz del sistema.

8. Libera el directorio si ya ha finalizado la búsqueda. La referencia fue realizada por la operación VOP_LOOKUP. Para el punto de comienzo de la búsqueda, la referencia fue obtenida de forma explícita por `lookuppn()`.
9. Finalmente, vuelve al principio del lazo y busca el siguiente componente en el directorio representado por el nuevo nodo-v.
10. El análisis termina cuando ya no quedan más componentes en la ruta, o si un componente no fue encontrado. Si la búsqueda se realizó con éxito, no liberar la referencia del nodo-v final y devuelve al proceso invocador un puntero a este nodo-v.

8.6.7.2 La caché de búsqueda de nombres en directorios

La *caché de búsqueda de nombres en directorios* es un recurso global del núcleo disponible para todos los sistemas de ficheros que deseen utilizarlo. Se trata de una caché software LRU de objetos que contienen: un puntero al nodo-v de un directorio, el nombre de un fichero en este directorio, y un puntero al nodo-v de dicho fichero.

Si un sistema de ficheros desea utilizar la *caché de búsqueda de nombres en directorios*, su función de búsqueda, es decir aquella que implementa la operación VOP_LOOKUP, primero busca el nombre deseado en la caché. Si lo encuentra, simplemente incrementa el contador de referencias del nodo-v y se lo devuelve al proceso invocador. De esta forma se evita buscar en el directorio y por lo tanto se ahorra varias lecturas a disco.

Los aciertos en la caché son bastante probables puesto que los programadores típicamente hacen varias peticiones de unos pocos ficheros y directorios que se utilizan frecuentemente. En el caso de un fallo en la caché, la función de búsqueda específica de cada sistema de ficheros buscará el nombre en el directorio padre. Cuando la componente es encontrada, se añade una nueva entrada en la caché de nombres con la información adecuada por si es necesitada de nuevo en el futuro.

8.6.7.3 La operación VOP_LOOKUP

VOP_LOOKUP es el interfaz a la función específica del sistema de ficheros que busca una componente de una ruta de acceso en un directorio. Se invoca a través de una macro de la siguiente forma:

```
error=VOP_LOOKUP(vp, componente, &tv, ...);
```

donde `vp` es un puntero al nodo-v del directorio padre y `componente` es el nombre de un componente en la ruta de acceso. Si se ejecuta con éxito, `tv` debe apuntar al nodo-v de `componente` y su contador de referencias debe ser incrementado.

Como otras operaciones en este interfaz, esto resulta en una llamada a una función de búsqueda de un sistema de ficheros específico. Usualmente, esta función busca el nombre en la caché de búsqueda de nombres en directorios. Si se produce un acierto, incrementa el contador de referencias y devuelve el puntero al nodo-v. En caso de fallo, busca el nombre en el directorio padre. Los sistemas de ficheros locales implementan la búsqueda iterando a través de las entradas del directorio bloque a bloque. Los sistemas de ficheros distribuidos envían una petición de búsqueda al nodo del servidor.

Si el directorio contiene el nombre que se buscaba, la función de búsqueda comprueba si el nodo-v del fichero se encuentra ya en memoria. Cada sistema de ficheros tiene su propio método para mantener la pista de sus objetos en memoria. En *ufs*, por ejemplo, la búsqueda en el directorio resulta en un número de nodo-i, que *ufs* utiliza como índice de búsqueda para buscar el nodo-i en una tabla de dispersión. El nodo-im en memoria contiene el nodo-v. Si el nodo-v es encontrado en memoria, la función de búsqueda incrementa su contador de referencias y retorna a su invocador.

A menudo la búsqueda en el directorio produce un acierto, pero el nodo-v no está en memoria. La función de búsqueda debe asignar e inicializar un nodo-v, así como las estructuras de datos privados dependientes del núcleo. Usualmente, el nodo-v es parte de la estructura de datos privados, y por lo tanto ambos son alojados como una sola unidad. Los dos objetos son inicializados leyendo los atributos del fichero. El campo `v_op` del nodo-v es configurado para que apunte al vector `vnodeops` para este sistema de ficheros, y una referencia es añadida al contador de referencias `v_count` del nodo-v. Finalmente, la función de búsqueda añade una entrada a la caché de búsqueda de nombres en directorios y la sitúa al final de la lista LRU de la caché.

8.6.7.4 Apertura de un fichero

La implementación de `open` es tratada casi por entero en la capa independiente del sistema de ficheros. El algoritmo es el siguiente:

1. Asignar un descriptor de fichero.
2. Asignar un objeto de fichero abierto (`struct file`) y almacenar un puntero a él en el descriptor del fichero. SVR4 asigna este objeto dinámicamente. Las distribuciones anteriores utilizaban una tabla estática de tamaño fijo (tabla de archivos).
3. Llamar a `lookupn()` para analizar la ruta de acceso y devolver el nodo-v del fichero para ser abierto. `lookupn()` también devuelve un puntero al nodo-v del directorio padre.
4. Comprobar el nodo-v (mediante la invocación de su operación `VOP_ACCESS`) para asegurarse que el invocador tiene los permisos necesarios para el tipo de acceso deseado.
5. Comprobar que no se realizan ciertas operaciones ilegales, tales como abrir un directorio o un fichero ejecutable activo para escribir (de lo contrario, el usuario ejecutando el programa obtendría resultados inesperados).
6. Si el fichero no existe, comprobar si la opción `O_CREAT` estaba especificada. Si es así, invocar `VOP_CREATE` sobre el directorio padre para crear el fichero. En caso contrario, devolver el código de error `ENOENT`.
7. Invocar la operación `VOP_OPEN` de este nodo-v para realizar el procesamiento dependiente del sistema de ficheros. Típicamente esta rutina no hace nada, pero algunos sistemas de ficheros pueden desear realizar tareas adicionales en este momento. Por ejemplo, el sistema de ficheros *specfs*, que trata todos los ficheros de dispositivo, podría desear llamar a la rutina `open` de un driver de dispositivo.
8. Si la opción `O_TRUNC` ha sido especificada, invocar a `VOP_SETATTR` para configurar el tamaño del fichero a 0. El código dependiente del sistema de ficheros realizará las operaciones de limpieza necesarias tales como liberar los datos de bloques del fichero.

9. Inicializar el objeto de fichero abierto. Almacenar el puntero al nodo-v y los indicadores del modo de apertura en su interior, configurar su contador de referencias a 1 y su puntero de desplazamiento a 0.
10. Finalmente, retornar el índice del descriptor de fichero al usuario.

Conviene darse cuenta de que `lookuppn()` incrementa el contador de referencias en el nodo-v y también inicializa su puntero `v_op`. Esto asegura que las siguientes llamadas al sistema puedan acceder al fichero usando el descriptor del fichero (el nodo-v permanece en memoria) y que las funciones dependientes del sistema de ficheros estarán adecuadamente encaminadas.

8.7 EL SISTEMA DE FICHEROS DEL UNIX SYSTEM V (S5FS)

8.7.1 Organización en el disco del s5fs

El sistema de ficheros reside en un único disco lógico o partición, y cada disco lógico puede contener un sistema de ficheros como máximo. Cada sistema de ficheros está autocontenido y completo, con su propio directorio raíz, subdirectorios, ficheros, y todos sus datos y metadatos asociados. El árbol de ficheros que es visible por el usuario está formado por la unión de uno o varios de estos sistemas de ficheros.

La Figura 8.14 muestra la estructura de una partición de disco para el sistema de ficheros del UNIX System V (*s5fs*). Una partición puede ser vista desde un punto de vista lógico como un array lineal de bloques. El tamaño de un bloque de disco es 512 bytes multiplicado por alguna potencia de dos (diferentes versiones han usado bloques de 512, 1024 o 2048 bytes). El *número de bloque físico* (o simplemente el número de bloque) es un índice dentro de este array, e identifica de forma única a un bloque en una partición de disco dada. Este número debe ser traducido por el manejador o driver del disco en cilindro, pista y número de sector. La traducción depende de las características físicas del disco (número de cilindros y pistas, sectores por pista, etc) y la localización de la partición en el disco.



Figura 8.14: Estructura en el disco del s5fs

Al comienzo de la partición se encuentra el *área de arranque*, que puede contener el código requerido para arrancar (carga e inicialización) del sistema operativo. De todas las particiones existentes solamente una de ellas necesita contener esta información, posiblemente el resto de particiones tendrán su área de arranque vacía.

A continuación del área de arranque se encuentra el *superbloque*, que contiene atributos y metadatos del propio sistema de ficheros. A continuación del superbloque se encuentra la *lista de nodos-i*, que es una array lineal de nodos-i. Hay un nodo-i por cada fichero. Cada nodo-i puede ser identificado por su *número de nodo-i*, que es igual al índice en la *lista de nodos-i*. El tamaño de un nodo-i es 64 bytes, luego varios nodos-i se pueden almacenar dentro de un bloque de disco.

La *lista de nodos-i* tiene un tamaño fijo (que se configura cuando se crea el sistema de ficheros en esta partición) que limita el número máximo de ficheros que la partición puede contener. El espacio después de la lista de nodos-i es el *área de datos*, que contiene bloques de datos para ficheros y directorios, así como *bloques indirectos*, que contienen punteros para bloques de datos de ficheros.

8.7.2 Directorios

Un *directorio en el sf5s* es un fichero especial que contiene una lista de ficheros y subdirectorios. Cada entrada en esta lista almacena 16 bytes por cada fichero o subdirectorio que contiene. De estos 16 bytes, los dos primeros contienen el número de nodo-i, y las catorce siguientes el nombre del fichero. Esta configuración establece un límite de 65535 ficheros por partición de disco (puesto que 0 no es un número de nodo-i válido) y 14 caracteres por nombre de fichero. Si el nombre del fichero tiene menos de catorce caracteres, éste termina con un carácter nulo '\0'.

Puesto que un directorio es un fichero, tiene también un nodo-i que contiene un campo que identifica al fichero como un directorio. Las dos primeras entradas del directorio son ".", que representa al propio directorio, y ".." que denota al directorio padre. Si el número de nodo-i de una entrada es cero, ello indica que el fichero correspondiente ya no existe. El directorio raíz de una partición, así como su entrada "..", siempre tienen un número de nodo-i de 2. Esta es la forma como el sistema de ficheros puede identificar a su directorio raíz.

♦ Ejemplo 8.10:

En la Tabla 8.2 se muestra el contenido de un directorio típico.

Número de nodo-i	Nombre del fichero
73	.
38	..
9	ctre
0	fichero_borrado
110	subdirectorio_1
65	prueba.txt

Tabla 8.2: Estructura de un directorio del s5fs



8.7.3 Nodos-i

Cada fichero tiene un nodo-i asociado. La palabra *nodo-i* deriva de *nodo índice*. El nodo-i contiene información administrativa, o metadatos del fichero. Está almacenado en el disco dentro de la lista de nodos-i. Cuando un fichero es abierto, o un directorio está activo, el núcleo copia el nodo-i del disco en memoria principal, en una estructura de datos que también es denominada nodo-i. Esta estructura de memoria principal contiene además otras informaciones adicionales. Como se pueden llegar a confundir, se utilizará el término *nodo-i* para referirse a la estructura de datos (`struct dinode`) almacenada en el disco y el término *nodo-im* para referirse a la estructura de datos (`struct inode`) almacenada en memoria principal. La Tabla 8.3 describe los campos en el nodo-i.

Campo	Bytes	Descripción
di_mode	2	Modo del fichero
di_nlinks	2	Número de enlaces duros al fichero
di_uid	2	<i>uid</i>
di_gid	2	<i>gid</i>
di_size	4	Tamaño del archivo en bytes
di_addr	39	Array de direcciones de los bloques de datos
di_gen	1	Número de generación (se incrementa cada vez que el nodo-i es reutilizado por un fichero nuevo)
di_atime	4	Fecha y hora del último acceso al fichero
di_mtime	4	Fecha y hora de la última modificación del fichero
di_ctime	4	Fecha y hora de la última modificación del contenido del nodo-i

Tabla 8.3: Campos de la estructura *dinode*.

Es importante distinguir entre modificar los contenidos de un nodo-i y modificar el contenido de su fichero asociado. El contenido de un fichero cambia únicamente cuando se escribe, mientras que los contenidos de un nodo-i cambian cuando se modifica el contenido de alguno de sus campos constituyentes. En conclusión, cambiar el contenido de un fichero automáticamente implica un cambio de su nodo-i asociado. Por el contrario los contenidos del i nodo-i pueden cambiar sin que necesariamente se haya modificado los contenidos del fichero.

♦ **Ejemplo 8.11:**

```
[1] Modo del fichero: -r-xr-xr-x
[2] Número enlaces: 1
[3] uid: 503
[4] gid: 204
[5] Tamaño: 5032
[6] Direcciones de bloques del fichero
[7] Número de generación: 2
[8] Ultimo acceso: dic 17 2002 3:30 PM
[9] Ultima modificación: dic 15 2002 9:15 AM
[10] Ultimo cambio del nodo-i : dic 23 2002 9:00 AM
```

Figura 8.15: Ejemplo del contenido de un nodo-i

En la Figura 8.15 se muestra un ejemplo del nodo-i asociado a un determinado fichero. La información contenida en el nodo-i es la siguiente: El campo **[1]** indica que se trata de un fichero regular con permisos de lectura y ejecución para su propietario, los miembros del grupo al que pertenece el propietario y el resto de usuarios, además los bits `S_ISUID`, `S_ISGID` y `S_ISVTX` están sin activar. El campo **[2]** indica que este nodo-i solamente tiene un enlace duro, es decir, un único nombre asignado. El campo **[3]** y el **[4]** dan información sobre el `uid=503` y el `gid=204` del propietario del fichero. El campo **[5]** indica que el fichero tiene un tamaño de 5032 bytes. El campo **[6]** daría información sobre las direcciones de los bloques de datos del fichero y el campo **[7]** indicaría que este nodo-i ha sido reutilizado dos veces por el sistema.

Por otra parte, la última vez que el fichero fue leído por algún usuario (campo **[8]**) fue el 17 de diciembre de 2002 a la 3:30 P.M. Por otra parte, la última vez que el fichero fue escrito por algún usuario (campo **[9]**) fue el 15 de diciembre de 2002 a la 9:15 A.M. Además, el nodo-i fue modificado por última vez (campo **[10]**) el 23 de diciembre de 2002 a las 9:00 A.M. Este campo **[10]** de modificaciones del nodo-i no contabiliza las modificaciones de los campos temporales de última escritura o lectura del fichero.

♦

En UNIX los bloques de datos de un mismo fichero no son contiguos en el disco. Por tanto es fácil aumentar y disminuir el tamaño de un fichero sin la fragmentación de disco inherente a los esquemas de alojamiento contiguos. Obviamente, la fragmentación no es eliminada por completo, puesto que el último bloque de cada fichero puede contener espacio no utilizado. En promedio, cada fichero desperdicia un espacio de medio bloque.

Para implementar este esquema de alojamiento no contiguo el sistema de ficheros debe mantener un mapa de la localización en el disco de cada bloque del fichero. Esta lista está organizada como un array de direcciones de bloques físicos. El tamaño de este array depende del tamaño del fichero. Así, un fichero muy grande puede requerir varios bloques de disco para almacenar este array. Sin embargo, la mayoría de los ficheros son bastante pequeños, y un array grande solamente desperdiciaría espacio. Además, almacenar el array de bloques de disco en un bloque separado incurriría en una lectura extra cuando se accede al fichero, lo que empobrece el rendimiento del sistema.

La solución de UNIX es almacenar una pequeña *lista de direcciones de bloques físicos* en el propio nodo-i, en concreto en el campo `di_addr`, y utilizar bloques extra para ficheros grandes. Esto es muy eficiente para ficheros pequeños, y suficientemente flexible para manejar ficheros grandes.

El campo de 39 bytes `di_addr` se compone de un array de 13 elementos (ver Figura 8.16), cada uno de los cuales almacena un número de bloque físico de 3 bytes. Los elementos 0 a 9 en el array contienen los números de bloques 0 a 9 del fichero, por eso, se les suele denominar como *entradas directas*. Así, para un fichero que conste de 10 bloques o menos, todas las direcciones de estos bloques se encuentran en el propio nodo-i.

El elemento 10 en el array es el número de bloque de un *bloque indirecto simple*, esto es, un bloque que contiene un array de números de bloques. Para acceder a los datos a través de una entrada indirecta, el núcleo debe leer el bloque cuya dirección indica la entrada indirecta y buscar en él la dirección del bloque donde realmente está el dato. El elemento 11 apunta a un *bloque indirecto doble*, que contiene los números de bloques de otros bloques indirectos. Finalmente, el elemento 12 apunta a un *bloque indirecto triple*, que contiene números de bloques de bloques indirectos dobles. En la práctica este método puede extenderse para soportar entradas indirectas cuádruples, quintuples, etc., pero se tiene más que suficiente con una indirección triple.

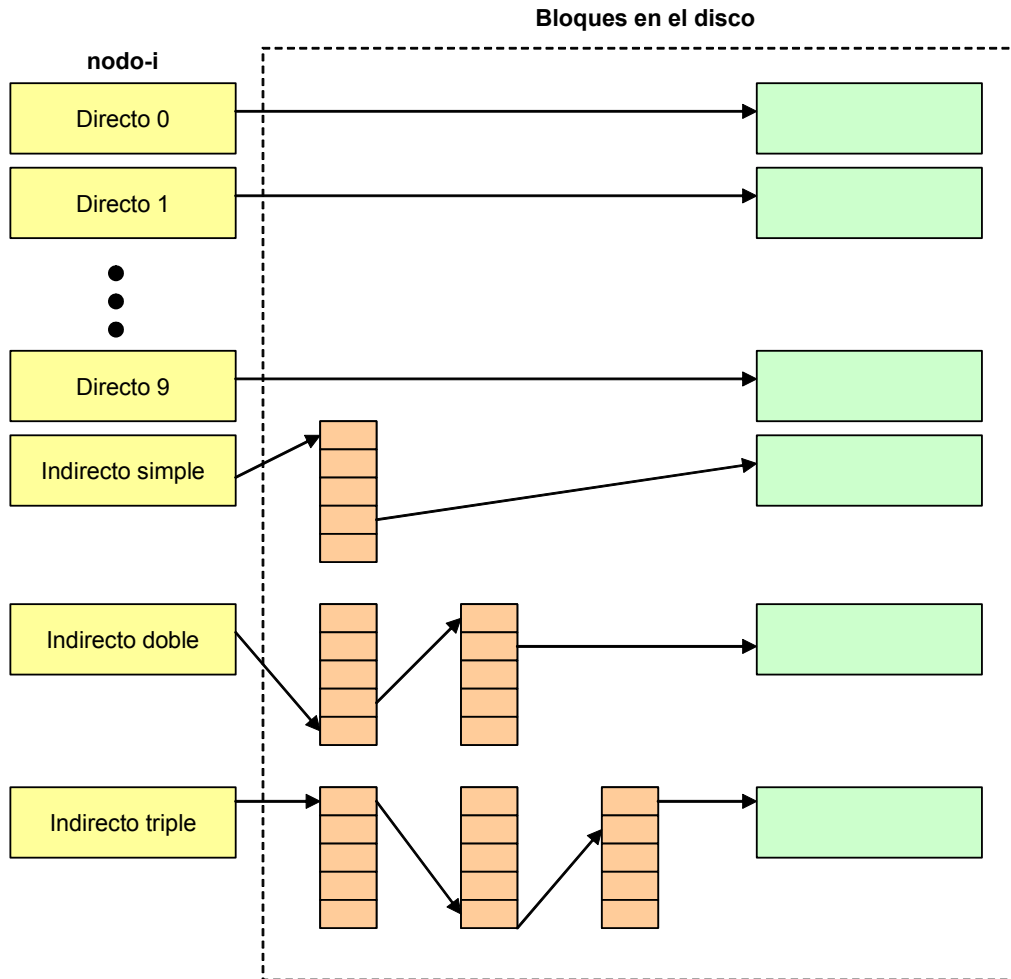


Figura 8.16: Lista de direcciones de bloques físicos almacenada en el campo `di_addr` del nodo-*i*.

♦ **Ejemplo 8.12:**

Supóngase que se dispone de un sistema de archivos con bloques de capacidad $S_B=1\text{Kbyte}$, y que un bloque se direcciona con $n=32$ bits. El espacio de direcciones de bloques es $N=2^{32}=2^2 \cdot 2^{30}=4\text{ Gb}$. Por otra parte, el número de direcciones N_D asociadas a los contenidos de un determinado bloque es:

$$N_D = \frac{S_B}{n}$$

Sustituyendo valores se obtiene:

$$N_D = \frac{1(\text{Kbyte} / \text{bloque})}{32(\text{bits} / \text{direccion})} = \frac{2^{10} \cdot 2^3 (\text{bits} / \text{bloque})}{2^5 (\text{bits} / \text{direccion})} = 256 \frac{\text{direcciones}}{\text{bloque}}$$

En la Tabla 8.4 se resume la capacidad de direccionamiento de los bloques de direcciones de un nodo- i para un sistema de archivos con bloques de $S_B = 1\text{Kbyte}$.

Tipo de entrada	Total de bloques de datos accesibles	Total de bytes accesibles
10 entradas directas	10 bloques de datos	$10 \cdot S_B = 10 \cdot 1\text{Kbytes} = 10\text{ Kbytes}$
1 entrada indirecta simple	1 bloque indirecto simple \rightarrow $N_D = 2^8 = 256$ bloques de datos	$N_D \cdot S_B = 2^8 \cdot 1\text{Kbytes} = 256\text{ Kbytes}$
1 entrada indirecta doble	1 bloque indirecto doble \rightarrow 2^8 bloques indirectos simples \rightarrow $(N_D)^2 = (2^8)^2 = 2^{16} = 65536$ bloques de datos	$(N_D)^2 \cdot S_B = 2^{16} \cdot 2^{10}\text{ bytes} = 2^6 \cdot 2^{20} = 64\text{ Mbytes}$
1 entrada indirecta triple	1 bloque indirecto triple \rightarrow 2^8 bloques indirectos dobles \rightarrow $(2^8)^2$ bloques indirectos simples \rightarrow $(N_D)^3 = (2^8)^3 = 2^{24} = 16777216$ bloques de datos	$(N_D)^3 \cdot S_B = 2^{24} \cdot 2^{10}\text{ bytes} = 2^4 \cdot 2^{30} = 16\text{ Gbytes}$

Tabla 8.4: Capacidad de direccionamiento de los bloques de direcciones de un nodo- i para un sistema de archivos con bloques de $S_B = 1\text{Kbyte}$

♦

Considérese la siguiente notación:

- D_L posición de un byte de un bloque lógico de fichero o desplazamiento en bytes de lectura/escritura con respecto a la posición de inicio de dicho bloque.
- D_D posición de un byte de un bloque físico de fichero o desplazamiento de lectura/escritura en bytes con respecto a la posición de inicio de dicho bloque.
- S_B tamaño de un bloque del disco
- B_L número de bloque lógico de un fichero.
- B_D número de bloque físico.
- E_i entrada i de la tabla de direcciones del nodo- i asociado a D_L

Los procesos accederán a los datos de un fichero indicando D_L . El fichero, por tanto, es visto como una secuencia de bytes que empieza en el byte 0 y llega hasta el byte cuya posición, con respecto al inicial, coincide con el tamaño del fichero menos uno. El núcleo se encarga de transformar las posiciones lógicas D_L de los bytes tal y como las ve el usuario, a posiciones físicas D_D de los bloques del disco.

♦ Ejemplo 8.13:

Supóngase que el campo `di_addr` de un nodo-*i*, que almacena la lista de direcciones de bloques físicos, tiene almacenado el contenido que se muestra en la Figura 8.17. Se supone también que un bloque tiene un tamaño de $S_B = 1\text{Kbytes}$. Si un proceso quiere acceder a un byte que se encuentra en la posición $D_L = 9125$ del fichero. Se desea calcular: (a) El número de bloque lógico B_L del fichero que contiene a D_L . (b) La entrada de la lista de direcciones del nodo-*i* E_i asociada a D_L . ¿A qué número de bloque B_D del disco apunta?. (c) La posición D_D del byte dentro del bloque físico B_D que se corresponde con D_L .

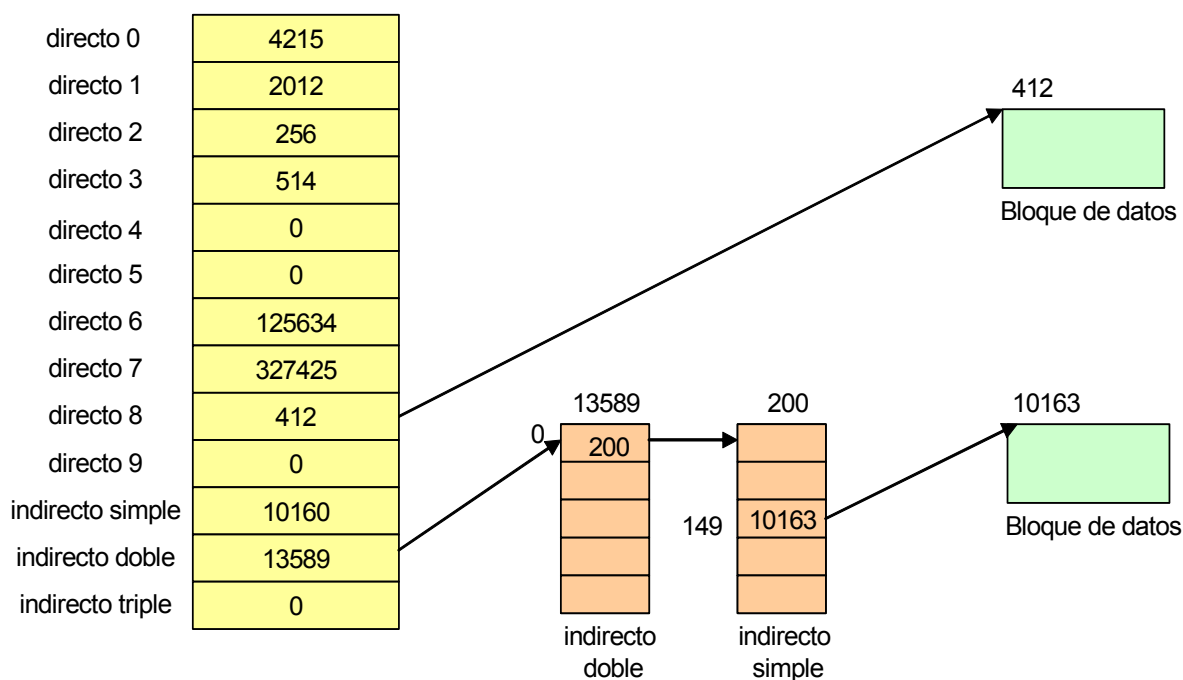


Figura 8.17: Lista de direcciones de bloques físicos almacenada en el campo `di_addr` del nodo-*i*

Solución:

a) Para calcular B_L hay que realizar la siguiente operación:

$$B_L = \frac{D_L}{S_B} - 1 = \frac{9125}{1024} - 1 = 8.9 - 1 = 7.9 \approx 8$$

Siempre hay que aproximar al entero mayor más próximo. Luego $B_L=8$.

b) El bloque lógico $B_L=8$ se corresponde con la entrada $E=8$ (recuérdese que las entradas se comienzan a numerar con el 0) de la tabla de direcciones del nodo-*i*. Dentro de la entrada $E=8$, puesto que es una entrada directa, se encuentra almacenada la dirección de un bloque de datos en el disco, que de acuerdo con la Figura 8.17, es el bloque del disco $B_D=412$.

c) Para calcular el desplazamiento D_D (los bytes del bloque en el disco se enumeran desde 0 a 1023) dentro del bloque $B_D = 412$ del disco que se corresponde con D_L , se realiza el siguiente cálculo:

$$D_D = D_L \bmod(S_B) = 9125 \bmod(1024) = 933 \text{ bytes}$$

Es decir, D_D es el resto de la división entera que tiene como dividendo a D_L y como divisor a S_B . Se ha obtenido que el byte alojado en la posición $D_D = 9125$ alojado en el bloque $B_L = 8$ del fichero se corresponde con el byte alojado en la posición **$D_B = 933$** del bloque $B_B = 412$ del disco.

♦

♦ **Ejemplo 8.14:** Resolver el ejemplo anterior suponiendo ahora que $D_L = 425.000$ bytes.

Solución:

a)

$$B_L = \frac{D_L}{S_B} - 1 = \frac{425000}{1024} - 1 = 415.04 - 1 = 414.04 \approx 415$$

Siempre hay que aproximar al entero mayor más próximo. Luego **$B_L = 415$** .

b) El total de bloques del disco al que se puede acceder con las entradas directas es 10 (bloques $B_L = 0$ a $B_L = 9$). Con la entrada indirecta simple se puede acceder a $2^8 = 256$ bloques (bloques $B_L = 10$ a $B_L = 265$). Mientras que con la entrada indirecta doble se puede acceder a $(2^8)^2 = 65536$ bloques (bloques $B_L = 266$ a $B_L = 65801$). En conclusión, el bloque $B_L = 415$ estará direccionado por la entrada indirecta doble **$E = 11$** de la tabla de direcciones del nodo-i.

De acuerdo con la Figura 8.17, el número de bloque físico que contiene la entrada indirecta doble es el $B''_D = 13589$, que contiene las direcciones de los bloques con entradas indirectas simples B'_D .

La entrada 0 del bloque indirecto doble B''_D permite el acceso a los bloques $B_L = 266$ a $B_L = 521$, ya que cada entrada actúa de indirección simple y da acceso a 256 bloques de datos. La entrada 0 del bloque B''_D contiene la dirección del bloque indirecto simple $B'_D = 200$.

Dentro del bloque indirecto simple $B'_D = 200$, la entrada que interesa es la diferencia entre $B_L = 415$ y $B_L = 266$ (bloque lógico inicial al que da acceso la entrada 0 del bloque indirecto doble B''_D). Es decir, $415 - 266 = 149$. Según la Figura 8.17 la entrada 149 del bloque del disco B'_D contiene el número **$B_D = 10163$** , que es el bloque del disco donde se encuentra el dato que se busca.

c) Para calcular el byte D_D (los bytes del bloque en el disco se enumeran desde 0 a 1023) dentro de $B_D = 10163$ que se corresponde con el byte D_L del fichero, se realiza el siguiente cálculo:

$$D_D = D_L \bmod(S_B) = 425000 \bmod(1024) = 40$$

Es decir, D_D es el resto de la división entera que tiene como dividendo a D_L y como divisor a S_B .

Se ha obtenido que el byte $D_L=425000$ del fichero alojado en el bloque número $B_L= 415$ del fichero se corresponde con el byte $D_D= 40$ del bloque número $B_D=10163$ del disco.

♦

Como se puede apreciar en la Figura 8.17 algunas de las entradas de la lista de direcciones de bloques físicos pueden contener el valor 0. Esto significa que no referencian a ningún bloque del disco y que los bloques lógicos correspondientes del fichero no tienen datos. Esta situación se da cuando se crea un fichero y nadie escribe en los bytes correspondientes a estos bloques, por lo que permanecen en su valor inicial 0. Al no reservar el sistema bloques de disco para estos bloques lógicos, se consigue un ahorro de espacio en disco.

♦ **Ejemplo 8.15:**

Supóngase que se crea un fichero y sólo se escribe 1 byte en la posición 1048276, esto significa que el fichero tiene un tamaño de 1 Mbyte. Si el sistema reservase bloques de discos para este fichero en función de su tamaño y no en función de los bloques lógicos que realmente tiene ocupados, el fichero ocuparía 1024 bloques de disco en lugar de 1, como en realidad ocupa.

♦

8.7.4 El superbloque

El *superbloque* contiene metadatos sobre el propio sistema de ficheros. Hay un único superbloque por cada sistema de ficheros, y reside al comienzo del sistema de ficheros en el disco, a continuación del área de arranque. El núcleo lee el superbloque cuando monta el sistema de ficheros y lo almacena en memoria hasta que el sistema de ficheros es desmontado. El superbloque contiene básicamente información administrativa y estadística del sistema de archivos, como por ejemplo:

- Tamaño en bloques del sistema de ficheros.
- Tamaño en bloques de la lista de nodos-i.
- Número de bloques libres y nodos-i libres.

- Comienzo de la lista de bloques libres.
- Lista parcial de nodos-i libres.

Puesto que el sistema de ficheros puede contener muchos nodos-i libres o bloques de disco libres, es poco práctico mantener ambas listas libres completamente en el superbloque. En el caso de los nodos-i, el superbloque mantiene una lista parcial de nodos-i libres. Un nodo-i se considera que está libre cuando su campo `di_mode` contiene el valor 0. Cuando la lista se vacía, el núcleo busca en el disco nodos-i libres para rellenar la lista comenzando por el *nodo-i recordado* y en sentido ascendente de número de nodo-i. El *nodo-i recordado* se define como el nodo-i de mayor número de nodo-i que se ha almacenado en la lista parcial de nodos-i libres desde la última vez que ésta fue rellenada.

Esta aproximación no es posible para la lista de bloques libres, puesto que no existe forma de determinar si un bloque está libre examinando su contenido. Por lo tanto, el sistema debe mantener una lista completa de todos los bloques libres en el disco.

En la Figura 8.18 se muestra un ejemplo de lista de bloques libres que se extiende a través de varios bloques de disco. El superbloque contiene la primera parte de la lista y añade o elimina bloques de su cola. El primer elemento en esta lista apunta al *bloque a* que contiene la siguiente parte de la lista. Asimismo el primer elemento de la lista contenida en el *bloque a* apunta al *bloque b* que contiene la siguiente parte de la lista, y así sucesivamente.

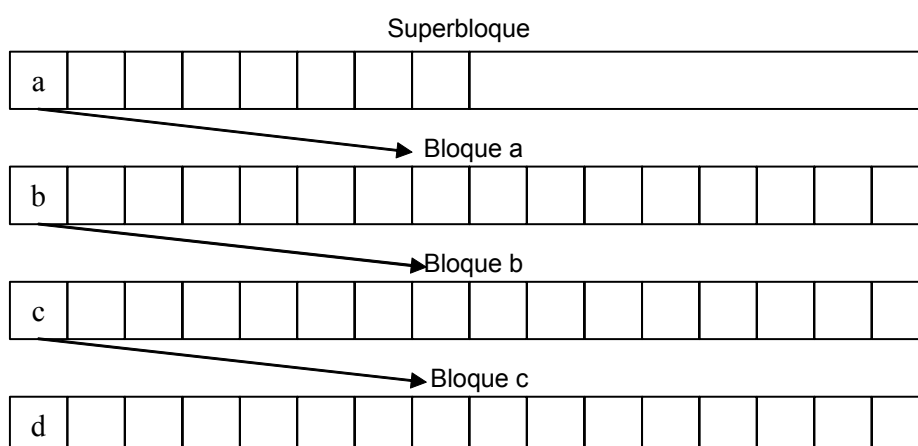


Figura 8.18: Lista de bloques libres en s5fs

Si en algún momento, la rutina de asignación de bloques descubre que la lista de bloques libres en el superbloque únicamente contiene un único elemento. El valor almacenado en dicho elemento es el número del bloque que contiene la siguiente parte de la lista de bloques libres (*bloque a* en la Figura 8.18). Copia el contenido de este bloque dentro del superbloque, y dicho bloque pasa a estar libre. Esto tiene la ventaja de que el espacio requerido para almacenar la lista de bloques libres depende directamente de la cantidad del espacio libre de la partición. Para un disco prácticamente lleno, no se necesita desperdiciar espacio para almacenar la lista de bloques libres.

8.7.5 Organización en la memoria principal del s5fs

Un nodo-i es el principal objeto dependiente del sistema de ficheros *s5fs*. Es la estructura de datos privada asociada de un nodo-v *s5fs*. Como se mencionó con anterioridad, los nodos-i en memoria principal (nodo-im) son diferentes de los nodos-i en el disco.

La estructura `inode` representa a un nodo-im, contiene una copia de una estructura `dinode`, es decir, de un nodo-i en el disco. Existe una pequeña diferencia y es que el campo `di_addr` de un nodo-im contiene una array de 13 elementos de 4 bytes cada uno, en vez de 3 bytes, con el fin de mejorar el rendimiento del sistema. Asimismo la estructura `inode` de un nodo-im contiene algunos campos adicionales para almacenar entre otras las siguientes informaciones:

- El nodo-v del fichero.
- El identificador de dispositivo de la partición que contiene el fichero.
- El número de nodo-i del fichero.
- Punteros para mantener al nodo-im en una *lista de nodos-im libres*.
- Punteros para mantener al nodo-im en una *cola de dispersión*.
- Número de bloque del último bloque leído o escrito.

El núcleo organiza los nodos-im del *s5fs* mediante una estructura muy similar a la caché de buffers de bloques. Es decir, mantiene varias colas de dispersión basadas en los números de nodos-i que le permite localizar rápidamente a los nodos-im cuando los necesite. Asimismo mantiene una lista de nodos-im libres.

♦ Ejemplo 8.16:

En la Figura 8.19 se representa una posible organización de los nodos-mi que consta de cuatro colas de dispersión. La cola hash nº 0 contiene los nodos-im marcados con los números de nodo-i 268, 40 y 1056, obsérvese que todos estos números cumplen la regla

$$N^{\circ} \text{ nodo-i} \% 4 = 0$$

es decir, al dividir el número de nodo-i por 4 su resto es 0. Se observa que las otras colas siguen reglas similares para los nodos-im que contienen.

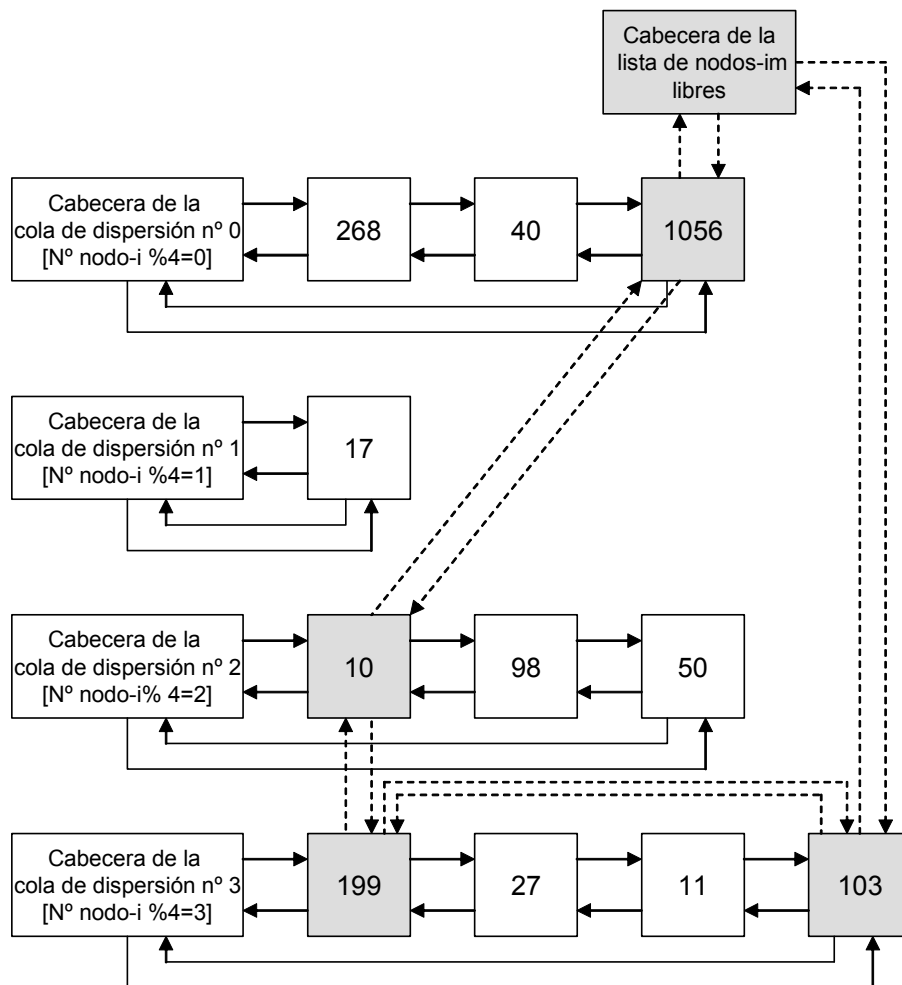


Figura 8.19: Organización de los nodos-im del s5fs (se ha resaltado la lista de nodos-im libres)

Asimismo en la Figura 8.20 se representa la lista de nodos-im libres. Se observa que forman parte de esta lista los nodos-im marcados con los números de nodo-i 1056, 10, 199 y 103.

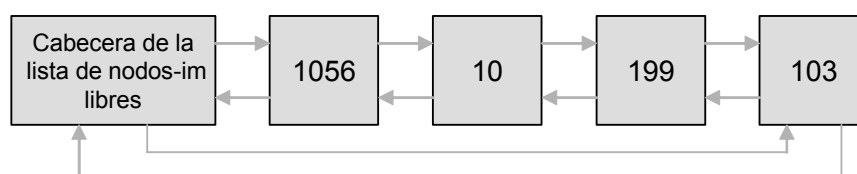


Figura 8.20: Lista de nodos-im libres

◆

8.7.5.1 Búsqueda de nodos-im

La función `lookuppn()` de la capa independiente del sistema de ficheros realiza el análisis de la ruta de acceso a un fichero. Como se describió en la sección 8.6.7.1, esta función va recorriendo cada componente de la ruta, y para cada una de ellas invoca a la operación `VOP_LOOKUP`. Cuando busca un directorio en un sistema de ficheros *s5fs*, esta operación se traduce en una llamada a la función `s5lookup()`, que en primer lugar comprueba la *caché de búsqueda de nombres en directorios*. En caso de un fallo en la caché, lee el directorio en busca de la entrada para el nombre de fichero especificado.

Si el directorio contiene una entrada válida para el fichero, `s5lookup()` obtiene el número de nodo-i de la entrada. Entonces llama al algoritmo `iget()` para localizar el nodo-im en la cola de dispersión adecuada. Si el nodo-im no está en la cola de dispersión, `iget()` le asigna un nodo-im, y lo inicializa leyendo el nodo-i del disco. Mientras copia los campos del nodo-i en el disco al nodo-im, expande los 13 elementos del array almacenado en `di_addr[]` de 3 bytes a 4 bytes cada uno. Además coloca el nodo-im en la cola de dispersión apropiada. También inicializa el nodo-v, configurando su campo `v_op` para que apunte al `vfs` al cual el fichero pertenece. Finalmente, devuelve un puntero al nodo-i para `s5lookup()`. Asimismo `s5lookup()`, al finalizar, devuelve un nodo-v a `lookuppn()`.

8.7.5.2 Asignación y recuperación de nodos-im

Cuando un fichero es accedido, sus páginas son copiadas en memoria. Estas páginas pueden ser localizadas a través de la *lista de páginas del nodo-v*, y accedidas a través de su campo `v_page`. Cuando el fichero se hace inactivo, algunas de sus páginas pueden todavía permanecer en memoria.

Un nodo-im permanece activo si su nodo-v tiene el contador de referencias `v_count` distinto de cero. Cuando el contador llega a cero, el código independiente del sistema de ficheros invoca a la operación `VOP_INACTIVE`, para liberar al nodo-v y sus objetos de

datos privados (en este caso el nodo-im). Cuando se libera el nodo-im, el núcleo comprueba la lista de páginas del nodo-v. El núcleo coloca al nodo-im delante de la lista de nodos-im libres si la lista de páginas está vacía, y lo coloca al final de la lista de nodos-im libres si cualquier página se encuentra todavía en memoria. Con el tiempo, si el nodo-im permanece inactivo, el sistema de paginación liberará sus páginas.

Cuando el algoritmo `iget()` no puede localizar un nodo-im en su cola de dispersión asociada, entonces utiliza el nodo-im más cercano a la cabecera de la lista de nodos-im libres, que es borrado de esta lista. Si este nodo-im tiene todavía sus páginas en memoria, `iget()` lo devuelve al final de la lista de nodo-im libres e invoca al asignador de memoria del núcleo para asignar una nueva estructura nodo-im.

Debido a la semejanza entre la organización de los nodos-im y la caché de buffers podría pensarse en gestionar la lista de nodos-im libres de la misma forma que la lista de buffers libres, es decir, mediante una política del tipo LRU. De hecho así se hacía en las distribuciones clásicas de UNIX como por ejemplo SVR3. Sin embargo, usar una política LRU en la lista de nodos-im libres empeora el rendimiento del sistema, ya que como se ha visto ciertos nodos-im libres son más útiles que otros.

8.7.6 Análisis del *s5fs*

Una de las características más relevantes del sistema de ficheros *s5fs* es la simplicidad de su diseño. Esta simplicidad, sin embargo, acarrea problemas de seguridad, rendimiento y funcionalidad.

El mayor problema de seguridad proviene del superbloque que contiene información vital sobre el sistema de ficheros, tales como la lista de bloques libres y el tamaño de la lista de nodos-i libres. Cada sistema de ficheros contiene una única copia de su superbloque. Si la copia está corrupta, todo el sistema de ficheros no podrá ser utilizado.

El rendimiento se deteriora por varias razones. En primer lugar *s5fs* agrupa todos los nodos-i en la lista de nodos-i, a continuación del superbloque. El espacio restante del sistema de ficheros contiene los bloques de datos de los ficheros. Acceder a un fichero requiere leer el nodo-i y después los datos del fichero, así esta segregación produce una mayor búsqueda en el disco entre las dos operaciones y por lo tanto incrementa los tiempos de E/S. Los nodos-i se alojan aleatoriamente, con ningún intento por agrupar los nodos-i relacionados como por ejemplo aquellos de los ficheros que se encuentran en el mismo directorio. Por lo tanto una operación que acceda a todos los ficheros en un

directorio como por ejemplo el comando `$ ls -l` causará un patrón de acceso a disco aleatorio.

En segundo lugar, el alojamiento de los bloques en el disco es también poco óptima. Cuando el sistema de ficheros es creado por primera vez, *s5fs* configura la lista de bloques libres de forma óptima para que los bloques sean alojados en un orden consecutivo rotacional. Sin embargo, cuando los ficheros son creados y posteriormente borrados, los bloques retornan a la lista en un orden aleatorio. Después de un cierto tiempo de utilización del sistema de ficheros, el orden de los bloques en la lista llega a ser completamente aleatorio. Esta circunstancia relantiza las operaciones de acceso secuencial a los ficheros, porque los bloques consecutivos pueden estar muy alejados en el disco.

En tercer lugar, el tamaño del bloque en el disco es otro aspecto que afecta al rendimiento. SVR2 utilizaba un bloque de 512 bytes, SVR3 lo elevó hasta 1024 bytes. Incrementar el tamaño del bloque permite alojar más datos que pueden ser leídos en un único acceso al disco, con lo que se mejora el rendimiento. Al mismo tiempo, se desperdicia más espacio en el disco, puesto que, en promedio, cada fichero desperdicia la mitad de un bloque. Este hecho pone de manifiesto la necesidad de un sistema más flexible de asignación de espacio para los ficheros.

Finalmente, existen algunas limitaciones de funcionalidad. En un sistema *s5fs* el tamaño de los nombres de los ficheros está restringido a 14 caracteres. Esta limitación quizás no importaba mucho hace algunos años, pero para un sistema operativo viable comercialmente y potente, tal restricción es inaceptable. Varias aplicaciones automáticamente generan nombres de ficheros, a menudo añadiendo extensiones adicionales a los ficheros, y se ven forzados a hacerlo eficientemente dentro de los 14 caracteres. Asimismo, el límite de 65535 nodos-i por cada sistema de ficheros es también bastante restrictivo.

Todos estos problemas condujeron al desarrollo de un nuevo sistema de ficheros en el UNIX BSD. Conocido como el sistema de ficheros rápido (FFS), que fue incorporado por primera vez en el BSD4.2.

TEMA 9

GESTION DE MEMORIA EN UNIX

9.1 INTRODUCCION

La *memoria principal* o *memoria física* de una computadora es típicamente una memoria de acceso aleatorio (RAM) cuyo tiempo de acceso es mucho más pequeño que el de la memoria secundaria (discos duros, máquinas en red,...). Sin embargo la memoria principal tiene un coste mucho mayor y una capacidad mucho más pequeña que la memoria secundaria. En definitiva la memoria principal es un recurso limitado muypreciado.

El sistema operativo debe administrar toda la memoria física y asignarla tanto a los subsistemas del núcleo como a los programas de usuario. Cuando el sistema arranca, el núcleo reserva parte de la memoria principal para su código y sus estructuras de datos estáticas. Esta parte nunca es liberada y por lo tanto no se encuentra disponible para ningún otro propósito. El resto de la memoria principal es administrada dinámicamente, el núcleo asigna porciones de memoria a sus numerosos clientes (procesos y subsistemas del núcleo), y la libera cuando ya no la necesitan.

La parte del núcleo responsable de gestionar la memoria principal es el *subsistema de administración de memoria* que interactúa fuertemente con la *unidad de administración de memoria* (MMU¹), que funcionalmente se sitúa entre la CPU y la memoria principal. La arquitectura de la MMU tiene un fuerte impacto sobre el diseño del sistema de administración de memoria del núcleo. La tarea principal de la MMU es la traducción de direcciones virtuales. La mayoría de los sistemas implementan los mapas de traducción de direcciones utilizando tablas de páginas, TLBs², o ambos.

¹ MMU es el acrónimo derivado del término inglés *Memory Management Unit* (MMU)

² TLB es el acrónimo derivado del término inglés *Translation Lookaside Buffer*. Un TLB es una memoria caché asociativa de traducción de direcciones virtuales accedidas recientemente.

Las primeras implementaciones de UNIX (versión 7 y anteriores) se ejecutaban sobre una máquina PDP-11, que tenía una arquitectura de 16 bits con un espacio de direcciones de 64 Kilobytes. Algunos modelos soportaban espacios de direcciones y de datos independientes, pero esto todavía restringía el tamaño de un proceso a 128 Kilobytes. Los mecanismos de administración de memoria estaban restringidos a una política de *intercambio* (Ver Figura 9.1). Los procesos eran cargados por completo en memoria de forma contigua. Solamente un pequeño número de procesos podían estar cargados al mismo tiempo en memoria principal. Si otro proceso tenía que ser ejecutado, uno de los procesos cargados en memoria tenía que ser intercambiado a memoria secundaria, en concreto a una partición predefinida en el disco duro denominada *partición o área de intercambio*. El *espacio de intercambio* era asignado en esta partición para cada proceso en el momento de la creación del proceso, así se garantizaba su disponibilidad cuando fuese necesitado.

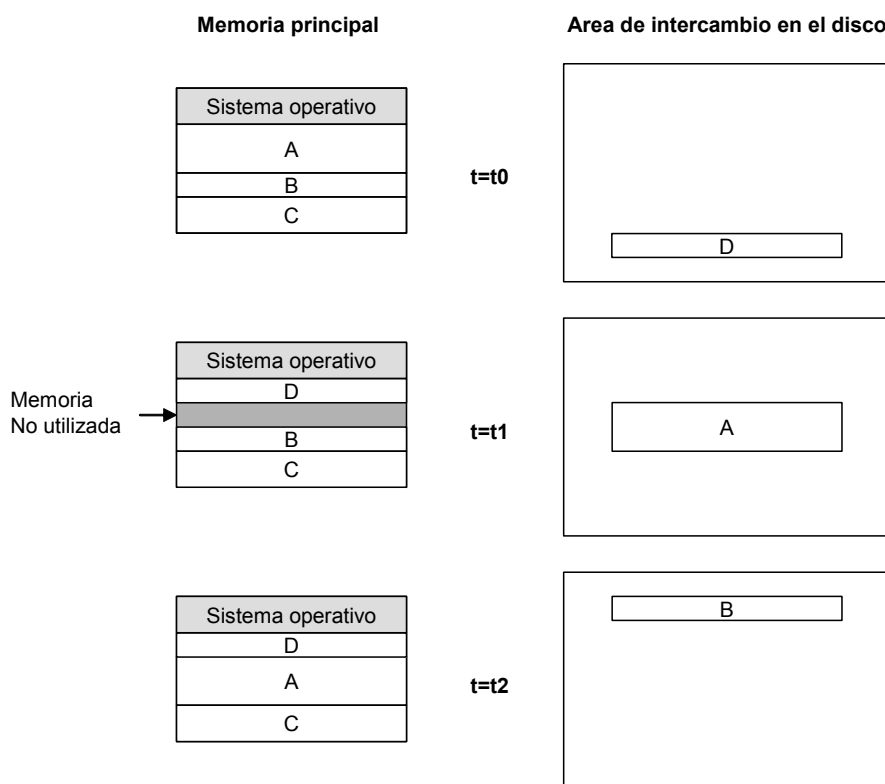


Figura 9.1: Administración de memoria basada en intercambio

La política de gestión de memoria mediante *demanda de página* hizo su aparición en UNIX con la aparición de la máquina VAX-11/780 en 1978, que tenía una arquitectura de 32 bits, un espacio direccionable de 4 Gigabytes, y soporte hardware para la realización de demanda de páginas. BSD3 fue la primera distribución de UNIX que soportaba

demanda de páginas. A mediados de los 80, todas las distribuciones de UNIX utilizaban demanda de página como principal política de administración de memoria principal, quedando la política de intercambio relegada a un segundo plano.

En un sistema con política de gestión de memoria mediante *demanda de página*, la memoria principal es dividida en bloques de tamaño fijo denominados *páginas físicas* o *marcos de página*. Asimismo los procesos también son divididos en páginas, que son cargadas en los marcos de página conforme son requeridas. Varios procesos pueden estar activos al mismo tiempo, y la memoria física puede contener solo algunas de las páginas de cada proceso (ver Figura 9.2).

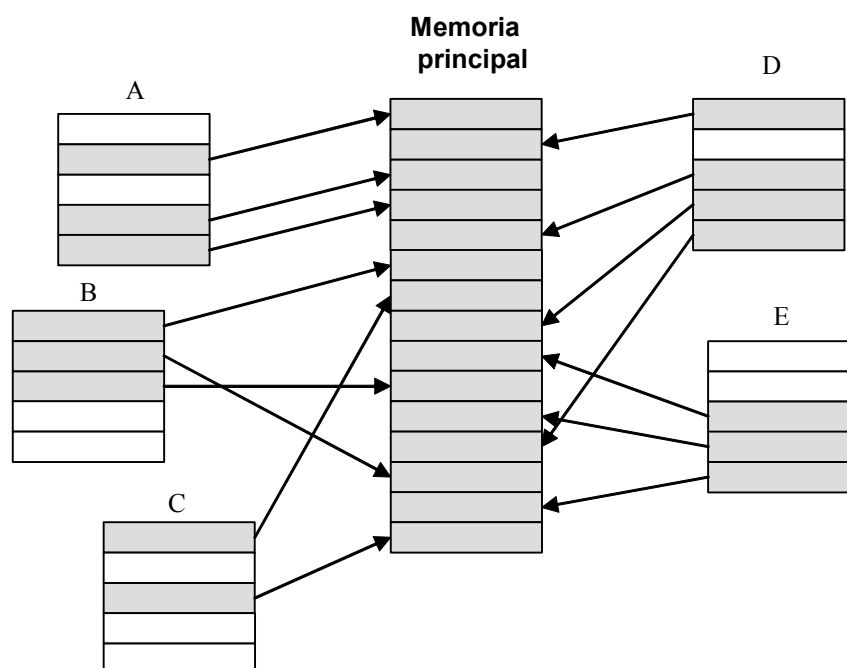


Figura 9.2: La memoria física almacena unas pocas páginas de cada proceso

Un esquema de demanda de páginas posee las siguientes ventajas con respecto a un esquema de intercambio:

- El tamaño de un programa está limitado sólo por la memoria virtual, para una máquina de 32 bits este tamaño puede ser cercano a los 4 Gigabytes.
- El arranque de los programas es rápido puesto que no es necesario que todo el programa se encuentre cargado en memoria principal para comenzar a ejecutarse.

- Muchos programas pueden estar cargados en memoria principal al mismo tiempo, puesto que solo unas pocas páginas de cada programa necesitan estar en memoria en un cierto instante.
- Mover páginas dentro y fuera de la memoria principal es mucho menos costoso que intercambiar procesos enteros o segmentos.

La base teórica que justifica la política de demanda de página es el hecho de que los programas cumplen con el *principio de localidad* es decir, los procesos tienden a ejecutar instrucciones que se encuentran cercanas en el código del mismo, como por ejemplo bucles. A partir del principio de localidad surge el concepto de *conjunto de trabajo* que es el conjunto de páginas que el proceso ha referenciado en sus últimos n accesos a memoria. El número n indica la ventana del conjunto de trabajo.

Puesto que UNIX es un sistema de *memoria virtual*, las páginas que son lógicamente contiguas en el espacio de direcciones virtual de un proceso no necesitan estar adyacentes físicamente en la memoria principal. Las direcciones del programa son *virtuales* y son divididas por la MMU en un número de página virtual y un desplazamiento desde el origen de la página. La MMU, junto con el sistema operativo, traduce el número de página virtual en el espacio de direcciones del programa a un *número de marco de página* para acceder a la localización adecuada. Cuando un proceso referencia a una página que no pertenece al conjunto de trabajo se produce una excepción denominada *fallo de página* en su tratamiento el núcleo realiza principalmente las siguientes acciones:

- 1) Suspender la ejecución de la instrucción en curso.
- 2) Buscar la página en memoria secundaria.
- 3) Cargar la página en un marco de página.
- 4) Reiniciar la instrucción que se estaba ejecutando en ese momento.

Cuando se necesita cargar una página en memoria principal y no existen marcos libres, el núcleo debe reemplazar una página que se encuentra actualmente en memoria. La política de sustitución de páginas hace referencia a como el núcleo decide que página en memoria debe ser reemplazada, típicamente se suele usar una política del tipo LRU. La página reemplazada es almacenada en un área de intercambio. Si una página que ha sido salvada en el área de intercambio es de nuevo accedida, el núcleo manipula el fallo de página cargándola en memoria principal desde el área de intercambio. Para poder hacerlo, debe mantener alguna clase de *mapa de intercambio* que describa la

localización de todas las páginas intercambiadas a dicha área. Si esta página debe ser intercambiada fuera de la memoria principal de nuevo, será salvada en el área de intercambio solamente si sus contenidos son diferentes de la copia salvada.

Por otra parte, otra política de gestión de memoria es la *segmentación*. Esta técnica divide el espacio de direcciones de un proceso en varios *segmentos* o *regiones*. Cada dirección en el programa consiste de un identificador del segmento y un desplazamiento desde la base del segmento. Cada segmento puede tener protecciones individuales (lectura/escritura/ejecución) asociadas con él. Los segmentos son cargados en memoria física de forma contigua, y cada segmento está descrito por un descriptor que contiene la dirección física en la que es cargado (su dirección base), su límite o tamaño y su protección. El hardware comprueba los límites del segmento en cada acceso a memoria, para prevenir que el proceso pueda corromper a un segmento adyacente. La unidad de carga e intercambio de un programa es el segmento en vez de todo el programa como ocurre con la política de intercambio.

La segmentación puede también ser combinada con la paginación para suministrar un mecanismo de administración de memoria híbrido que resulta bastante flexible. En tales sistemas, los segmentos no necesitan estar físicamente contiguos en memoria. Cada segmento tiene su propio mapa de traducción, que traduce desplazamientos dentro del segmento a posiciones de memoria física. La arquitectura Intel 80x86 (es decir, Intel 80386, Intel 80486, y Pentium), por ejemplo, soporta este modelo.

Los programadores típicamente piensan en el espacio de direcciones virtual de un proceso como formado por las regiones de código, datos y pila, y la noción de segmentos traduce bien esta perspectiva. Aunque muchas versiones de UNIX explícitamente definen estas tres regiones, estas son usualmente soportadas como una abstracción a alto nivel compuestas de un conjunto de páginas virtuales contiguas y no como segmentos reconocidos por el hardware. La segmentación no ha sido muy popular en las distribuciones más utilizadas de UNIX.

Por su importancia conceptual y mayor sencillez este tema está dedicado principalmente a describir la política de gestión de memoria mediante *demand a de página* implementada en un sistema UNIX clásico como SVR3. En primer lugar se analizan las estructuras de datos del núcleo necesarias para implementar la política de demanda de página. En segundo lugar se describe como se realizan las llamadas al sistema `fork` y `exec` en un sistema con demanda de página. En tercer lugar se describe la transferencia de páginas de memoria principal al área de intercambio. En cuarto lugar se describe la

manipulación que realiza el núcleo de los fallos de página. Por último, disponiendo ya de todos los elementos necesarios para su adecuada comprensión, se ofrece una explicación del cambio de modo de un proceso desde el punto de vista de la gestión de memoria. Asimismo se describe la localización en memoria del área U de un proceso.

9.2 POLITICA DE DEMANDA DE PÁGINAS EN EL SVR3

9.2.1 Estructuras de datos asociadas a la gestión de memoria mediante demanda de páginas

El núcleo mantiene fundamentalmente cuatro tipos de estructuras de datos para implementar la política de memoria mediante demanda de página: las tablas de páginas, las tablas de descriptores de bloques de disco (tabla *dbd* para abreviar), la tabla de datos de los marcos de página (tabla *dmp* para abreviar) y la tabla de intercambio.

El núcleo asigna espacio para la tabla *dmp* una vez durante el tiempo de vida del sistema por el contrario asocia páginas de memoria para las otras estructuras dinámicamente.

9.2.1.1 Relación entre regiones y páginas: Tablas de páginas

El concepto de *región* es una abstracción de alto nivel independiente de las políticas de administración de memoria implementadas por el sistema operativo. Como ya se estudió en el Tema 2, una *región* es un subconjunto o área de direcciones contiguas de memoria virtual. En cualquier programa se pueden distinguir al menos tres regiones: la región de código o texto, la región de datos y la región de pila.

Cada proceso tiene asignada una *tabla de regiones por proceso*, cada una de sus entradas contiene entre otras informaciones la dirección virtual de comienzo de una región DIR_{V0} asociada al proceso y un puntero que señala a una entrada de la *tabla de regiones*.

Por otra parte, en una arquitectura que trabaje con páginas, *cada región* es dividida en múltiples páginas de tamaño S_p . De esta forma cada entrada de la *tabla de regiones* contiene entre otras informaciones, un puntero a una *tabla de páginas*. Es decir, hay una tabla de páginas por cada entrada de la tabla de regiones. El núcleo almacena las tablas de páginas en memoria principal y accede a ellas como a cualquier otra de sus estructuras de datos.

Cada entrada i de una tabla de páginas contiene los siguientes campos:

- DIR_{F0} , *dirección física de inicio* de una página.
- *Edad*, que se utiliza para indicar cuanto tiempo lleva la página perteneciendo al conjunto de trabajo de un proceso
- *Copiar al escribir*, este campo consta de un único bit, que se chequea en la llamada al sistema `fork`, indica que el núcleo debe crear una nueva copia de la página si un proceso modifica su contenido, en caso contrario se compartirá.
- *Modificada*, este campo consta de un único bit que se activa si un proceso ha modificado recientemente el contenido de la página.
- *Referenciada*, este campo consta de un único bit que se activa si un proceso ha referenciado a la página recientemente.
- *Valida*, este campo consta de un único bit que se activa si el contenido de una página es legal, pero la referencia a dicha página no es necesariamente ilegal si este bit está sin activar. Este bit está desactivado cuando la página no pertenece al conjunto de trabajo del proceso o bien no tienen memoria física asignada.
- *Bits de protección*, que configuran los permisos de acceso (lectura, escritura, ejecución) de la página.

En general, el núcleo es el encargado de manipular los campos de *valida*, *copiar al escribir* y *edad*, mientras que el hardware se encarga de los campos de *referenciada* y *modificada*.

Para acceder a una dirección virtual DIR_V contenida en una determinada región, una forma de hacerlo es especificando la dirección virtual de comienzo DIR_{V0} de la región y el desplazamiento relativo DES_V dentro de la misma. Se verifica la siguiente relación:

$$DIR_V = DIR_{V0} + DES_V \quad (1)$$

De forma análoga, para acceder a una dirección física DIR_F en memoria principal asociada a una determinada página, una forma de hacerlo, es especificar la dirección física de comienzo DIR_{F0} de la página y el desplazamiento relativo DES_F dentro de la misma. Se verifica la siguiente relación:

$$DIR_F = DIR_{F0} + DES_F \quad (2)$$

Por otra parte, conocido el desplazamiento relativo DES_V dentro de una región asociada a un proceso, la tabla de páginas asociada a dicha región y el tamaño de una página S_P , es posible calcular la entrada i de dicha tabla de páginas que le corresponde mediante la siguiente expresión:

$$i = \text{floor}\left(\frac{DES_V}{S_P}\right) \quad (3)$$

La función matemática $\text{floor}(X)$ redondea X hacia el entero más cercano a menos infinito, por ejemplo, $\text{floor}(2.1)=2$, $\text{floor}(2.5)=2$, $\text{floor}(2.8)=2$.

Si se conoce la entrada i , se obtiene de forma inmediata la dirección física DIR_{F0} de comienzo de la página donde se va a encontrar la dirección física DIR_F asociada a la dirección virtual DIR_V .

Para calcular DIR_F mediante la expresión (2), sería necesario calcular previamente el desplazamiento relativo DES_F dentro de la página. Se utiliza la siguiente expresión:

$$DES_F = DES_V \bmod S_P = DES_V \% S_P \quad (4)$$

Es decir, DES_F es el resto de la división entera que tiene como dividendo a DES_V y como divisor a S_P .

♦ Ejemplo 9.1:

En la Figura 9.3 se muestra la asignación de memoria física de un proceso A, que desea acceder a la dirección virtual expresada en decimal $DIR_V = 68432$. Supuesto que el tamaño de página es $S_P = 1\text{Kbytes}$. Calcular la dirección física DIR_F asociada a DIR_V .

Solución:

De acuerdo con la *tabla de regiones por proceso* del proceso A representada en la Figura 9.3, la dirección DIR_V hace referencia a una posición de la región de pila, ya que ésta comienza en la dirección virtual $DIR_{V0} = 64\text{K} = 64 \cdot 2^{10} = 65536$. Si se supone que el crecimiento de la pila se realiza hacia las direcciones virtuales más altas, entonces el desplazamiento relativo DES_V asociado a DIR_V desde el comienzo DIR_{V0} de la región de pila se calcularía, despejando DES_V de la ecuación (1) de la siguiente forma:

$$DES_V = DIR_V - DIR_{V0} = 68432 - 65536 = 2896$$

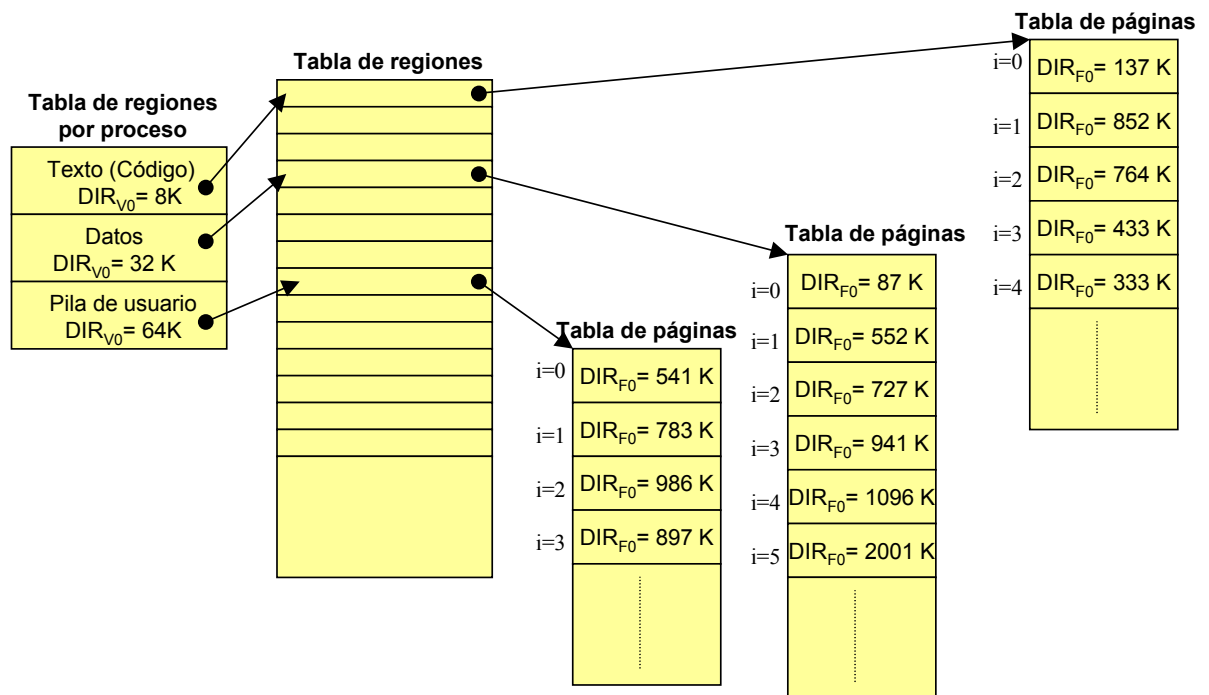


Figura 9.3: Asignación de memoria física de un proceso A.

Por otra parte, la entrada de la *tabla de regiones por proceso* que contiene la región de pila del proceso A, apunta a una entrada de la *tabla de regiones*, que entre otras informaciones, contiene la dirección física de memoria principal donde comienza la *tabla de páginas* asociada a dicha región. De la Figura 9.3 es inmediato identificar la *tabla de páginas* asociada a la región de pila del proceso A.

Hay que calcular la entrada i de dicha tabla de páginas para conocer la dirección DIR_{F0} de comienzo de la página donde se va a encontrar la dirección física DIR_F asociada a la dirección virtual DIR_V . Para ello se utiliza la expresión (3):

$$i = \text{floor}\left(\frac{DES_V}{S_p}\right) = \text{floor}\left(\frac{2896}{1024}\right) = \text{floor}(2.82) = 2$$

De acuerdo con la Figura 9.3, la entrada $i=2$ de la tabla de páginas asociada a la región de pila del proceso A, indica que $DIR_{F0}=986K$. A partir de la expresión (4) se calcular el desplazamiento relativo DES_F dentro de dicha página:

$$DES_F = 2896 \bmod 1024 = 848$$

Finalmente el cálculo de la dirección física DIR_F expresada en decimal asociada a DIR_V , se realiza utilizando la expresión (2):

$$DIR_F = 986K + 848 = 986 \cdot 1024 + 848 = 1010512$$



Por último, comentar que la generación y el mantenimiento de las tablas de páginas dependen fuertemente de la computadora y de la distribución del sistema UNIX que se considere.

9.2.1.2 *Tabla de descriptores de bloques de disco*

Cada una de las *tabla de páginas* tiene asignada una *tabla de descriptores de bloques de disco* (tabla *dbd*). El número de entradas de una tabla *dbd* es igual al número de entradas de la tabla de páginas a la que está asignada.

La entrada *i* de una tabla *dbd* contiene información sobre la copia en memoria secundaria de la página a la que hace referencia la entrada *i* de la tabla de páginas a la que está asociada. Se distinguen los siguientes campos:

- *Dispositivo de intercambio*. Es el número que identifica al dispositivo lógico de memoria secundaria donde se encuentra la copia de la página.
- *Número de bloque* del dispositivo de intercambio donde se almacena la copia de la página.
- *Tipo*. Este campo permite al núcleo conocer donde se encuentra alojada una página en memoria secundaria: en un área de intercambio (*tipo=disco*) o en un bloque de disco asociado a un fichero ejecutable (*tipo=fichero*). Asimismo este campo también permite al núcleo conocer las acciones que debe realizar sobre el marco de página, donde se va alojar una página asociada a una región de un fichero ejecutable creada a través de la llamada al sistema `exec` cuando dicha página es accedida por primera vez por un proceso. Se distinguen dos acciones:
 - *Llenar de ceros la página física (tipo=DZ)*. Si la página pertenece a la región de datos no inicializados del fichero, la página física tiene que ser llenada de ceros cuando la página es cargada en memoria. A esta acción se le denotará por el acrónimo DZ que se deriva del término inglés “*Demand Zero*”.

- *Cargar el contenido del marco de página* con el contenido de una página de un fichero ejecutable. A esta acción se le denotará por el acrónimo DF que se deriva del término inglés “*Demand Fill*”.

Los procesos que comparten una región por lo tanto acceden a las mismas entradas de las tablas de páginas y descriptores de los bloques de disco. El contenido de una página virtual está o en un bloque particular en un dispositivo de intercambio o en un bloque de un fichero ejecutable en el disco. Si la página se encuentra en el área de intercambio, el descriptor de bloque de disco contiene el número de dispositivo lógico y el número de bloque que contiene los contenidos de la página. Si la página está contenida en un fichero ejecutable en disco, el descriptor de bloque del disco contiene el número de bloque lógico en el fichero que contiene la página. El núcleo puede rápidamente traducir este número en direcciones de disco.

9.2.1.3 La tabla de datos de marcos de página

La *tabla de datos de marcos de páginas* (tabla *dmp*) del núcleo se inicializa al arrancar el sistema y describe cada marco de página o página física de la memoria principal. Esta tabla es indexada por el número de página. Cada entrada de esta tabla posee los siguientes campos:

- *Estado de página*. Indica si la página se encuentra en el área de intercambio o en un fichero ejecutable en el disco. Además indica si la página está siendo leída actualmente del dispositivo de intercambio. También indica si la página puede ser reasignada.
- *Contador de referencias*, que indica el número de procesos que hacen referencia a la página física. Este contador de referencias es igual al número de entradas en las tablas de páginas que hacen referencia a dicha página física. Puede diferir del número de procesos que comparten regiones que contengan esta página, como se verá posteriormente en la sección 9.2.2 cuando se reconsidere el algoritmo `fork`.
- *El dispositivo lógico* (área de intercambio o sistema de ficheros) y *el número de bloque* que contiene una copia de la página.
- *Punteros a otras entradas de la tabla dmp*. El núcleo usa estos punteros para mantener una *lista de marcos de páginas libres*, que contiene a los marcos de

páginas que están disponibles para ser reasignados. El núcleo utiliza esta lista a modo de *caché software de páginas* y la gestiona mediante una política LRU.

Asimismo, el núcleo usa estos punteros para mantener un *conjunto de colas de dispersión*. Cada entrada ocupada de la tabla *dmp*, en función de su número de dispositivo y número de bloque, pertenecerá a una determinada cola de dispersión. De esta forma dados un número de dispositivo y número de bloque el núcleo puede acceder a la cola de dispersión adecuada para determinar rápidamente si la página que busca está cargada en memoria.

Tanto la *lista de marcos de página libres* o *caché de páginas* como las *colas de dispersión* guardan una fuerte analogía con la caché de bloques de disco. De hecho algunas distribuciones tales como SVR4 usan la misma caché tanto para bloques como para páginas.

Cuando se requiere un marco de página libre el núcleo accede a lista de marcos libres, elimina la entrada de la tabla *dmp* situada a la cabeza de la lista (será el marco libre usado menos recientemente), actualiza su número de dispositivo y número de bloque, y la pone en la cola de dispersión correcta. Asimismo cuando se produce un fallo de página el núcleo consulta esta lista por si alguno de sus marcos de página contuviera aún la página que necesita, evitándose así el tener que realizar operaciones de lectura innecesarias en el dispositivo de intercambio o en el disco.

Supuesto que se dispone de una memoria principal de una capacidad C_{Mp} y que el tamaño de página es S_p entonces el número total de marcos de páginas N_{TM} de la memoria principal se calcula de la siguiente forma:

$$N_{TM} = \frac{C_{Mp}}{S_p} \quad (5)$$

Los marcos de la memoria principal se van a identificar por *número de marco* j que puede tomar los siguientes valores $j=0,1,2,\dots,N_{TM}-1$. Luego cada entrada de la tabla *dmp* viene indexada por el número j .

Por otra parte una dirección de memoria principal (dirección física) DIR_F expresada en binario se puede descomponer en dos campos (ver Figura 9.4): el número j de marco de página y el desplazamiento relativo dentro de la página DES_F .

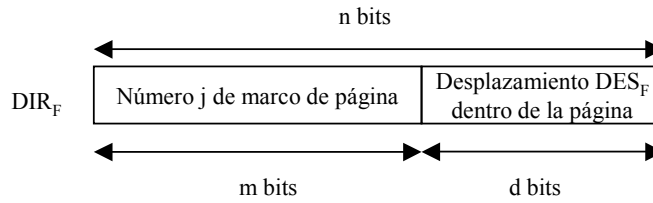


Figura 9.4: Dirección de memoria principal

♦ Ejemplo 9.2:

Supóngase un computador con una memoria principal de capacidad $C_{Mp}=2$ Mbytes y un tamaño de página $S_p=1$ Kbytes. Calcular el contenido en binario y en decimal de cada uno de los campos en que se descompondría la dirección física $DIR_F=1010512$.

El tamaño n de una dirección de memoria es el número de bits que se necesitan para codificar el número total de posiciones direccionables de memoria, puesto que su capacidad es $C_{Mp}=2^{21}$ bytes, supuesto que cada posición de memoria contiene una palabra y que ésta tiene un tamaño de 1 bytes, entonces:

$$n = \log_2 2^{21} = 21 \text{ bits}$$

Por otro lado el número total de marcos de página, se calcularía con la ecuación (5):

$$N_{TM} = \frac{C_{Mp}}{S_p} = \frac{2^{21}}{2^{10}} = 2^{11} = 2048 \text{ marcos de página}$$

El tamaño m del campo “Número j de marco de página” se puede obtener de la siguiente forma

$$m = \log_2 N_{TM} = \log_2 2^{11} = 11 \text{ bits}$$

Y el tamaño d del campo DES_F , se obtiene entonces como:

$$d = n - m = 21 - 11 = 10 \text{ bits}$$

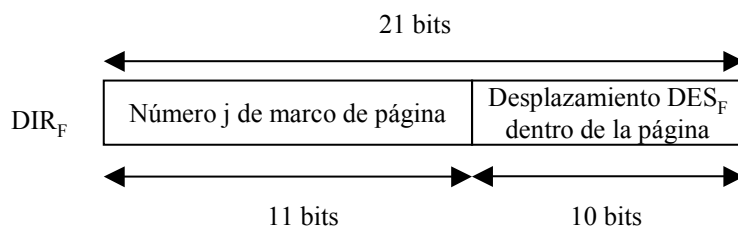
Luego DIR_F tiene la configuración que se muestra en la Figura 9.5

Pasando a binario DIR_F se obtiene:

$$DIR_F = 01111011010 \ 1101010000$$

$$\text{Luego } j = 01111011010 = 986 \text{ y } DES_F = 1101010000 = 848$$

Este resultado está de acuerdo con el obtenido en el Ejemplo 9.1.

Figura 9.5: Configuración de DIR_F

♦

9.2.1.4 Tabla de intercambio

El núcleo dispone de una *tabla de intercambio* que contiene una fila por cada copia de una página situada en un dispositivo de intercambio. En cada fila de esta tabla se mantiene un *contador de referencias* o *contador de entradas* que indica el número de entradas de las tablas de páginas que apuntan a una misma copia de página situada en un dispositivo de intercambio.

♦ Ejemplo 9.3:

Considérese la dirección virtual $DIR_V=68432$, en el Ejemplo 9.1 se calculó que la dirección física a la que hace referencia es $DIR_F=1010512$ y que estaba contenida en la página con $DIR_{F0}=986$ K. Por otra parte en el Ejemplo 9.2 se calculó que el número de marco de página j asociado a DIR_F era $j=986$.

En la Figura 9.6 se han representado dentro de la memoria principal, varios marcos de página que contienen páginas, una tabla de páginas, una tabla *dbd*, la tabla *dmp* y la tabla de intercambio. Además se ha representado varios bloques almacenados en un dispositivo de intercambio (identificado mediante el número 1) en memoria secundaria.

La dirección virtual $DIR_V=68432$ de un proceso está asociada a una entrada de una tabla de páginas cuyo campo DIR_{F0} apunta a la dirección de memoria 986K y al marco de página $j=986$. Asimismo la entrada de la tabla *dbd* asociada a dicha entrada de la tabla de página indica que una copia de esta página existe en el bloque nº 2743 del dispositivo de intercambio 1.

Por otra parte, la entrada $j=986$ de la tabla *dmp* indica que una copia de dicha página existe en el bloque nº 2743 del dispositivo de intercambio 1 y que su contador de referencias contiene el valor 1, lo que indica que solamente un proceso está haciendo referencia a dicha página.

Por otra parte, la entrada de la tabla de intercambio posee un contador de entradas que marca el valor 1, lo que indica que una única entrada de las tablas de páginas apunta a la copia de la página en el dispositivo de intercambio.

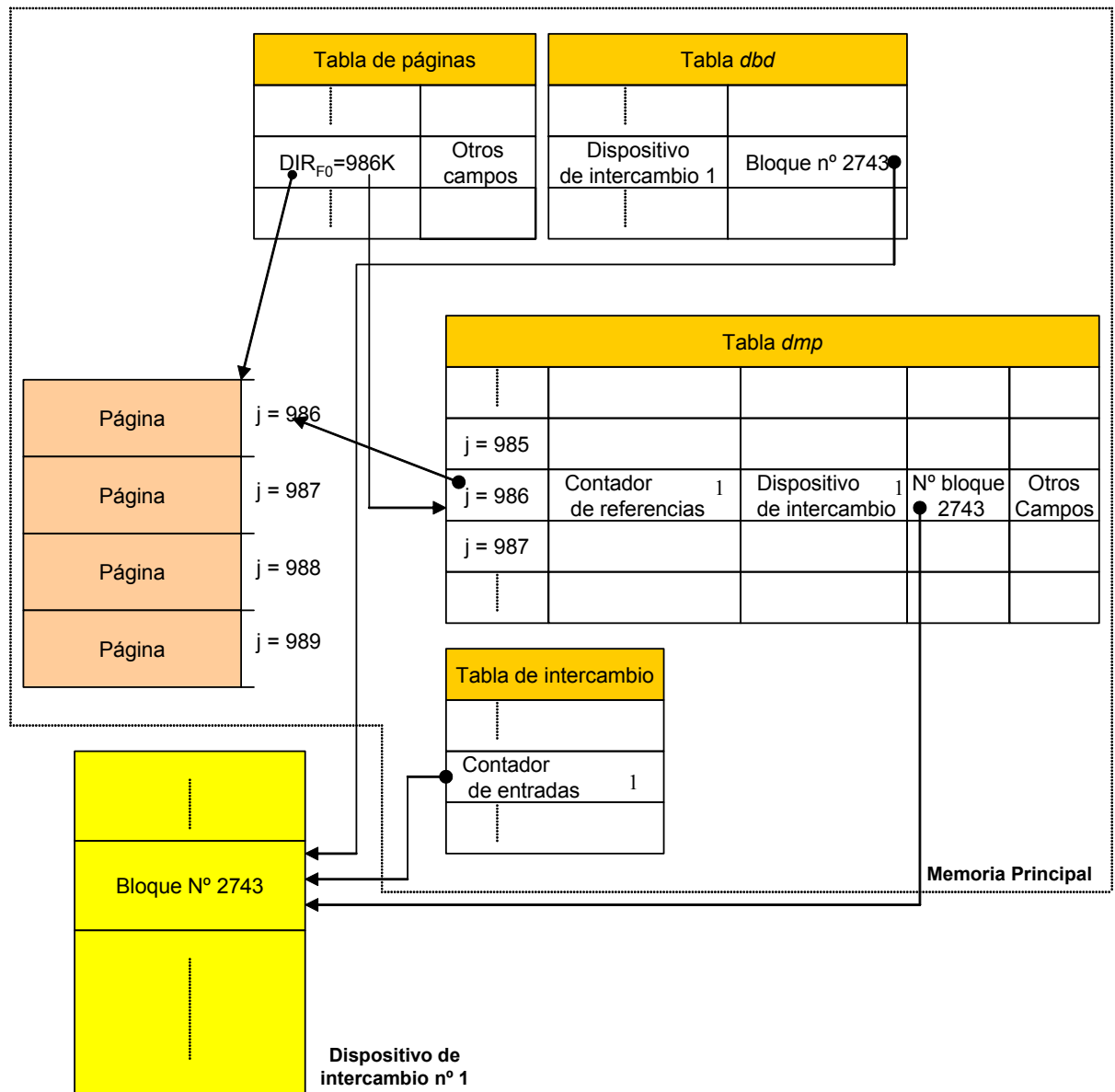


Figura 9.6: Estructuras de datos asociadas a la gestión de la memoria mediante demanda de páginas.

♦

9.2.2 La realización de la llamada al sistema `fork` en un sistema con paginación

Como se explicó en la sección 5.2 al describir la llamada al sistema `fork`. El núcleo duplica cada región del proceso padre y se la asigna al proceso hijo. Tradicionalmente, el núcleo en un sistema con intercambio hace una copia física del espacio de direcciones del padre para asignársela al proceso hijo.

En el sistema de paginación del System V, el núcleo evita realizar la copia del espacio de direcciones del padre, mediante la adecuada manipulación de la tabla de regiones, las tablas de páginas y la tabla *dmp*. El núcleo simplemente incrementa el contador de referencias en la tabla de regiones de las regiones compartidas (como por ejemplo la región de código) por el proceso padre y el proceso hijo. Para regiones privadas tales como la región de datos o la de pila, sin embargo, el núcleo asigna una nueva entrada de la tabla de regiones y una nueva tabla de páginas, y después examina cada entrada de la tabla de páginas del padre. Si una página es válida, incrementa el contador de referencias ubicado en la entrada de la tabla *dmp*, que indica el número de procesos que comparten la página a través de diferentes regiones (en oposición al número de procesos que comparten la página por compartir la región). Además, si la página existe en un dispositivo de intercambio, incrementa el contador de entradas de la tabla de intercambio.

La página ahora puede ser referenciada a través de ambas regiones, que comparten la página hasta que un proceso la escriba. En dicho caso el núcleo entonces copia la página para que cada región tenga una copia privada. Para poder proceder de este modo, el núcleo activa el bit *copiar al escribir* en cada entrada de la tabla de páginas asignada a una región privadas del padre y del hijo durante *fork*. Si un proceso escribe una página, provocará un fallo de protección, y cuando se trate el fallo, el núcleo hará una nueva copia de la página para el proceso que provocó el fallo. La copia física de la página es así aplazada hasta que un proceso realmente la necesita.

♦ Ejemplo 9.4:

Supóngase que un cierto proceso P ha realizado una llamada al sistema *fork* para generar un proceso hijo H. En la Figura 9.7 se representan ciertas estructuras de datos del núcleo una vez finalizada la llamada. Se observa que el proceso P y el proceso H comparten la región de código, y por ello el contador de referencias de la entrada de la tabla de regiones asociada a dicha región contiene el valor 2. Al compartir la región de código, P y H también están compartiendo la tabla de páginas asociada a dicha región. Por este motivo, el contador de referencias de las entradas de la tabla *dmp* para las páginas en la región de texto contendrá el valor 1. Por ejemplo a la dirección física 967K, supuesto páginas de 1K, le corresponde el marco de página $j=967$, cuya entrada asociada en la tabla *dmp* tiene el contador de referencias a 1.

Por otra parte el núcleo ha asignado para H una región de datos, que es una copia de la región de datos del proceso padre P, por eso las tablas de páginas de las dos regiones son idénticas. Por lo tanto, el contador de referencias de las entradas de la tabla *dmp* para las páginas asociadas a dichas región de datos contendrá el valor 2. Por ejemplo a la dirección física 613K, supuesto

páginas de 1K, le corresponde el marco de página $j=613$, cuya entrada asociada en la tabla *dmp* tiene el contador de referencias a 2, ya que es apuntada por dos tablas de páginas, la asociada a la región de datos de P y la asociada a la región de datos de H.

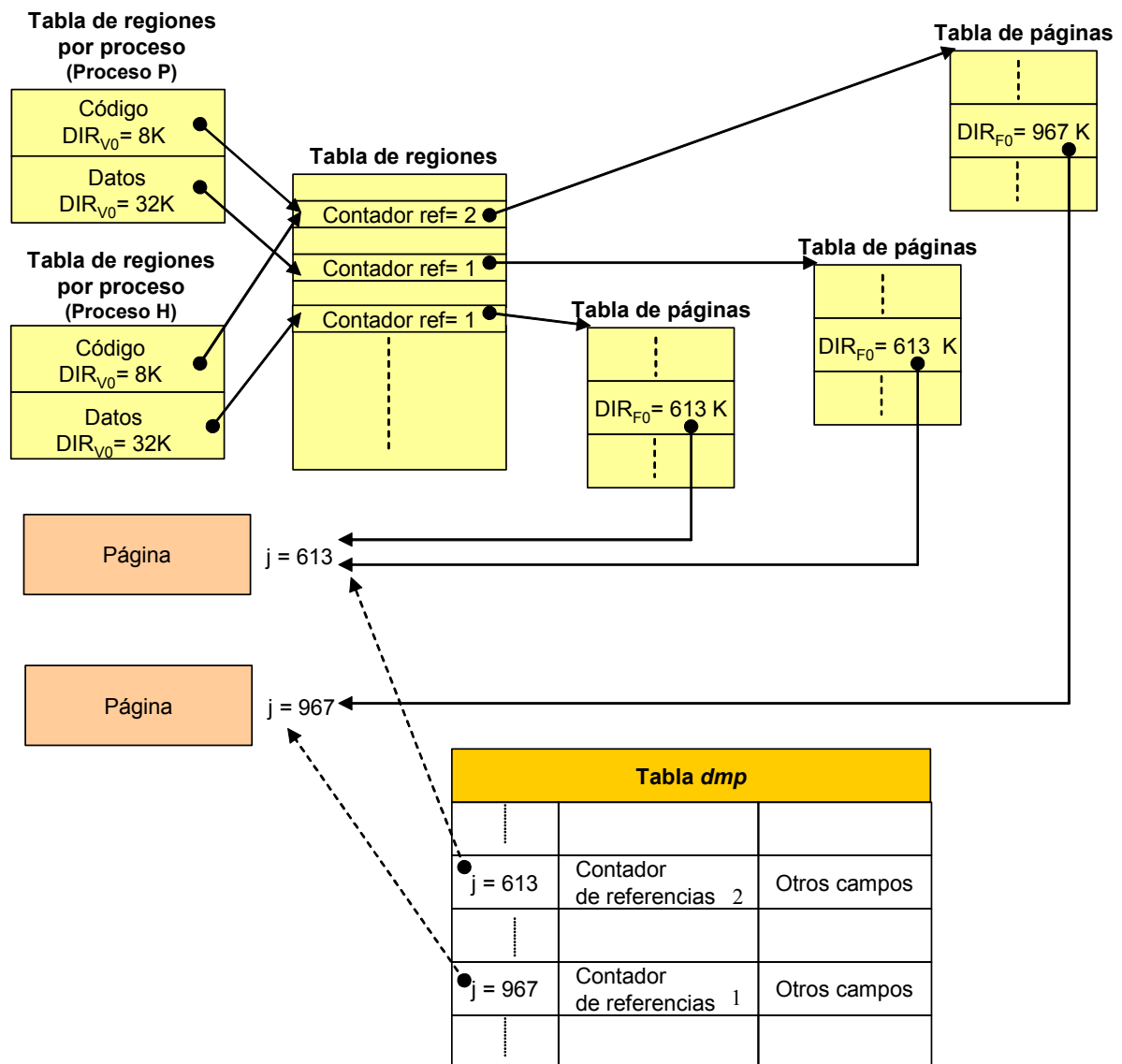


Figura 9.7: Una página en un proceso que ha realizado una llamada al sistema *fork*.

♦

9.2.3 Exec en un sistema de paginación

Cuando un proceso invoca a la llamada al sistema *exec*, el núcleo carga el fichero ejecutable en memoria principal desde el sistema de ficheros, como se describió en la sección 5.7. En un sistema con demanda de página, el fichero ejecutable puede ser demasiado grande para caber en la memoria principal disponible. Por lo tanto, el núcleo

no preasigna memoria al fichero ejecutable, sino que se la va asignando según la va necesitando, es decir, conforme se van produciendo fallos de página

Primero asigna las tablas de páginas y las tablas *dbd* para las regiones del fichero ejecutable, marcando las entradas de las tablas de páginas asociada como DF o DZ. Cuando el núcleo va cargando el fichero ejecutable en memoria, el proceso incurre en un fallo de página en cada lectura de página. El manipulador de fallos comprueba si la página es DF o DZ para realizar en cada caso las acciones oportunas. Si no existe espacio libre en memoria, el proceso del núcleo denominado *ladrón de páginas* periódicamente intercambiará páginas a memoria secundaria para hacer sitio para el fichero.

Existen varios inconvenientes en este esquema de funcionamiento. En primer lugar, un proceso provoca un fallo de página cuando se lee cada una de sus páginas desde el fichero ejecutable. En segundo lugar, el ladrón de páginas puede intercambiar páginas del propio fichero ejecutable fuera de memoria principal antes de que la llamada al sistema `exec` esté completada, lo que resulta en dos operaciones de intercambio extra si el proceso necesita dicha página de nuevo.

Para hacer a la llamada al sistema `exec` más eficiente, el núcleo puede solicitar las páginas directamente desde el fichero ejecutable. Para poder implementar este esquema el núcleo obtiene todos los números de bloque de disco del fichero ejecutable cuando ejecuta la llamada `exec` y adjunta la lista al nodo-*i* del fichero. Cuando configura las tablas de páginas para el fichero ejecutable, el núcleo marca el descriptor del bloque de disco con el número de bloque lógico (empezando por el bloque 0 del fichero) que contiene la página; posteriormente el manipulador de fallos de página utilizará esta información para cargar la página desde el fichero.

♦ **Ejemplo 9.5:**

Supóngase que el núcleo tiene que cargar en memoria una página perteneciente a un fichero ejecutable. El núcleo accede la región a la que esta asociada la tabla de página que contiene dicha página, y sigue el puntero (almacenado en dicha región) al nodo-*i* asociado al fichero ejecutable. Por otra parte en la entrada apropiada de la tabla *dbd* asociada a dicha página, encuentra que el descriptor de bloque en disco es, por ejemplo, 84. Entonces accede al nodo-*i* y en la lista de números de bloque de disco del fichero ejecutable adjuntada al nodo-*i* durante la llamada a `exec` busca la posición 84, que de acuerdo a la Figura 9.8 está asociada al número de bloque de disco 279. Por lo tanto el bloque en disco número 279 contiene la página que se desea cargar en memoria.

0	14
...	...
83	756
84	279
85	26
	...

Figura 9.8: Ejemplo de lista de números de bloques del fichero ejecutable almacenada en el nodo-*i* durante la ejecución de la llamada al sistema `exec`

♦

9.2.4 Transferencia de páginas de memoria principal al área de intercambio

El *ladrón de páginas* es un proceso del núcleo que se encarga de transferir al dispositivo de intercambio las páginas que ya no forman parte del conjunto de trabajo de un proceso. El núcleo crea al ladrón de páginas durante la inicialización del sistema y lo invoca cuando disminuye el número de páginas físicas libres.

Cuando una página se encuentra en memoria principal su campo de *edad* (en la entrada de la tabla de páginas asociada a la página) se incrementa si no es referenciada. Para observar si una página ha sido referenciada el núcleo examina el campo *referenciada* de la entrada de la tabla de páginas asociada a la página. El sistema trabaja sobre un valor *umbral* para dicho campo *edad*, de tal forma que pueden darse dos posibles casos:

- *edad* < *umbral*, la página no es elegible para transferencia ya que hace poco tiempo que se encuentra en memoria principal.
- *edad* > *umbral*, la página será candidata para ser transferida al dispositivo de intercambio.

El núcleo tiene un valor máximo y un valor mínimo para el espacio libre que se debe mantener en memoria principal. Estos valores pueden ser ajustados por el administrado del sistema. Cuando el espacio libre en la memoria principal se encuentra por debajo del valor mínimo establecido el núcleo despierta al *ladrón de páginas* para que transfiera páginas al dispositivo de intercambio. El *ladrón de páginas* se ejecutará hasta conseguir

el valor máximo de espacio libre. De esta manera se consigue reducir el efecto de *thrashing*, es decir, tener que estar transfiriendo páginas que se encuentran en memoria principal hacia un dispositivo de intercambio con el objetivo de conseguir espacio y poder almacenar nuevas páginas necesarias para la ejecución de un determinado proceso.

Cuando el *ladrón de páginas* pretende realizar una transferencia de una página al dispositivo de intercambio debe considerar si ya existe una copia de dicha página en el dispositivo, se pueden presentar tres casos:

- 1) *No existe una copia de la página en el dispositivo de intercambio.* Entonces el núcleo “planifica” la página para ser transferida, es decir, coloca la página en una lista de páginas que deben ser transferidas. Cuando esta lista alcanza un cierto tamaño (que depende de las capacidades del manejador del disco) el núcleo copia todas las páginas de esta lista en el dispositivo de intercambio.
- 2) *Existe una copia de la página en el dispositivo de intercambio y no se ha modificado el contenido de la página de memoria principal* (el campo *modificada* de la entrada de tabla de páginas asociada a dicha página está sin activar). Entonces el núcleo desactiva el campo *válida*, decrementa el contador de referencias en la entrada de la tabla *dmp* y coloca dicha entrada en la lista de marcos de página libres.
- 3) *Existe una copia de la página en el dispositivo de intercambio y se ha modificado el contenido de la página almacenada en memoria principal.* Entonces el núcleo “planifica” la página para ser transferida, y libera el espacio que ocupaba la copia de la página en el dispositivo de intercambio. Cuando se vuelva almacenar la página en el dispositivo de intercambio, su copia se almacenará en otra posición distinta.

En conclusión el *ladrón de páginas* únicamente copia una página en el dispositivo de intercambio si se dan los casos 1 o 3.

Para ilustrar la diferencia entre los casos 2 y 3, supóngase que una página está en un dispositivo de intercambio y es intercambiada a la memoria principal después de que un proceso haya provocado un fallo de página. Supóngase que el núcleo no elimina la copia de la página ubicada en el dispositivo de intercambio automáticamente. Puede suceder que en un determinado momento, el *ladrón de páginas* tenga que intercambiar de nuevo la página de memoria principal al dispositivo de intercambio. Si ningún proceso ha escrito dicha página desde que se puso en memoria principal, la copia en memoria es

idéntica a la existente en el dispositivo de intercambio y por lo tanto no hay ninguna necesidad de modificar la copia existente en el dispositivo de intercambio. Por el contrario, si un proceso ha escrito la página desde que se puso en memoria principal, la copia en memoria difiere de la existente en el dispositivo de intercambio y por lo tanto el núcleo debe escribirla en el dispositivo de intercambio, aunque lo hace en una posición distinta a la que ocupaba la primera copia.

El *ladrón de páginas* va llenando una lista con las páginas que deben ser transferidas, posiblemente de diferentes regiones, y las transfiere al dispositivo de intercambio cuando la lista está llena. Cuando el núcleo escribe una página en el dispositivo de intercambio, desactiva el campo *valida* de su entrada de la tabla de páginas, y decrementa el *contador de referencias* de su entrada de la tabla *dmp*. Si el contador alcanza el valor 0, coloca la entrada de la tabla *dmp* al final de la lista de marcos de página libres. Asimismo si el contador no alcanza el valor 0, significa que varios procesos están compartiendo la página como resultado de una llamada al sistema `fork` realizada con anterioridad, pero aún así el núcleo transferirá la página. Finalmente, el núcleo asigna espacio en el dispositivo de intercambio, salva la dirección del dispositivo de intercambio donde se ha almacenado la página en la entrada de la tabla *dbd asociada a dicha página*, e incrementa el contador de entradas de la tabla de intercambio.

Puesto que el contenido de una página física es válido hasta que ésta es reasignada, el núcleo cuando se produce un fallo de página consulta la lista de marcos de página libres por si alguno de sus marcos de página contuviera aún la página que necesita para evitar así tenerla que leerla del dispositivo de intercambio. No obstante, la página será, de todos modos, intercambiada si ya se ha colocado en la lista de página que deben ser transferidas.

♦ Ejemplo 9.6:

Supóngase que el *ladrón de páginas* debe transferir a un dispositivo de intercambio 30, 40, 50 y 20 páginas de los procesos A, B, C y D, respectivamente, y que en una operación de escritura puede transferir 64 páginas al dispositivo de intercambio. En la Figura 9.9 se muestra la secuencia de operaciones de intercambio de páginas que sucedería si el *ladrón de páginas* examina las páginas de los procesos en el orden A, B, C y D. Se distinguen tres pasos:

- 1) El ladrón de páginas asigna espacio para 64 páginas y transfiere al dispositivo de intercambio 30 páginas del proceso A y 34 páginas del proceso B. Luego ha transferido todas las páginas del proceso A, le restan por transferir 6 páginas del proceso B, 50 del proceso C y 20 del proceso D.

- 2) El ladrón de páginas asigna espacio para otras 64 páginas y transfiere al dispositivo de intercambio 6 páginas del proceso B, las 50 páginas del proceso C y 8 páginas del proceso D. Luego ha transferido todas las páginas del proceso B y del proceso C y le restan por transferir 12 páginas del proceso D.
- 3) El ladrón de páginas guarda las 12 páginas que restan por intercambiar del proceso D en la lista de páginas de intercambio y no las intercambiará hasta que la lista este llena.

Por otra parte, las dos áreas del dispositivo de intercambio utilizadas en los pasos 1) y 2) no tienen porque ser necesariamente contiguas.

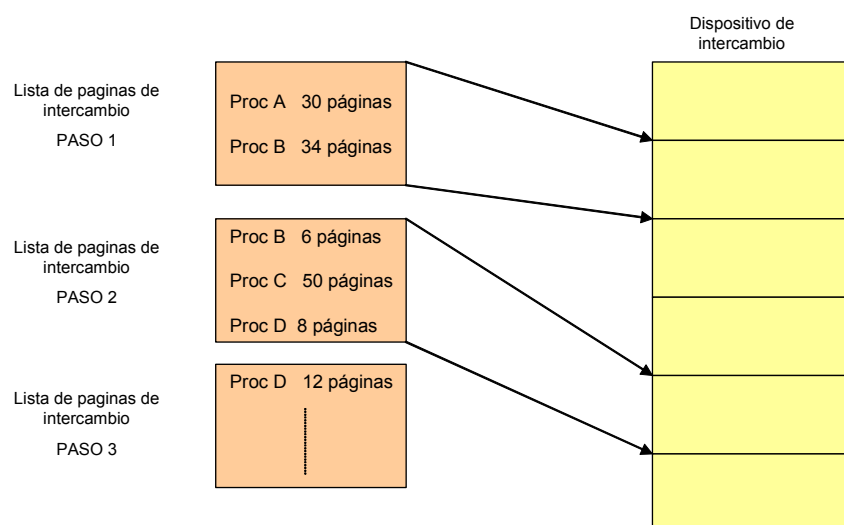


Figura 9.9: Asignación del espacio de intercambio en un esquema de gestión de memoria por demanda de página.

♦

9.2.5 Tratamiento de los fallos de página

El sistema puede incurrir en dos tipos de fallos de página: *fallos de validez* y *fallos de protección*.

- Un *fallo de validez* se produce cuando un proceso intenta acceder a una página cuyo bit *válida* (en su entrada asociada en una tabla de páginas) no está activado. El bit *válida* no está activado para aquellas páginas que no pertenecen al espacio de direcciones virtuales del proceso, ni para aquellas páginas que siendo parte del espacio de direcciones virtuales del proceso no tienen asignado actualmente un marco de página.

- Un *fallo de protección* se produce cuando un proceso intenta acceder a una página válida cuyos bits de protección no permiten acceder a la página (por ejemplo si un proceso intenta escribir en su región de código). Asimismo un proceso puede incurrir en un fallo de protección cuando intenta escribir durante la llamada al sistema `fork` una página cuyo bit *copiar al escribir* esté activado. El núcleo debe determinar la causa del fallo de protección.

Cuando se produce un fallo de página, la MMU genera una excepción que es tratada por un manipulador del núcleo. Cada tipo de fallo de página tiene un cierto manipulador asociado. Estos manipuladores son una excepción a la regla general de que los manipuladores de interrupciones no pueden dormir, ya que un manipulador de fallos cuando se precisa leer una página del disco tiene que dormir mientras se realiza la operación de E/S. Estos manipuladores siempre duermen en el contexto del proceso que provocó el fallo de página.

9.2.5.1 El manipulador de fallos de validez

Para tratar un fallo de validez el núcleo invoca al *manipulador de fallos de validez*, que necesita como argumento de entrada la dirección virtual que al ser accedida ha provocado el fallo de validez. Esta dirección es suministrada al núcleo por la MMU. El manipulador en primer lugar busca la región, la entrada de la tabla de páginas, y la entrada de la tabla dbd asociadas a dicha dirección. En segundo lugar bloquea la región. A continuación comprueba si la dirección que ha provocado el fallo se encuentra fuera del espacio de direcciones virtuales del proceso. En caso afirmativo, el intento de referencia a memoria no es válido y el núcleo envía una señal de violación de segmento (SIGSEGV) al proceso que lo provocó, que al ser tratada provocará la finalización del proceso. Si la referencia a memoria es legal, el núcleo asigna un marco de memoria para la página y lo carga con la página correspondiente que debe ser leída desde el área de intercambio o desde el fichero ejecutable ubicado en el disco.

La página que provocó el fallo se encontrará en uno de los siguientes cinco estados:

- 1) Fuera de memoria principal alojada en un dispositivo de intercambio.
- 2) En la lista de marcos de páginas libres de memoria principal.
- 3) Fuera de memoria principal en un fichero ejecutable en el disco.
- 4) Marcada como DZ.
- 5) Marcada como DF.

Se van a considerar cada uno de estos casos en detalle.

Si una página se encuentra fuera de memoria principal alojada en un dispositivo de intercambio (caso 1), eso significa que dicha página residió en el pasado en memoria principal pero el ladrón de páginas tuvo que intercambiarla fuera de ella. A partir de la entrada correspondiente de la tabla *dbd*, el núcleo encuentra el dispositivo de intercambio y el número de bloque de disco donde la página se encuentra almacenada. Asimismo verifica que la página no se encuentra en la lista de marcos de página libres por si pudiera ahorrarse la operación de lectura en disco. El núcleo actualiza la entrada de la tabla de páginas para que apunte al marco de página donde se va cargar la página, sitúa la entrada de la tabla *dmp* en la cola de dispersión correspondiente, y lee la página desde el dispositivo de intercambio (si fuera necesario). El proceso que provocó el fallo duerme hasta que la operación de E/S se completa, entonces el núcleo despierta a los procesos que estaban esperando a que dicha página fuese cargada en memoria.

♦ Ejemplo 9.7:

Supóngase (ver Figura 9.10) que un proceso provoca un fallo de validez cuando intenta acceder a la dirección virtual 66K. El manipulador de fallos examina la entrada asociada de la tabla *dbd* y encuentra que la página está contenida en el bloque 847 del dispositivo de intercambio (supuesto que solamente hay un dispositivo de intercambio). Por tanto, la dirección virtual es legal, es decir, se encuentra dentro del espacio de direcciones virtuales del proceso. A continuación, el manipulador de fallos de validez busca en la lista de marcos de página libres pero no encuentra una entrada que contenga el bloque 847. Por lo tanto no hay una copia de la página cargada en la memoria principal, y el manipulador de fallos debe leerla desde el dispositivo de intercambio.

Supóngase que el núcleo asigna el marco de página 1776 (ver Figura 9.11), entonces copia en dicho marco la página desde el dispositivo de intercambio, y actualiza la entrada de la tabla de páginas para que apunte a la página física 1776. Finalmente, actualiza la entrada de la tabla *dbd* para indicar que existe todavía una copia de la página en el dispositivo de intercambio, y la entrada de la tabla *dmp* asociada al marco 1776 para indicar que el bloque 847 del dispositivo de intercambio contiene una copia de la página.

DIR _{v0}	Tabla de páginas		Tabla <i>dbd</i>	
	Marco	Válida	Tipo	Bloque de disco
0K				
1K	1648	0	Fichero	3
2K				
3K	Ninguna	0	DF	5
4K				
⋮	⋮	⋮	⋮	⋮
64K	1917	0	Disco	1206
65K	Ninguna	0	DZ	
66K	1036	0	Disco	847
67K				

Tabla <i>dmp</i>		
Marco	Contador de referencias	Bloque de disco
⋮		
1036	0	387
⋮		
1648	1	1618
⋮		
1861	0	1206
⋮		

Figura 9.10: Detalle en un determinado instante de tiempo de parte del contenido de algunas de las estructuras asociadas a la gestión de memoria mediante demanda de páginas.

DIR _{v0}	Tabla de páginas		Tabla <i>dbd</i>	
	Marco	Válida	Tipo	Bloque de disco
	⋮			
66K	1776	1	Disco	847

Tabla <i>dmp</i>		
Marco	Contador de referencias	Bloque de disco
⋮		
1776	1	847

Figura 9.11: Detalle después de tratar el fallo de validez de parte del contenido de algunas de las estructuras asociadas a la gestión de memoria mediante demanda de páginas.



El núcleo no siempre tiene que hacer una operación de E/S cuando incurre en un fallo de validez, si la entrada de la tabla *dbd* indica que la página está intercambiada (caso 2). Es posible que el núcleo no haya reasignado el marco de página después de transferir la página fuera de la memoria principal, o que otro proceso haya provocado que la misma página se haya cargado en otro marco de página. En ambos casos, el manipulador de fallos encuentra la página en la lista de marcos de página libres. Entonces configura la entrada de la tabla de páginas para que apunte a la página física que se acaba de encontrar, incrementa el contador de referencias de la entrada de la tabla *dmp* asociada al marco de página donde se encuentra la página, y elimina el marco de página de la lista de marcos de páginas libres, si fuese necesario.

◆ Ejemplo 9.8:

Supóngase que un proceso provoca un fallo de validez cuando intenta acceder a la dirección virtual 64K (ver Figura 9.10). Supóngase además que buscando la página en la lista de marcos de página libres, el núcleo encuentra que el marco de página 1861 está asociado con el bloque de disco 1206, que coincide con el bloque contenido en de la tabla *dbd* asociada a dicha dirección virtual. Entonces configura la entrada de la tabla de páginas asociada a la dirección virtual 64K para que apunte a la página física 1861, activa el bit *válida* y finaliza el manipulador. Por lo tanto el número de bloque de disco permite asociar una entrada de una tabla *dbd* (y por tanto una entrada de una tabla de páginas) con una entrada de la tabla *dmp*. lo que justifica el que ambas tablas lo almacenen.

De forma similar, el manipulador de fallos de validez (*mfv2*) no tiene que cargar la página en memoria si otro proceso con anterioridad ha provocado un fallo en la misma página y aún no se ha completado su lectura. El manipulador *mfv2* se encontrará a la región asociada a dicha dirección virtual bloqueada por otra instancia del manipulador de fallos de validez (*mfv1*), por lo que *mfv2* duerme hasta que *mfv1* se completa. Cuando despierta *mfv2* se encuentra con que la página ahora si es válida y finaliza.

Si la página se encuentra fuera de memoria principal en un fichero ejecutable en el disco (caso 3), el núcleo leerá la página del fichero ejecutable. El manipulador de fallos accede a la entrada de la tabla *dbd* asociada a la página, y allí obtiene el número de bloque lógico del fichero que contiene la página. Asimismo accede en la tabla de regiones a la región asociada a la dirección virtual que ha provocado el fallo y sigue el puntero al nodo-*i* del fichero ejecutable. Usa el número de bloque lógico como un desplazamiento dentro de la lista de números de bloques de disco adjuntada al nodo-*i* durante la llamada al sistema `exec`. Conocido el número de bloque de disco, lee allí la página y la copia en un marco de memoria principal.

♦ **Ejemplo 9.9:**

Supóngase que un proceso provoca un fallo de validez cuando intenta acceder a la dirección virtual 1K (ver Figura 9.10). En la tabla *dbd* asociada a dicha dirección se observa que el tipo de la página es *fichero*. Esto indica que la página está en un fichero ejecutable, en concreto en el bloque lógico nº 3. El manipulador usa el número de bloque lógico como un desplazamiento dentro de la lista de números de bloques de disco adjuntada al nodo-*i* durante la llamada al sistema `exec`. Conocido el número de bloque de disco, lee allí la página, la copia en un marco de memoria principal y actualiza el contenido de la entrada de la tabla de páginas.

♦

Si un proceso incurre en un fallo de página para una página marcada como DF o DZ (casos 4 y 5), el núcleo asigna un marco de página libre en memoria y actualiza la entrada adecuada de la tabla de páginas. Si la página es DZ entonces llena el marco de página con ceros, si es DF llena el marco de página con una página del fichero ejecutable. Finalmente, desactiva el indicador DZ o DF. La página es ahora válida.

♦ **Ejemplo 9.10:**

Supóngase que un proceso provoca un fallo de validez cuando intenta acceder a la dirección virtual 3K (ver Figura 9.10). En la entrada de la tabla de página asociada a dicha dirección se observa que la página no tenía una dirección física asignada. Esto es debido a que ningún proceso había accedido a ella desde que se había realizado la llamada al sistema `exec`. Por otra

parte, en la tabla *dbd* asociada a dicha dirección se observa que el tipo de la página es DF, y que el bloque lógico del fichero es 5. El manipulador usa el número de bloque lógico como un desplazamiento dentro de la lista de números de bloques de disco adjuntada al nodo-i durante la llamada al sistema *exec*. Conocido el número de bloque de disco, lee allí la página, la copia en un marco de memoria principal y actualiza el contenido de la entrada de la tabla de páginas.

Por otra parte, supóngase que un proceso provoca un fallo de validez cuando intenta acceder a la dirección virtual 65K (ver Figura 9.10). En la entrada de la tabla de página asociada a dicha dirección se observa que la página no tenía una dirección física asignada puesto que ningún proceso había accedido a ella desde que se había realizado la llamada al sistema *exec*. Por otra parte, en la tabla *dbd* asociada a dicha dirección se observa que el tipo de la página es DZ (por eso no tiene un número de bloque lógico). El núcleo asigna un marco de página libre en memoria y lo llena con ceros. A continuación actualiza la entrada adecuada de la tabla de páginas, la página es ahora válida y no tiene copia ni en un área de intercambio ni en un sistema de ficheros.



Una vez realizada las acciones descritas en función del estado en que se encontrará la página, el manipulador de fallos de página activa el bit *válida* de la página, desactiva el bit *modificada* y pone a 0 el campo *edad*. Además recalcula la prioridad del proceso, puesto que el proceso puede haber dormido en el manipulador de fallos en una prioridad a nivel de núcleo, dándole una injusta ventaja de planificación cuando retorna al modo usuario. Finalmente, desbloquea la región bloqueada al comienzo del manipulador.

9.2.5.2 Manipulador de fallos de protección

Para tratar un fallo de protección el núcleo invoca al manipulador de fallos de protección, que necesita como argumento de entrada la dirección virtual que al ser accedida ha provocado el fallo de protección. Esta dirección es suministrada al núcleo por la MMU. El núcleo al ejecutar el manipulador en primer lugar busca la región, la entrada de la tabla de páginas, la entrada de la tabla *dbd* y la entrada de la tabla *dmp* (*edmp1*) asociadas a dicha dirección. En segundo lugar bloquea la región para que el ladrón de páginas no pueda seleccionar la página para ser intercambiada mientras el manipulador está trabajando sobre ella. A continuación comprueba si el fallo de protección se ha producido porque se ha intentado acceder a una página válida cuyos bits de protección no permiten acceder a la página. En dicho caso envía una señal SIGBUS³ al proceso que provocó el fallo, desbloquea la región y finaliza. Cuando la señal sea tratada provocará la finalización del proceso.

³ No todas las distribuciones envían esta misma señal.

Por otra parte, si el manipulador determina que el fallo fue causado porque el bit *copiar al escribir* estaba activado, y si la página física es compartida con otros procesos, el núcleo asigna un nuevo marco de página y copia en él la página que originó el fallo; los otros procesos mantienen sus referencias a la página física original. Después de copiar la página en el nuevo marco y actualizar la entrada de la tabla de páginas con el nuevo número de página física, el núcleo decreuenta el contador de referencias de *edmp1*.

♦ **Ejemplo 9.11:**

Supóngase que tres procesos comparten la página física 828 (ver Figura 9.12). El proceso B escribe la página e incurre en un fallo de protección, puesto que el bit *copiar al escribir* estaba activado. El manipulador de fallos de protección entonces asigna el marco de página 786, copia la página contenida en el marco 838 en el marco 786, decreuenta el contador de referencias del marco 828, y actualiza la entrada de la tabla de páginas accedida por el proceso B para que apunte a la página física 786 (ver Figura 9.13).

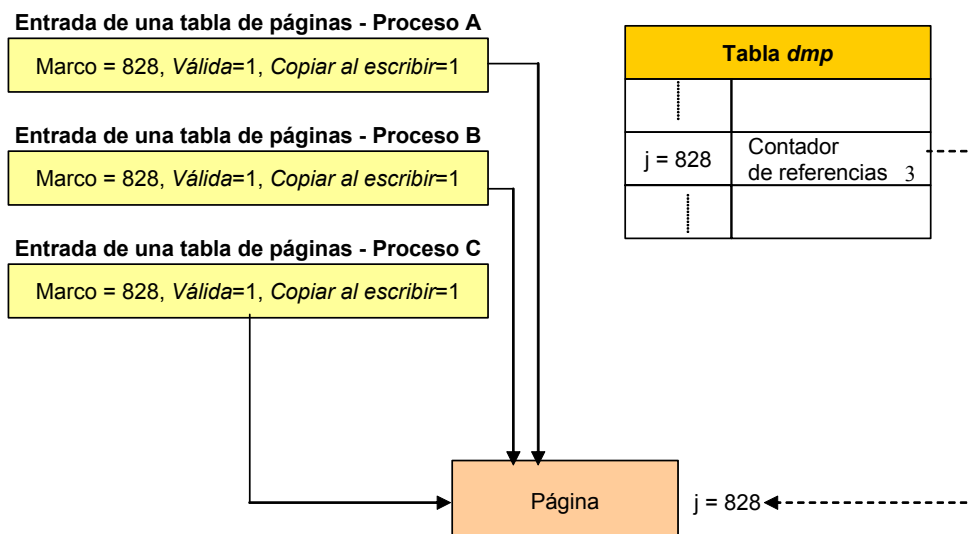


Figura 9.12: Detalle de parte del contenido de algunas de las estructuras asociadas a la gestión de memoria mediante demanda de páginas en un determinado instante de tiempo

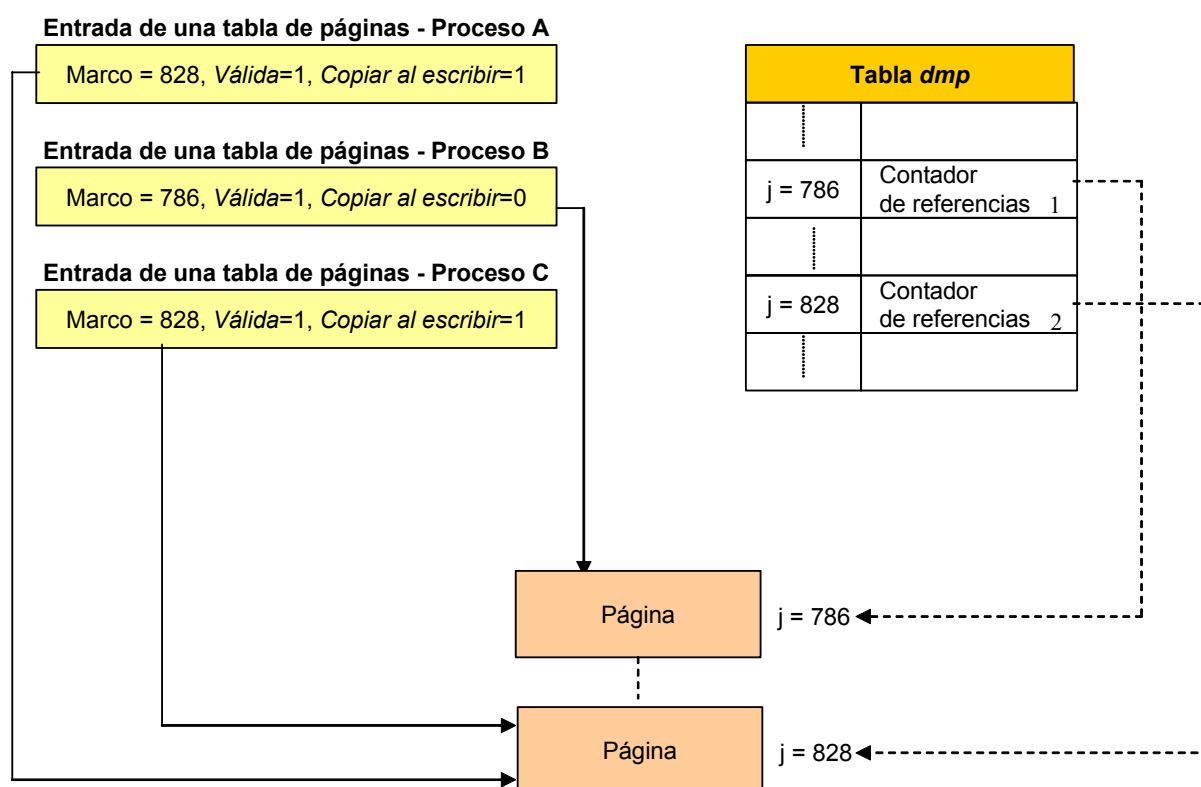


Figura 9.13: Detalle de parte del contenido de algunas de las estructuras asociadas a la gestión de memoria mediante demanda de páginas después de gestionar el fallo de protección

Si el bit *copiar al escribir* está activado pero ningún otro proceso comparte la página, el núcleo permite al proceso reutilizar la página física. Desactiva el bit *copiar al escribir* y desasocia la página de su copia en el disco, si existe alguna, puesto que otros procesos pueden compartir la copia en el disco. A continuación, en la tabla de intercambio decrementa el contador de entradas para la página, si el contador llega a 0, libera el espacio de intercambio.

Si una entrada de una tabla de páginas es no válida y su bit *copiar al escribir* está activado para causar un fallo de protección, se va a suponer que el sistema trata primero el fallo de validez cuando un proceso accede a dicha página. No obstante, el manipulador de fallos de protección debe comprobar que una página es todavía válida, porque podría dormir cuando se bloquea una región, y el ladrón de páginas podría mientras tanto intercambiar la página fuera de memoria. Si la página es inválida, el manipulador de fallos retorna inmediatamente, y el proceso incurrirá en un fallo de validez. El núcleo tratará el fallo de validez, pero el proceso incurrirá después en un fallo de protección. Lo más probable, es que trate este fallo de protección sin ninguna interferencia más, puesto que

la página tardará un tiempo en envejecer lo suficiente para poder ser intercambiada fuera de memoria.

Las últimas acciones que realiza el manipulador de fallos de protección antes de finalizar su ejecución son: activar los bits de *protección* y el bit *modificada*, desactivar el bit *copiar al escribir*, recalcular la prioridad del proceso, y desbloquear la región que había bloqueado al comienzo de su ejecución.

9.2.6 Explicación desde el punto de vista de la gestión de memoria del cambio de modo de un proceso

Supóngase que la memoria está organizada en páginas físicas de 1Kbytes, a las que se accede a través de *tablas de páginas*. Asimismo supóngase que la máquina dispone de un conjunto de *registros triples* de administración de memoria. El primer registro del registro triple contiene la dirección de memoria de una tabla de páginas en memoria física, el segundo registro contiene la primera dirección virtual que traduce la tabla de páginas, y el tercer registro contiene información de control tal como el número de páginas en la tabla de páginas y permisos de acceso a la página (solo lectura, lectura-escritura). Este modelo se corresponde al modelo de región. Cuando el núcleo prepara a un proceso para ser ejecutado, carga el conjunto de registros triples con los datos correspondientes almacenados en las entradas de la tabla de regiones por proceso.

Aunque el núcleo se ejecuta en el contexto de un proceso, la traducción de la memoria virtual asociada con el núcleo es independiente de todos los procesos. El código y los datos del núcleo residen en el sistema permanentemente, y todos los procesos lo comparten. Cuando la máquina es arrancada, carga el código del núcleo dentro de memoria y configura las tablas y registros necesarios para poder traducir sus direcciones virtuales en direcciones físicas. Las tablas de páginas del núcleo son análogas a las tablas de páginas asociadas a los procesos de usuarios.

En muchas máquinas, el espacio de direcciones virtuales de un proceso es dividido en varias clases, incluyendo sistema y usuario, y cada clase tiene su propia tabla de páginas. Cuando se ejecuta en modo núcleo, el sistema permite el acceso a las direcciones del núcleo. El acceso a estas direcciones está prohibido cuando se ejecuta en modo usuario. Así, cuando se cambia de modo usuario a modo núcleo como resultado de una interrupción o una llamada al sistema, el sistema operativo colabora con el hardware para permitir referencias a las direcciones del núcleo, y cuando se vuelve de modo núcleo a modo usuario se prohíben tales referencias. Otras máquinas cambian la

traducción de direcciones virtuales cargando registros especiales cuando se ejecutan en modo núcleo.

♦ **Ejemplo 9.12:**

Supóngase que las direcciones virtuales del núcleo están comprendidas en el rango 0 a 4M-1, y las direcciones virtuales de usuario empiezan a partir de 4M. En la Figura 9.14 se observan dos conjuntos de registros triples de administración de memoria, uno para las direcciones del núcleo y otro para las direcciones de usuario. Cada registro triple apunta a la tabla de páginas que contiene los números de páginas físicas correspondientes a las direcciones de páginas virtuales.

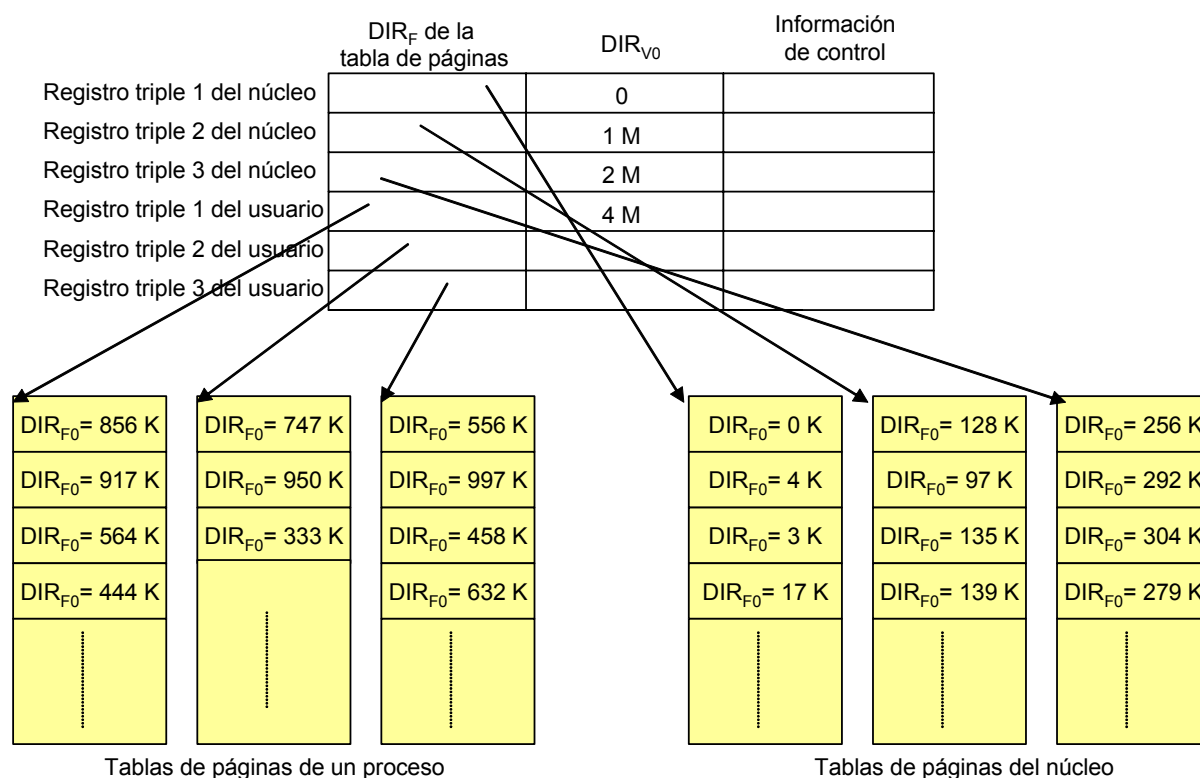


Figura 9.14: Ubicación en memoria de las tablas de páginas del núcleo y de un cierto proceso.

El sistema permite las referencias de direcciones a través de los registros triples del núcleo solamente cuando se encuentra en modo núcleo. Por lo tanto, el cambio de modo núcleo a modo usuario o viceversa, requiere únicamente que el sistema permita o prohíba las referencias de direcciones a través de los registros triples del núcleo.

9.2.7 Localización en memoria del área U de un proceso

Como ya se describió en la sección 4.4.3 cada proceso tiene su propia área U. Sin embargo el núcleo accede a ella como si solamente existiera un única área U en todo el sistema, la del proceso actual. El núcleo cambia su mapa de traducción de direcciones virtuales de acuerdo con el proceso que se está ejecutando para acceder al área U correcta. Cuando se compila el sistema operativo, el cargador asigna a la variable u una dirección virtual fija asociada siempre al área U del proceso actual. Luego el núcleo únicamente puede acceder simultáneamente al área U de un cierto proceso, el proceso actual.

El valor de la dirección virtual del área U es conocida para otras partes del núcleo, en concreto, el módulo que realiza el cambio de contexto. Puesto que el núcleo conoce el lugar exacto, dentro de sus tablas de administración de memoria, donde se realiza la traducción de direcciones virtuales del área U, cuando el núcleo planifica un proceso para ejecutar, encuentra la correspondiente área U en memoria física y la hace accesible por medio de su dirección virtual. Para ello cambia dinámicamente la traducción de direcciones del área U a las direcciones físicas asociadas al área U del nuevo proceso actual.

♦ Ejemplo 9.13:

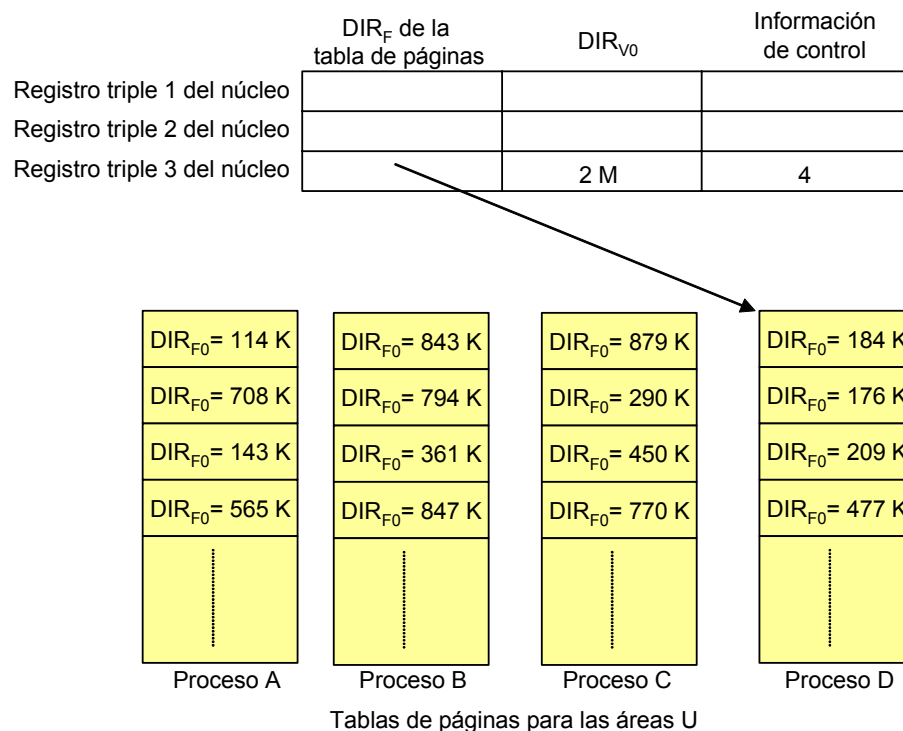


Figura 9.15: Localización en memoria de las tablas de páginas del área U de diferentes procesos.

Supóngase que el área U tiene un tamaño de 4 Kbytes y reside en la dirección virtual del núcleo 2M. En la Figura 9.15 se observa que los dos primeros registros triples se refieren al código y datos del núcleo (las direcciones y punteros no son mostrados). Estos registros nunca cambian, puesto que todos los procesos comparten el código y los datos del núcleo.

Por otra parte el tercer registro triple del núcleo se refiere al área U del proceso D. Si el núcleo desea acceder al área U del proceso A, entonces copia en este registro triple la información apropiada de la tabla de páginas asociada al área U del proceso A. Por lo tanto, en cualquier instante, el tercer registro triple del núcleo se refiere al área U del proceso actualmente planificado para ejecución.

◆

BIBLIOGRAFIA

- Bach, J.M. *The design of the Unix Operating System*. Prentice-Hall. 1986.
- Goltfried, B. *Programación en C*. McGraw Hill. 1997
- Márquez, F.C. *UNIX: Programación Avanzada*. R.A.M.A. 1996.
- Vahalia, U. *UNIX Internal: The New Frontier*. Prentice Hall. 1996.

