El sistema operativo UNIX es un sistema multiusuario y multiproceso escrito en lenguaje C que desde su nacimiento a principio de la década de los setenta ha ido alcanzado bastante éxito y popularidad tanto en el ámbito universitario como en el empresarial. Asimismo UNIX, es la base del sistema operativo de libre distribución *Linux* que es un clon de UNIX nacido a principios de la década de los noventa que cada vez está comenzando a interesar a un mayor número de usuarios y de empresas.

En este libro se ofrece principalmente una visión interna de UNIX, es decir, se estudian las estructuras de datos, las llamadas al sistema y los algoritmos que conforman este sistema operativo, así como la interrelación existente entre todos estos elementos para garantizar el correcto funcionamiento del mismo.

UNED

FUNDAMENTOS DEL SISTEMA OPERATIVO UNIX

Jose Manuel Díaz Martínez
Rocío Muñoz Mansilla

UNED

Jose Manuel Díaz Martínez

Rocio Muñoz Mansilla

Fundamentos del sistema operativo UNIX

U.N.E.D

Fundamentos del sistema operativo UNIX

ISBN: 978-84-691-0884-0

Enero 2008

Editores: Jose Manuel Díaz - Rocio Muñoz Mansilla

Dpto. Informática y Automática
E.T.S.I de Informática.
Universidad de Educación a Distancia (UNED).

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

CONTENIDOS

PRÓLOGO

El sistema operativo UNIX es un sistema multiusuario y multiproceso escrito en lenguaje C que desde su nacimiento a principio de la década de los setenta ha ido alcanzado bastante éxito y popularidad tanto en el ámbito universitario como en el empresarial. Asimismo UNIX, es la base del sistema operativo de libre distribución *Linux* que es un clon de UNIX nacido a principios de la década de los noventa que cada vez está comenzando a interesar a un mayor número de usuarios y de empresas.

Hay dos formas de estudiar un sistema operativo, la primera forma conocida como *visión externa* estudia un sistema operativo desde su administración por parte de un usuario. La segunda forma conocida como *visión interna* estudia las estructuras de datos, las llamadas al sistema y los algoritmos que conforman un sistema operativo, así como la interrelación existente entre todos estos elementos para garantizar el correcto funcionamiento del mismo.

Existen en la actualidad numerosos libros en español dedicados a la visión externa de UNIX. Por el contrario, el número de libros en esta lengua dedicados a la visión interna de UNIX es muy reducido. El presente libro pretende contribuir a ir equilibrando la balanza. Así, el objetivo fundamental de este libro es dar una visión interna básica del sistema operativo UNIX, de tal forma que el lector sea capaz de comprender de forma global el funcionamiento de este sistema operativo.

Este libro consta de diez capítulos cuyos contenidos han sido organizados y seleccionados con el objetivo de asegurar, en la medida de lo posible, un aprendizaje secuencial. Así, el Capítulo 1 está dedicado al lenguaje de programación C, que es el lenguaje en que está escrito UNIX. En el Capítulo 2, con el objetivo de dar al lector una idea global de UNIX, se realizan unas consideraciones generales sobre este sistema operativo. En el Capítulo 3, se dan unas nociones básicas sobre la administración de UNIX. El Capítulo 4 está dedicado al estudio de las estructuras de datos que mantiene el núcleo de UNIX para poder soportar la ejecución de distintos programas o procesos. En el Capítulo 5 se describen los principales algoritmos que el núcleo utiliza para controlar la ejecución de los procesos. El Capitulo 6 analiza la planificación de procesos en UNIX. En el Capítulo 7 se

estudian los mecanismos de comunicación y sincronización de los procesos en UNIX. El Capítulo 8 se dedica a la descripción de los sistemas de ficheros en UNIX. En el Capítulo 9 se estudia la gestión de memoria en UNIX. Finalmente, el Capítulo 10 se dedica a describir el subsistema de entrada/salida de UNIX.

El libro contiene tres apéndices. En el Apéndice A se incluyen las principales consideraciones que se deben tener en cuenta antes de instalar el sistema operativo Linux y se enumeran las principales distribuciones de Linux existentes actualmente. La mayoría de los ejemplos que se incluyen en este libro han sido realizados sobre una distribución de un sistema operativo Linux. Por ello, con el objetivo de practicar con los contenidos que vaya aprendiendo en cada capítulo, se recomienda al lector que se instale en su PC cualquier distribución de Linux, asegurándose de disponer de un compilador de lenguaje C, por ejemplo, `gcc`.

En el Apéndice B se recopilan las funciones de biblioteca de uso más frecuente. Mientras que en el Apéndice C se resumen las llamadas al sistema que han ido apareciendo en las diez capítulos del libro.

Finalmente, sólo desearle al lector que la lectura de este libro le sea de interés y provecho. Los autores estaremos encantado de recibir en `soii@iti.uned.es` las sugerencias y las erratas detectadas en el texto con el objetivo de ir mejorando las futuras ediciones de este libro.

CAPÍTULO 1

El lenguaje de programación C

1.1 INTRODUCCIÓN

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 también están escritas 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 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 puede compilar y ejecutar en muchas computadoras diferentes sin tener que realizar en la mayoría de los casos ninguna modificación en los programas.

- Los compiladores de C son frecuentemente compactos y generan programas objeto que son pequeños y muy eficientes.

1.2 CICLO DE CREACIÓN DE UN PROGRAMA

Un *compilador* es un programa que toma como entrada un texto escrito en un lenguaje de programación de alto nivel, denominado *fuente* 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 convertirse después 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 nombra, por convenio, añadiéndole la extensión `.c`. Se va suponer en lo que resta de sección que dicho fichero se llama `prog.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 la orden:

```
$ gcc prog.c
```

Si no existen errores de compilación está orden se ejecutará correctamente generando como resultado el fichero ejecutable `a.out`.

Si se quiere personalizar el nombre del fichero de salida se debe escribir la orden

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

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

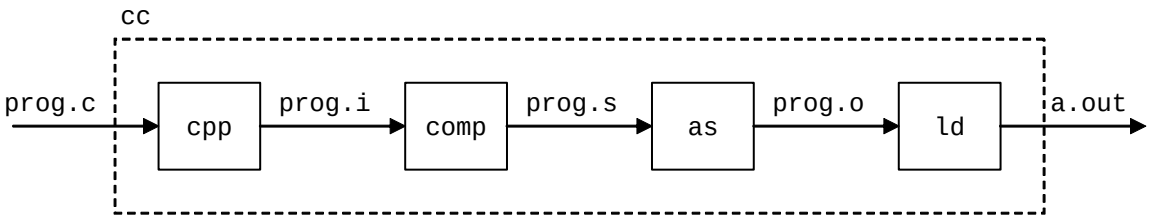


Figura 1.1: Fases del proceso de compilación

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

- 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* `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 procede a escribir las N funciones cuyos prototipos se han definido anteriormente.

♦ **Ejemplo 1.1:**

Considérese el siguiente programa escrito en lenguaje C

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

La primera línea del programa es un comentario sobre el programa. En C los comentarios 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` que contiene funciones estándar de entrada/salida. La tercera línea es la declaración de función principal `main`. El indicador de tipo `void` al comienzo de la línea indica que `main` no genera ningún parámetro de salida. Mientras que el indicador de tipo `void` encerrado entre paréntesis al final de la línea indica que `main` no tiene parámetros de entrada. Algunos compiladores pueden dar mensajes de aviso o inclusos errores de compilación por usar el indicador de tipo `void` como tipo del parámetro de salida (y/o de entrada), por lo que alternativamente a la sentencia

```
void main(void)
```

se puede utilizar dependiendo del compilador la sentencia

```
main(void)
```

o la sentencia

```
main()
```

Las restantes líneas de este programa se corresponden al cuerpo de la función principal, que en este caso sólo consta de una sentencia simple del tipo `printf` que imprime en el dispositivo de salida estándar (típicamente el monitor) mensajes de texto. Obsérvese que de forma general todas las sentencias simples terminan en punto y coma.

Supóngase que el código fuente de este programa se encuentra en un fichero de texto llamado `programa1_1.c` y que al fichero ejecutable de este programa se le desea llamar `prog1`. La orden que hay que invocar desde la línea de órdenes (`$`) para generar este fichero ejecutable es:

```
$ gcc –o prog1 programa1_1.c
```

Para ejecutar `prog1` se teclea la orden

```
$ prog1
```

la ejecución de este programa mostrará por pantalla el mensaje

```
" HOLA A TODOS "
```

Debe tenerse en cuenta que si el directorio de trabajo actual donde está el fichero `prog1` no está añadido en la lista de directorios de la variable de entorno PATH (ver sección 3.6.9) entonces el programa no se ejecutará ya que el intérprete de comandos no lo encontrará y mostrará un mensaje en la pantalla avisando de esta circunstancia.

♦

1.4 CONCEPTOS BÁSICOS 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 ANSI[1].

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.

`Auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Tabla 1.1: Palabras reservadas en C según el estándar ANSI

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:

- *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 (base 10), octal (base 8) y hexadecimal (base16).

  - *Constante entera decimal*. Puede ser cualquier combinación de dígitos tomados del conjunto `0` a `9`. Si la constante tiene dos o más dígitos, el primero de ellos debe ser distinto de `0`.

  - *Constante entera octal*. Puede estar formada por cualquier combinación de dígitos tomados del conjunto `0` a `7`. El primer dígito debe ser obligatoriamente `0`, con el fin de identificar la constante como un número octal.

[1] ANSI es el acrónimo derivado del término inglés "*American National Standards Institute*" (Instituto Nacional Americano de Estándares).

  - *Constante entera hexadecimal*. Debe comenzar por `0x` o `0X`. Puede aparecer después cualquier combinación de dígitos tomados del conjunto `0` a `9` y de `a` a `f` (tanto minúsculas como mayúsculas).

- *Constantes en coma flotante*. Es un número decimal que contiene un punto decimal o un exponente (o ambos).

- *Constantes de carácter*. Es un sólo carácter encerrado con comillas simples. Cada carácter tiene un valor entero equivalente de acuerdo con el código ASCII[2]

- *Constantes de cadenas de caracteres*. Consta de cualquier número de caracteres consecutivos encerrados entre comillas dobles.

♦ **Ejemplo 1.2:**

A continuación se muestran algunos ejemplos de diferentes tipos de constantes:

```
0      10     743    32767                  (Constantes enteras decimales)
01     025    0547   07777                  (Constantes enteras octales)
0x1    0X1a   0X7F   0xabcd                 (Constantes enteras hexadecimales)
0.5    2.3    827.602  1.666E12             (Constantes en coma flotante)
'Z'    '?'    ' '    'a'                    (Constantes de carácter)
"trabajo"   "Sistema operativo UNIX"        (Constantes de cadena de caracteres)
```

♦

Las constantes enteras y en coma flotante representan números, por lo que se las denomina en general como *constantes de tipo numérico*. Las siguientes reglas se pueden aplicar a todas las constantes numéricas:

1) No se pueden incluir comas ni espacios en blanco en la constante.

2) Una constante puede ir precedida de un signo menos (`-`). Realmente, el signo menos es un operador que cambia el signo de una constante positiva.

[2] ASCII es el acrónimo derivado del término inglés "*American Standard Code for Information Interchange*" (Código Estadounidense Estándar para el Intercambio de Información).

3) El valor de una constante no puede superar un límite máximo y un límite mínimo especificados. Para cada tipo de constante, estos límites dependen del compilador de C utilizado.

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 '`;`'

♦

Se denomina *secuencia de escape* a una combinación de caracteres que comienza siempre con una barra inclina hacia atrás `\` y es seguida por uno o más caracteres especiales. Una secuencia de escape siempre representa un solo carácter, aun 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 utilizadas más frecuentemente

El compilador inserta automáticamente un carácter nulo (`\0`) al final de toda constante de cadena de caracteres, aunque este carácter no aparece cuando se visualiza

la cadena. El saber que el último carácter de una cadena de caracteres es siempre el carácter nulo es de gran ayuda si la cadena es examinada carácter a carácter, como sucede en muchas aplicaciones.

Es importante darse cuenta que una constante de carácter (por ejemplo `'J'`) y su correspondiente cadena de caracteres (por ejemplo `"J"`) no son equivalentes ya que la cadena no consta de un único carácter sino de dos (`'J'` y `'\0'`).

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_variable = constante;
```

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

♦ **Ejemplo 1.4:**

A continuación se muestran algunos ejemplos de declaración e inicialización de variables:

```
/*Declaración */
      char letra;
      int entero;
      float real;
/*Inicialización*/
      letra='a';
      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*. Son aquellos suministrados por el lenguaje.
- *Tipos derivados*. Son aquellos 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.
- `int`. Define un número entero de 16 o 32 bits (dependiendo del procesador).
- `long`. Define un número entero de 32 o 64 bits (dependiendo del procesador).
- `short`. Define un número entero de tamaño menor o igual que `int`. En general se cumple que: tamaño(`short`) ≤ tamaño(`int`) ≤ tamaño(`long`).

Estos tipos pueden ir precedidos del modificador `unsigned` que indica que el tipo sólo representa números positivos o el cero. Es usual utilizar la abreviatura `u` para designar a `unsigned`, en dicho caso el modificador precede al tipo de datos pero sin un espacio entre medias.

♦ **Ejemplo 1.5:**

Con las sentencias

```
int numero=3;
char letra='c';
unsigned short contador=0;
ulong barra=123456789;
```

se están declarando: la variable `numero` de tipo `int` inicializada a 3, la variable `letra` de tipo `char` inicializada a 'c', la variable `contador` de tipo `unsigned short` inicializada a 0 y la variable `barra` de tipo `ulong`, abreviatura de `unsigned long`, inicializada a `123456789`.

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:**

Con las sentencias

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

se están declarando: la variable `ganancia` de tipo `float` inicializada a `125.23` 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 son necesarios para acceder a un elemento particular del array.

La declaración formal de una array multidimensional de tamaño N es la siguiente:

$$\texttt{tipo\_array nombre\_array[rango}_1\texttt{][rango}_2\texttt{]...[rango}_N\texttt{];}$$

donde $\texttt{rango}_1$, $\texttt{rango}_2$,... y $\texttt{rango}_N$ son expresiones enteras positivas que indican el número de elementos del array asociados con cada índice.

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]`;

♦

Los array pueden ser inicializados en el momento de su declaración. Para ello los valores iniciales deben aparecer en el orden en que serán asignados a los elementos del array, encerrados entre llaves y separados por comas. Todos los elementos del array que no sean inicializados de forma explícita serán inicializados por defecto al valor cero.

El tamaño de un array unidimensional no necesita ser especificado explícitamente cuando se incluyen los valores iniciales de los elementos. Con un array numérico el tamaño del array será fijado automáticamente igual al número de valores iniciales incluidos dentro de la definición del array.

En el caso de las cadenas de caracteres la especificación del tamaño del array normalmente se omite. El tamaño adecuado de la cadena es asignado automáticamente como el número de caracteres que aparecen en la cadena más el carácter nulo `'\0'` que se añade automáticamente al final de cada cadena de caracteres.

♦ **Ejemplo 1.8:**

La siguiente definición

```
int datos[5]={1, 6, -5, 4, 0};
```

o equivalentemente la definición

```
int datos[]={1, 6, -5, 4, 0};
```

declara el array `datos` de cinco elementos de tipo entero que son inicializados con los siguientes valores: `datos[0]=1`, `datos[1]=6`, `datos[2]=-5`, `datos[3]=4` y `datos[4]=0`.

La definición

```
float X[3]={12.26, 25.31};
```

declara el array `X` de tres elementos de tipo coma flotante que son inicializados con los siguientes valores: `X[0]=12.26`, `X[1]=25.31` y `X[2]=0`. Nótese que la definición

```
float X[]={12.26, 25.31};
```

no es equivalente a la anterior ya que ahora al no especificarse explícitamente el tamaño del array este se calcula a partir del número de valores iniciales especificados. Luego ahora la dimensión del array `X` es de 2 elementos y no de 3 como en el caso anterior.

La definición

```
char nombre[]="Rocio";
```

declara una cadena de caracteres de seis elementos: `nombre[0]='R'`, `nombre[1]='o'`, `nombre[2]='c'`, `nombre[3]='i'`, `nombre[4]='o'` y `nombre[5]='\0'`.

La definición anterior se podría haber escrito equivalentemente como

```
char nombre[6]= "Rocio";
```

Nótese como al especificar la dimensión del array se debe incluir un elemento extra para el carácter nulo `'\0'`.

♦

En el caso de la inicialización de los array multidimensionales se debe tener cuidado en el orden en que se asignan los elementos del array. La regla que se utiliza es que el último índice (situado más a la derecha) es el que se incrementa más rápido y el primer índice (situado más a la izquierda) es el que se incrementa más lentamente. De esta forma, los elementos de un array bidimensional se deben asignar por filas comenzando por la primera.

El orden natural en el que los valores iniciales son asignados se puede alterar formando grupos de valores iniciales encerrados entre llaves. Los valores dentro de cada par interno de llaves serán asignados a los elementos del array cuyo último índice varíe más rápidamente. Por ejemplo, en un array bidimensional, los valores almacenados dentro del par interno de llaves serán asignados a los elementos de la primera fila, ya que el segundo índice (columna) se incrementa más rápidamente. Si hay pocos elementos dentro de cada par de llaves, al resto de los elementos de cada fila se les asignará el valor cero. Por otra parte, el número de valores dentro de cada par de llaves no puede exceder del tamaño de fila definido.

♦ **Ejemplo 1.9:**

La siguiente definición

```
int base[2][3]={1, 6, -5, 4, 0, 8};
```

o equivalentemente la definición

```
int base[2][3]={
      {1, 6, -5},
      {4, 0, 8}}
};
```

declara el array bidimensional `base` que puede ser considerado como una tabla de dos filas y tres columnas (tres elementos por fila) que es inicializada con los siguientes valores:

```
base[0][0]=1      base[0][1]=6      base[0][2]=-5
base[1][0]=4      base[1][1]=0      base[1][2]=8
```

Nótese que los valores iniciales se asignan por filas, el índice situado más a la derecha se incrementa más rápidamente.

La definición

```
int base[2][3]={
      {1, 6},
      {4, 0}}
};
```

asigna valores únicamente a los dos primeros elementos de cada fila. Luego los elementos del array `base` tendrán los siguientes valores iniciales:

```
base[0][0]=1      base[0][1]=6      base[0][2]=0
base[1][0]=4      base[1][1]=0      base[1][2]=0
```

La definición

```
int base[2][3]={
      {1, 6, -5, 6},
      {4, 0, 8, -9}}
};
```

produciría un error de compilación, ya que el número de valores dentro de cada par de llaves (cuatro valores en cada par) excede el tamaño definido del array (tres elementos en cada fila).

♦

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 son usados frecuentemente en C ya que tienen una gran cantidad de aplicaciones. Por ejemplo, pueden ser usados para pasar 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 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.

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.

♦ **Ejemplo 1.10:**

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`.

Supónganse ahora las siguientes tres sentencias:

```
int z, *pz;

pz=&z;

*pz=25;
```

La primera sentencia está declarando una 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`.

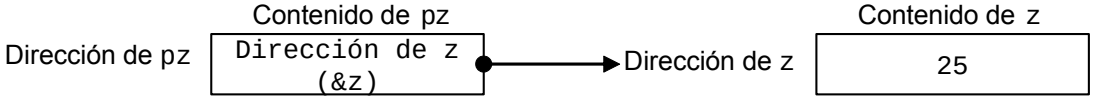


Figura 1.2: Relación entre el puntero pz y la variable z

♦

Por otra parte, es posible realizar ciertas operaciones con una variable puntero, a continuación se resumen las operaciones permitidas:

- A una variable puntero se le puede asignar la dirección de una variable ordinaria (por ejemplo, `pz=&z`).
- A una variable puntero se le puede asignar el valor de otra variable puntero (`pz=py`), siempre que ambos punteros apunten al mismo tipo de datos.
- A una variable puntero se le puede asignar un valor nulo (cero). 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;
```

- A una variable puntero se le puede sumar o restar una cantidad entera (por ejemplo, `px +3`, `pz++`, etc).
- Una variable puntero puede ser restada de otra con tal que ambas apunten a elementos del mismo array.
- Dos variables puntero pueden ser comparadas siempre que ambas apunten a datos del mismo tipo.

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 elemento `0` (primer elemento) del array se puede expresar tanto como `&x[0]` o simplemente como `x`. Además el contenido del elemento `0` del array se puede expresar como `x[0]`o como `*x`. La dirección del elemento `1` (segundo elemento) del array se puede escribir como `&x[1]` o como `(x+1)`. Mientras que el contenido del elemento `1` del array se puede expresar como `x[1]`o como `*(x+1)`. En general, la dirección del elemento `i` del array se puede expresar bien como `&x[i]` o como `(x+i)`. Mientras que el contenido del elemento `i` del array se puede expresar como `x[i]`o como `*(x+i)`.

♦ **Ejemplo 1.11:**

El siguiente programa en C ilustra las dos formas equivalentes de acceso a los elementos de un array unidimensional:

```
#include <stdio.h>
main()
{
      int vector[3]={1, 2, 3};
      printf("\n %d %d %d", vector[0],vector[1],vector[2]);
```

```
        *vector=4;      /*Esta sentencia equivale a vector[0]=4*/
        *(vector+1)=5; /*Esta sentencia equivale a vector[1]=5*/
        *(vector+2)=6; /*Esta sentencia equivale a vector[2]=6*/

        printf("\n %d %d %d", *vector,*(vector+1),*(vector+2));
}
```

♦

Puesto que el nombre de un array es realmente un puntero al primer elemento dentro del array, es posible definir el array como una variable puntero en vez de como un array convencional. Sintácticamente las dos definiciones son equivalentes. Sin embargo, la definición convencional de un array produce la reserva de un bloque fijo de memoria al principio de la ejecución del programa, mientras que esto no ocurre si el array se representa en términos de una variable puntero. En este último caso la reserva de memoria la debe realizar el programador de forma explícita en el código del programa mediante el uso de la función `malloc` (ver sección 1.9).

Un array multidimensional se puede expresar como un array de punteros. En este caso el nuevo array será de una dimensión menor que el array multidimensional. Cada puntero del array indicará el principio de un array de dimensión *N*-1. Así la declaración de un array multidimensional de orden N

$$\texttt{tipo\_array nombre\_array[rango}_1\texttt{][rango}_2\texttt{]...[rango}_N\texttt{];}$$

puede sustituirse equivalentemente por la declaración de un array de punteros de dimensión N-1:

$$\texttt{tipo\_array *nombre\_array[rango}_1\texttt{]...[rango}_{N-1}\texttt{];}$$

Obsérvese que cuando un array multidimensional de dimensión N se expresa mediante un array de punteros de dimensión N -1 no se especifica $\texttt{[rango}_N\texttt{]}$.

El acceso a un elemento individual dentro del array se realiza simplemente usando el operador `*`.

Al igual que en el caso unidimensional, el usar un array de punteros para representar a un array multidimensional requiere la reserva explícita de memoria mediante el uso de la función `malloc`.

♦ **Ejemplo 1.12:**

Considérese que `z` es un array bidimensional de números en coma flotante con 3 filas y 4 columnas. Se puede definir `z` como

```
float z[3][4];
```

o como un array unidimensional de punteros escribiendo

```
float *z[3];
```

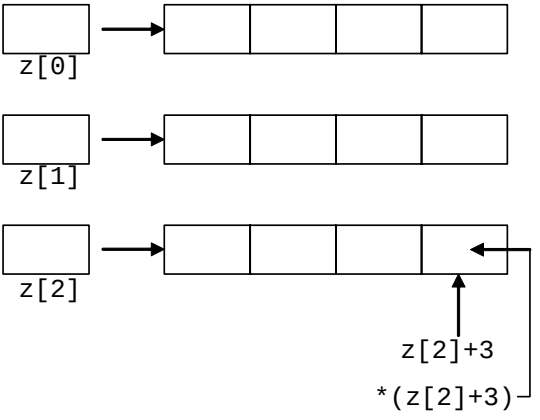


Figura 1.3: Uso de un array de punteros para referirse a un array bidimensional de 3 filas y 4 columnas.

En este segundo caso (ver Figura 1.3), `z[0]` apunta al elemento 0 (primer elemento) de la fila 0 (primera fila), `z[1]` apunta al elemento 0 de la fila 1 (segunda fila) y `z[2]` apunta al elemento 0 de la fila 1 (tercera fila). Obsérvese que el número de elementos dentro de cada fila no está especificado.

Para acceder al elemento de la fila 2 situado en la columna 3 (`z[2][3]`) se puede escribir

```
*(z[2]+3)
```

En esta expresión `z[2]` es un puntero al elemento 0 (primer elemento) de la fila 2, de modo que `(z[2]+3)` apunta al elemento 3 (cuarto elemento) de la fila 2. Luego `*(z[2]+3)` se refiere al elemento en la columna 3 de la fila 2, que es `z[2][3]`.

♦

El uso de arrays de punteros es muy útil para almacenar cadenas de caracteres. En esta situación, cada elemento del array es un puntero de tipo carácter que indica donde comienza la cadena. De esta forma un array de punteros de *n* elementos de tipo carácter puede apuntar a *n* cadenas diferentes. Cada cadena puede ser accedida haciendo uso de su puntero correspondiente. También es posible asignar un conjunto de valores iniciales como parte de la declaración del array de punteros. Estos valores iniciales serán cadenas de caracteres, donde cada una representa a un elemento distinto del array.

♦ **Ejemplo 1.13:**

Supóngase que se desean almacenar durante la ejecución de un programa las siguientes cadenas de caracteres en un array de tipo carácter:

```
Sol
Tierra
Luna
Vía Láctea
```

Estas cadenas se pueden almacenar, usando las sentencias apropiadas, en un array bidimensional de tipo carácter, por ejemplo

```
char astronomía[4][11];
```

Nótese que `astronomía` tiene 4 filas para 4 cadenas. Cada fila debe ser suficientemente grande para almacenar por lo menos 11 caracteres, ya que `Vía Láctea` tiene 10 caracteres más el carácter nulo (`\0`) al final. Otra forma de almacenar este array bidimensional es definir un array de 4 punteros:

```
char *astronomía[4];
```

Nótese que no es necesario incluir el número de columnas dentro de la declaración del array. No obstante, posteriormente en el código del programa se tendrá que reservar una cantidad especifica de memoria para cada cadena de caracteres haciendo uso de la función `malloc` (ver sección 1.9). También se podía haber inicializado el array con las cadenas de caracteres al realizar la declaración del array.

```
char *astronomía[4]={
                        "Sol"
                        "Tierra"
                        "Luna"
                        "Vía Láctea"
                    };
```

Así el elemento 0 (primer elemento) del array de punteros `astronomía[0]` apuntará a `Sol`, el elemento 1 (segundo elemento) `astronomía[1]` apuntará a `Tierra` y así sucesivamente. Como la declaración del array incluye valores iniciales, no es necesario especificar en la declaración de forma explícita el tamaño del array. Por lo tanto, la declaración anterior se puede escribir equivalentemente de la siguiente forma:

```
char *astronomía[]={
                        "Sol"
                        "Tierra"
                        "Luna"
                        "Vía Láctea"
                    };
```

♦

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 '.'. Asimismo 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 1.14:**

Las siguientes sentencias

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

están declarando una estructura del tipo `cliente`. Por otra parte la sentencia:

```
struct cliente uno;
```

está declarando la variable de estructura `uno` del tipo `cliente`. Con la sentencia

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

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

♦

♦ **Ejemplo 1.15:**

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 1.16:**

Considérense las siguientes sentencias:

```
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 1.17:**

Las siguientes sentencias son un ejemplo de como es posible declarar una estructura dentro de otra estructura:

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

♦

1.4.5.4 Uniones

Las uniones se definen de forma parecida a las estructuras, es decir, contienen miembros cuyos tipos de datos pueden ser diferentes. Sin embargo, todos los miembros que componen una unión comparten la misma área de almacenamiento dentro de la memoria, mientras que cada miembro dentro de una estructura tiene asignada su propia área de almacenamiento. Por lo tanto, las uniones se utilizan para ahorrar memoria. Resultan bastante útiles para aplicaciones que requieren múltiples miembros donde únicamente se requiere asignar simultáneamente valor a un único miembro. El tamaño de la memoria reservada a una unión va a ser fijado por el compilador tomando como referencia el tamaño del mayor de los miembros de dicha unión. La declaración de una unión es la siguiente:

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

La asignación de valores a los miembros de una unión se realiza de forma parecida a como se realiza para los miembros de una estructura.

♦ **Ejemplo 1.18:**

Considérese la siguiente declaración de una unión.

```
union multiuso
{
      int numeroZ;
      char campo[6];
} uno;
```

Las sentencias anteriores son la definición de la variable de unión `uno` del tipo `multiuso`. Esta variable puede representar en un momento dado o un número entero (`numeroZ`) o una cadena de 6 caracteres (`campo`). Puesto que la cadena necesita más memoria que el entero, el compilador reserva para esta variable de unión un bloque de memoria suficientemente grande para poder almacenar la cadena de 6 caracteres.

♦

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 1.19:**

Con las sentencias

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

se está definiendo el identificador `entero` como sinónimo de `int` y se está declarando la variable `y` de tipo `entero`.

♦

1.4.6 Tipos de almacenamiento

Las variables se pueden clasificar por su *tipo de almacenamiento* al igual que por su tipo de datos. El tipo de almacenamiento especifica la parte del programa dentro del cual se reconoce a la variable. Hay cuatro especificaciones diferentes de tipo de almacenamiento en C: automática, externa, estática y registro. Están identificadas por las siguientes palabras reservadas: `auto`, `extern`, `static` y `register`.

Si se desea especificar el tipo de almacenamiento de una variable éste debe colocarse antes del tipo de datos de la variable:

```
tipo_almacenamiento tipo_dato nombre_variable;
```

A veces se puede establecer el tipo de almacenamiento asociado a una variable por la posición de su declaración en el programa por lo que no es necesario utilizar la palabra reservada asociada a dicho tipo de almacenamiento. En otras situaciones, la palabra reservada que especifica un tipo particular de almacenamiento se tiene que colocar al comienzo de la declaración de la variable.

1.4.6.1 Variables automáticas

Las *variables automáticas* se declaran siempre dentro de una función y son locales a dicha función. Luego una variable automática no mantiene su valor cuando se transfiere el control fuera de la función en que está definida. Asimismo las variables automáticas definidas en funciones diferentes serán independientes unas de otras, incluso aunque tengan el mismo nombre.

Cualquier variable declarada dentro de una función se interpreta por defecto como una variable automática a menos que se incluya dentro de la declaración un tipo de almacenamiento distinto. Por lo tanto no es obligatorio incluir al principio de la declaración de una variable automática la palabra reservada `auto`.

1.4.6.2 Variables externas (globales)

Las *variables externas o globales*, a diferencia de las variables automáticas, no están confinadas a una determinada función. Su ámbito se extiende desde el punto de definición hasta el resto del programa. Por lo tanto, a una variable externa se le puede asignar un valor dentro de una función y este valor puede usarse (al acceder a la variable externa) dentro de otra función.

No es necesario escribir el especificador de tipo de almacenamiento `extern` en una definición de una variable externa, porque las variables externas se identifican por la localización de su definición en el programa. De hecho algunos compiladores producen errores cuando se usa esta palabra clave en la declaración de una variable global.

1.4.6.3 Variables estáticas

Las *variables estáticas* se definen dentro de funciones individuales y tienen, por tanto, el mismo ámbito que las variables automáticas, es decir, son locales a la función en que están definidas. Sin embargo, a diferencia de las variables automáticas, las variables estáticas retienen sus valores durante todo el tiempo de vida del programa. Por lo tanto, si se sale de la función y posteriormente se vuelve a entrar, las variables estáticas definidas dentro de esa función mantendrán sus valores previos. Esta característica permite a las funciones mantener información permanente a lo largo de toda la ejecución del programa.

Las variables estáticas se definen dentro de una función de la misma forma que las variables automáticas, excepto que la declaración de variables tiene que comenzar con la palabra clave `static`.

En ocasiones dentro de un mismo programa se definen variables automáticas o estáticas que tienen el mismo nombre que variables externas. En tales casos las variables locales tienen precedencia sobre las variables externas, aunque los valores de las variables externas no se verán afectados por la manipulación de las variables locales.

♦ **Ejemplo 1.20:**

Supóngase el siguiente programa escrito en lenguaje C:

```
int r=1, s=2, t=3;
void func1(void);
main()
{
      static int r=4;
      printf("\nA: %d %d %d\n", r, s, t);
      r=r+1;
      func1();
      printf("\nB: %d %d %d\n", r, s, t);
      func1();
      printf("\nC: %d %d %d\n", r, s, t);
}
void func1(void)
{
      static int r=0;
      int s=0;
      if (r==0)
            r=r+s+t+2;
      else
            r=r+10;
      printf("\nD: %d %d %d\n", r, s, t);
}
```

En este programa las variables de tipo entero `r`, `s` y `t` son variables externas. Sin embargo, `r` ha sido redefinida dentro de `main` como variable de tipo entero estática. Las modificaciones que se realicen a la variable `r` dentro de la función `main` serán locales a esta función y no afectarán al valor de la variable global `r`. Luego dentro de la función `main` sólo se reconocen como variables externas a `s` y `t`

De forma análoga, en la función `func1` se define la variable estática `r` y la variable automática `s`, ambas de tipo coma flotante. Luego `r` mantendrá su valor previo si se vuelve a entrar dentro de la función `func1`, mientras que `s` perderá su valor siempre que se transfiera el control del programa fuera de `func1`. Por otra parte dentro de func1 sólo se reconoce como variable externa a `t`.

La ejecución del fichero ejecutable que se genera al compilar este programa mostraría la siguiente traza de ejecución en pantalla:

```
A: 4 2 3
D: 5 0 3
B: 5 2 3
D: 15 0 3
C: 5 2 3
```

♦

1.4.6.4 Variables registro

Los registros son áreas especiales de almacenamiento dentro de la unidad central de procesamiento (CPU) de una computadora cuyo tiempo de acceso es mucho más pequeño que la memoria principal. El tiempo de ejecución de algunos programas se puede reducir considerablemente si ciertos valores pueden almacenarse dentro de los registros en vez de en la memoria principal.

En C, los valores de las *variables registro* se almacenan dentro de los registros de la CPU. A una variable se le puede asignar este tipo de almacenamiento simplemente precediendo la declaración de tipo con la palabra reservada `register`. Pero sólo puede haber unas pocas variables registro dentro de cualquier función. Típicamente dos o tres, aunque depende de la computadora y del compilador.

Los tipos de almacenamiento `register` y `automatic` están muy relacionados, ya que su ámbito definición es el mismo, es decir son locales a la función en la que han sido declaradas.

El declarar varias variables como `register` no garantiza que sean tratadas realmente como variables de tipo `register`. La declaración será válida solamente si el espacio requerido de registro está disponible. En caso contrario, la variable declarada se tratará como si fuese una variable automática.

Finalmente comentar que el operador dirección (`&`) no se puede aplicar a las variables registro.

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. 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`.

Operador	Acción
-	Resta
+	Suma
*	Multiplicación
/	División[3]
%	Resto de la división
--	Decremento
++	Incremento

Tabla 1.3: Operadores aritméticos

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

♦ **Ejemplo 1.21:**

Las siguientes sentencias son un ejemplo del uso de la representación simplificada del operador suma y del operador resta:

```
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,

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

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
&&	AND lógica
||	OR lógica
!	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

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.

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

1.5.5 Conversión de tipos

Los operandos de una expresión que difieren en el tipo pueden sufrir una conversión de tipo antes de que la expresión alcance su valor final. En general, el resultado final se expresará con la mayor precisión posible, de forma consistente con los tipos de datos de los operandos. Asimismo aparte de esta conversión implícita, si se desea, es posible convertir explícitamente el valor resultante de una expresión a un tipo de datos diferente. Para ello la expresión debe ir precedida por el nombre del tipo de datos deseado, encerrado con paréntesis:

```
(tipo de datos) expresión;
```

A este tipo de construcción se le denomina *conversión de tipos* (*cast*).

♦ **Ejemplo 1.22:**

Supóngase que `h` es una variable en coma flotante cuyo valor es `3.5`. La expresión

```
f%3
```

no es válida ya que `h` está en coma flotante en vez de ser un entero. Sin embargo, la expresión

```
((int)h)%3
```

hace que el primer operando se transforme en un entero y por lo tanto es válida, dando como resultado el resto entero `0`. Sin embargo, obsérvese que `h` sigue siendo una variable en coma flotante con un valor de 3.5, aunque el valor de `h` se convirtiese en un entero (`3`) al efectuar la operación del resto.

Supóngase ahora que `a` es una variable entera de valor 20. La expresión

```
((int)(a+h))%3
```

hace que el primer operando `(a+h)` se transforme en un entero y da como resultado el resto entero 2.

♦

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.

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, por defecto el teclado. Su sintaxis es

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

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

♦ **Ejemplo 1.23:**

Considérense el siguiente par de sentencias:

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

En la primera sentencia se declara la variable `c` de tipo carácter. La segunda instrucción hace que se lea del dispositivo de entrada estándar un carácter y 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, por defecto el monitor. Su sintaxis es

```
putchar(variable);
```

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

♦ **Ejemplo 1.24:**

Considérense el siguiente par de sentencias:

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

En la primera instrucción se declara la variable `c` de tipo carácter y se inicializa con el carácter '`a`'. La segunda instrucción hace que se visualice el valor de `c` en el dispositivo de salida estándar.

♦

1.6.3 Introducción de datos: función `scanf`

Mediante la función de biblioteca `scanf` se pueden 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 las direcciones de memoria donde se encuentran los datos.

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
`x`	Entero hexadecimal
`[...]`	Cadena de caracteres que puede incluir caracteres de espaciado

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

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 estar precedido por un carácter ampersand (`&`), salvo en el caso de los arrays.

♦ **Ejemplo 1.25:**

Considérese el siguiente programa en C:

```
#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 lo 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 1.26:**

Considérese el siguiente programa en C:

```
#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 terminada con el carácter de nueva línea `'\n'` desde el dispositivo de entrada estándar. La declaración de esta función es:

```
char *gets(char *s);
```

El significado de esta declaración es el siguiente: `gets` es una función que acepta un argumento que es un puntero a carácter `char *s` y devuelve un puntero a carácter `char *gets(. . .)`.

La sintaxis típica de esta función es

```
gets(s);
```

donde `s` es el nombre de la cadena de caracteres (o el nombre del puntero a carácter) donde se va almacenar la información leída. Esta función reemplaza el carácter de nueva línea `'\n'` que aparece al final de la cadena y lo sustituye por el carácter nulo `'\0'`.

Nótese que la función `gets` desconoce la longitud de `s` ya que solamente se le está pasando como argumento la dirección de inicio del array de caracteres. En consecuencia si se introduce una línea con más caracteres de los que se pueden almacenar en `s` se estaría sobrescribiendo en la zona de memoria que hay a continuación del espacio reservado a `s` y que estará asociado a otras variables, con lo que se estaría modificando el valor de las mismas. No será posible darse cuenta de este error hasta que no se usen estas variables. Este es el motivo por el cual si se compila un programa que contiene la función `gets` el compilador muestra un aviso (warning) que indica que el uso de esta función puede ser peligroso. Una posible solución es usar la función `fgets` definida en la biblioteca `stdio.h` (ver Complemento 2.A).

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

```
puts(s);
```

donde `s` es la cadena de caracteres que se desea visualizar. Esta función añade al final de la cadena `s` el carácter de nueva línea `'\n'`.

♦ **Ejemplo 1.27:**

El siguiente programa en C 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 1.28:**

Las siguientes sentencias son un ejemplo de bloque de proposiciones:

```
{  /* 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 1.29**

Las siguientes sentencias ilustran el uso de la instrucción `if`:

```
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 1.30:**

Las siguientes sentencias ilustran el uso de la instrucción `if - else`:

```
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(expresión n-1)
        proposición n-1;
```

```
else
      proposición_n;
```

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 parámetro inicialmente asignado por `inicialización`. `Proposición` se ejecutará mientras `expresión` sea `Verdadera`. La proposición puede ser simple o compuesta.

♦ **Ejemplo 1.31:**

Las siguientes sentencias ilustran el uso de la instrucción `for`:

```
int dígito;
for (dígito=0; dígito<=3; dígito++)
      printf("%d\n",dígito);
```

En este bucle el índice de control `dígito` se inicializa al valor `0`. La condición que debe ser satisfecha para que se continúe la ejecución del bucle es que `dígito` sea menor o igual a `3`. El índice se incrementa en una unidad (`dígito++`) al finalizar una ejecución del bucle. En cada ejecución del bucle se muestra en el dispositivo en la pantalla el valor de `dígito`. Luego la ejecución de estas líneas generaría la siguiente traza en la pantalla:

```
0
1
2
3
```

Al finalizar la ejecución de este bucle el valor de `dígito` es `4`.

♦

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 1.32:**

Las siguientes sentencias ilustran el uso de la instrucción `while`:

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

♦

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 1.33:**

Las siguientes sentencias ilustran el uso de la instrucción `do - while`:

```
int dígito=0;
do{
      printf("%d\n",dígito++);
}
while (dígito<=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 1.34:**

Las siguientes sentencias ilustran el uso de las instrucciones `break` y `continue`:

```
int x=100;
while (x<=100){
      x=x-1
      if (x<0){
            printf("Valor negativo de x\n");
            break;
      }
      if (x==50){
            printf("Reducción de x a la mitad\n");
            continue;
      }
      printf("Decrementar\n");
}
printf("Final\n");
```

Se tiene un bucle de tipo `while` cuya condición para ejecutarse es que el valor de `x` sea menor o igual a `100`. El valor inicial de esta variable es `100`, de acuerdo a su declaración. En cada pasada del bucle en primer lugar se decrementa en una unidad el valor de `x`. En segundo lugar se comprueba si `x` es menor que cero, en caso afirmativo se imprime en la pantalla el mensaje

```
Valor negativo de x
```

y se ejecuta la instrucción `break` que hace que finalice el bucle `while`. Con lo que la próxima instrucción que se ejecuta es `printf("Final\n");`

En tercer lugar, si `x` no es menor que cero, se comprueba si es igual a `50`, en caso afirmativo se imprime en la pantalla el mensaje

```
Reducción de x a la mitad
```

y se ejecuta la instrucción `continue` que hace que se salte directamente a la siguiente pasada del bucle `while`, por lo que no se ejecuta en la pasada actual del bucle la instrucción `printf("Decrementar\n");`

En cuarto y último lugar, si las dos comprobaciones anteriores han dado resultado negativo se muestra por pantalla el mensaje

```
Decrementar
```

Y se procede a realizar la siguiente pasada a través del bucle.

♦

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`. La primera instrucción de cada grupo debe ir precedida por una o varias etiquetas `case`. Estas etiquetas permiten identificar el grupo de instrucciones asociado a un determinado valor de la expresión. Uno de los grupos de instrucciones se puede etiquetar con `default`. Este grupo se seleccionará si el valor de la expresión no coincide con ninguno de los valores especificados en las etiquetas `case`. Si no se especifica un grupo de instrucciones con la etiqueta `default` y el valor de la expresión no coincide con ninguno de los valores especificados en las etiquetas `case` entonces la instrucción `switch` no hace nada.

De forma general la sintaxis de una instrucción `switch` es:

```
switch(expresión)
{
      case valor_expresión_1:
            instrucción 1;
            instrucción 2;
            ...
            break;

      case valor_expresión_2:
            instrucción 1;
            instrucción 2;
            ...
            break;
      ...
      default:
```

```
            instrucción 1;
            instrucción 2;
            ...
            break;
}
```

También es posible asignar varias etiquetas `case` a un mismo grupo de instrucciones:

```
      case valor_expresión_1:
      case valor_expresión_2:
      case valor_expresión_3:
      ...
      case valor_expresión_m:
            instrucción 1;
            instrucción 2;
            ...
            break;
```

♦ **Ejemplo 1.35:**

Las siguientes sentencias ilustran el uso de la instrucción `switch`:

```
      char eleccion;
      switch (eleccion=getchar()){
      case 'f':
      case 'F':
            printf("Aviso 1\n");
            break;
      case 'g':
      case 'G':
            printf("Aviso 2\n");
            break;
      case 't':
      case 'T':
            printf("Aviso 3\n");
            break;
      default:
            printf("\nEntrada errónea");
}
```

Si el carácter introducido por el teclado es `'f'` o `'F'` se mostrará por pantalla el mensaje `Aviso 1`. Si el carácter introducido es `'g'` o `'G'` se mostrará por pantalla el mensaje `Aviso 2`. Finalmente si el carácter introducido es `'t'` o `'T'` se mostrará por pantalla el mensaje `Aviso 3`.

Si el carácter introducido no es ninguno de los anteriores se ejecutará el grupo de instrucciones asociado a la etiqueta `default`, que en este caso consta de una única instrucción que muestra por pantalla el mensaje

```
Entrada errónea
```

♦

1.8 FUNCIONES

Una *función* es un segmento de programa que realiza determinadas tareas bien definidas. Todo programa en C consta de una o más funciones. Una de estas funciones debe ser la función principal `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. Es decir, 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. Una función puede devolver un puntero a cualquier tipo de datos. 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, prototipo y acceso a una función

De forma general la definición de una función tiene la siguiente forma:

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

En esta definición se observan dos componentes principales: la primera línea y el cuerpo de la función.

La primera línea de la definición de una función contiene la especificación del tipo de valor devuelto por la función (`tipo`), seguido del nombre de la función (`nombre_función`) y un conjunto de argumentos (`tipo1 arg1, tipo2 arg2,...,tipoN argN)`, separados por comas y encerrados entre paréntesis. Cada argumento viene precedido por su declaración de tipo. Es posible definir una función que no requiera argumentos en dicho caso al nombre de la función le seguirán un par de paréntesis vacíos. Los tipos de datos se suponen enteros sino se indican explícitamente. Sin embargo, la omisión de los tipos de datos se considera una práctica de programación poco elegante.

Los argumentos de la definición de una función se denominan *argumentos o parámetros formales*, ya que representan los nombres de los elementos que se transfieren a la función desde la parte del programa que hace la llamada. Los identificadores que son usados como argumentos formales son locales, es decir, no son reconocidos fuera de la función.

Al resto de líneas que componen la definición de la función se le denomina el *cuerpo de la función* y contiene los siguientes elementos: la definición de diferentes variables locales, diversas proposiciones y una instrucción `return` para devolver el valor de `expresión` al punto del programa desde donde se llamó a la función. La aparición de `(expresión)` en `return` es opcional, si se omite simplemente se devuelve el control al punto de llamada, sin transferir ninguna información. Sólo se puede incluir una expresión en la instrucción por lo tanto, una función sólo puede devolver un valor al punto de llamada mediante la instrucción `return`. No es obligatorio que la instrucción `return` aparezca en la definición de una función, sin embargo su omisión se considera una práctica de programación poco elegante.

Se denomina *prototipo de una función* a la primera línea de una definición de función añadiendo al final un punto y coma. La forma general del prototipo de una función es por lo tanto:

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

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.

Al escribir el prototipo de una función es posible omitir los nombres de sus argumentos, sin embargo los tipos de datos de dichos argumentos no pueden ser omitidos ya que son esenciales. Además los nombres de los argumentos del prototipo de una función pueden ser distintos a los nombres de los argumentos de la definición de esa misma función. Es por este motivo por el que a los argumentos del prototipo de una función se les denomina en ocasiones *argumentos ficticios*.

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. Los argumentos reales pueden ser constantes, variables simples, o expresiones más complejas. El nombre de los argumentos reales puede ser distinto del nombre de los argumentos formales. El número de argumentos reales debe coincidir con el número de argumentos formales. Además cada argumento real debe ser del mismo tipo de datos que el correspondiente argumento formal. El valor de cada argumento real es transferido a la función y asignado al correspondiente argumento formal.

♦ **Ejemplo 1.36:**

Considérese el siguiente programa escrito en lenguaje C que calcula el factorial de un número entero positivo menor de 26 que debe ser introducido por el usuario a través del dispositivo estándar de entrada (usualmente el teclado).

```
#include <stdio.h>
double factorial(int x);
main()
{
      int n, ok=1;

      while (ok==1)
      {
            printf("\n n= ");
            scanf("%d", &n);
            if (n<=25)
            {
                  if (n<0) n=-n;
                  printf("\n n!= %.0f", factorial(n));
                  ok=0;
            }
```

```
            else
                  printf("\n Error n>25\n");
      }
}

double factorial(int x)
{
      int i;
      double prod=1;
      if (x>1)
            for(i=2;i<=x;++i)
                  prod *=i;
      return(prod);
}
```

En este programa se definen dos funciones: `main` y `factorial`. El acceso que se realiza a la función `factorial` en este programa tiene como argumento real al entero `n` cuyo valor es pasado al argumento formal `x`. Obsérvese por lo tanto que no es necesario que el nombre del argumento real coincida con el del argumento formal pero si debe ser el mismo tipo de dato.

El prototipo de `factorial` aparece antes de la definición de la función `main`, esto indica al compilador que más adelante en el programa se definirá una función `factorial`, que acepta una cantidad entera y devuelve un número en coma flotante de doble precisión. Nótese que aunque el factorial de un número entero *n* es otro número entero para evitar errores numéricos se ha usado el tipo `double` (64 bits) en vez del tipo `int` (8 bits) o `long int` (32 bits).

Otras formas posibles de escribir el prototipo de la función `factorial` serían:

```
double factorial(int);
double factorial(int var);
```

En el primer caso se ha omitido el nombre del argumento, mientras que en el segundo se ha utilizado otro nombre distinto para dicho argumento.

♦

1.8.2 Paso de argumentos a una función

1.8.2.1 Formas de pasar 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 la 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, modificados dentro de ella y luego devueltos modificados al programa. 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

Cuando los punteros se utilizan como argumentos formales de una función deben ir precedidos por un asterisco `*`. Esta regla se aplica también a los prototipos de las funciones.

♦ **Ejemplo 1.37:**

Considérese el siguiente programa escrito en lenguaje C

```
#include <stdio.h>
void f1(int u, int v);     /*Prototipo de la función f1*/
void f2(int *pu, int *pv); /*Prototipo de la función f2*/
main()
{
      int x=2;
      int y=4;
      printf("\nAntes de la llamada a f1:  x=%d  y=%d", x, y);
      f1(x,y);
      printf("\nDespués de la llamada a f1:  x=%d  y=%d", x, y);
      printf("\nAntes de la llamada a f2:  x=%d  y=%d", x, y);
      f2(&x,&y);
      printf("\nDespués de la llamada a f2:  x=%d  y=%d", x, y);
}
void f1(int u, int v)
{
      u=0;
      v=0;
```

```
        printf("\nDentro de f1:  u=%d  v=%d", u, v);
        return;
}
void f2(int *pu, int *pv)
{
        *pu=0;
        *pv=0;
        printf("\nDentro de f2:  *pu=%d  *pv=%d", *pu, *pv);
        return;
}
```

Este programa contiene dos funciones `f1` y `f2`. La función `f1` tiene como argumentos dos variables enteras. Estas variables tienen asignado inicialmente los valores `2` y `4`, respectivamente. Los valores son modificados a `0` y `0` dentro de `f1`. Sin embargo, los nuevos valores no son reconocidos en `main`, ya que los argumentos fueron pasados por valor y cualquier cambio sobre los argumentos únicamente es local a la función en la cual se han producido los cambios.

La función `f2` tiene como argumentos dos punteros a variables enteras. Dentro de esta función los contenidos de las direcciones apuntadas son modificados a `0` y `0`. Como estas direcciones son reconocidas tanto en `f2` como en `main`, los nuevos valores serán reconocidos dentro de `main` tras la llamada a `f2`. Por lo tanto, las variables enteras `x` e `y` habrán cambiado sus valores de `2` y `4` a `0` y `0`.

De acuerdo con el análisis realizado la ejecución de este programa genera la siguiente salida por el dispositivo de salida estándar:

```
Antes de la llamada a f1: x=2  y=4
Dentro de f1:  x=0  y=0
Después de la llamada a f1:  x=2  y=4
Antes de la llamada a f2:  x=2  y=4
Dentro de f2:  x=0  y=0
Después de la llamada a f2:  x=0  y=0
```

Finalmente, comentar que otra forma equivalente de escribir los prototipos de las funciones `f1` y `f2` es omitiendo los nombres de los argumentos:

```
void f1(int, int);
void f2(int *, int *);
```

♦

1.8.2.2 Paso de arrays a una función

Para pasar un array (unidimensional o multidimesional) a una función como argumento real únicamente se escribe el nombre del array sin corchetes ni índices. Recuérdese que el nombre de un array representa la dirección del primer elemento del array y por lo tanto se trata como un puntero cuando se pasa a una función. En

consecuencia, el paso de arrays como parámetros reales de funciones siempre se realiza por referencia.

Por otra parte, cuando se declara un array multidimensional como un argumento formal se debe especificar el tamaño en todos los índices excepto en el primero (el situado más a la izquierda) cuyo par de corchetes se deja vacío. El prototipo correspondiente debe escribirse de la misma manera.

♦ **Ejemplo 1.38:**

Considérese el siguiente boceto de programa escrito en lenguaje C

```
float fun1(int x, float a[], int b[][3]);
main()
{
      int n;
      float vector[4];
      int matriz[2][3];
      ...
      r=fun1(n,vector,matriz);
      ...
}
float fun1(int x, float a[], int b[][3])
{
...
}
```

Dentro de la función `main` existe una llamada a la función `fun1`, que tiene tres argumentos reales: la variable entera `n`, el array unidimensional de cuatro números en coma flotante `vector` y el array bidimensional de 2 filas y 3 columnas de números enteros `matriz`. Obsérvese que ni `vector` ni `matriz` incluyen sus corchetes e índices.

Por otra parte, la primera línea de la definición de la función incluye tres argumentos formales: la variable entera `x`, el array unidimensional de números en coma flotante `a` y el array bidimensional de números enteros `b`. Obsérvese que dentro de la declaración formal del argumento `a` no se especifica el tamaño del array, aparece únicamente el par de corchetes vacío. Asimismo dentro de la declaración formal del argumento `b` no se especifica el primer índice apareciendo su par de corchetes asociados vacíos.

Finalmente comentar que el prototipo de `fun1` se podría haber escrito equivalentemente omitiendo los nombres de los argumentos:

```
float fun1(int, float[], int[][3]);
```

♦

1.8.2.3 Paso de estructuras o uniones a una función

Una estructura o una unión se pueden transferir por referencia a una función pasando un puntero a la estructura como argumento. En este caso si cualquiera de los miembros de una estructura es alterado dentro de la función, las alteraciones serán reconocidas fuera de la función. Asimismo, una estructura se puede pasar por valor a una función. De este modo si cualquiera de los miembros de la estructura es alterado dentro de la función, las alteraciones no serán reconocidas fuera de la función. Los compiladores más recientes de C permiten que una estructura completa sea transferida directamente a una función como argumento y devuelta directamente mediante la instrucción `return`.

También es posible pasar los miembros de una estructura como argumentos de la llamada a una función. Asimismo, un miembro de una estructura puede ser devuelto por una función mediante una instrucción `return`.

♦ **Ejemplo 1.39:**

Considérese el siguiente programa escrito en lenguaje C:

```
#include <stdio.h>
typedef struct{
                  long int dni;
                  float nota;
            } datos;
void cambiar(datos *a);
main()
{
      datos alumno={70534213, 7.5};
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
      cambiar(&alumno);
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
}
void cambiar(datos *a)
{
      a->dni=23789345;
      a->nota=8.0;
      return;
}
```

Este programa ilustra la transferencia de una estructura a una función por referencia. Se tiene una estructura `alumno` del tipo `datos` cuyos miembros tienen asignado unos valores iniciales. Cuando el programa se ejecuta en primer lugar se visualizan en la pantalla los valores iniciales de los

miembros de la estructura. A continuación se llama a la función `cambiar` pasándole como argumento la dirección de la estructura, es decir, un puntero. Dentro de esta función se le asignan nuevos valores a los miembros de la estructura. Finalmente estos nuevos valores son visualizados en la pantalla. De acuerdo con el funcionamiento comentado la ejecución de este programa genera la siguiente salida por pantalla:

```
70534213 7.5
23789345 8.0
```

♦

♦ **Ejemplo 1.40:**

Considérese el siguiente programa escrito en C:

```
#include <stdio.h>
typedef struct {
                  long int dni;
                  float nota;
            } datos;
void cambiar(datos a);
main()
{
      datos alumno={70534213, 7.5};
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
      cambiar(alumno);
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
}
void cambiar(datos a)
{
      a.dni=23789345;
      a.nota=8.0;
      return;
}
```

Este programa es similar al del ejemplo anterior pero ha sido convenientemente modificado para ilustrar la transferencia de una estructura a una función por valor. Nótese como ahora la función `cambiar` acepta una estructura del tipo `datos` en vez de un puntero a este tipo de estructura. Cuando se ejecuta este programa, se obtiene la siguiente salida:

```
70534213 7.5
70534213 7.5
```

Se comprueba que al haber pasado la estructura por valor a la función, los cambios realizados a los miembros de la estructura dentro de la función no son reconocidos fuera de la misma. Si se dispone de un compilador de C reciente toda la estructura puede ser devuelta al punto de llamada de la función. Simplemente habría que hacer algunos pequeños cambios en el código del programa:

```
#include <stdio.h>
typedef struct {
                  long int dni;
                  float nota;
            } datos;
datos cambiar(datos a);
main()
{
      datos alumno={70534213, 7.5};
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
      alumno=cambiar(alumno);
      printf("%ld %.1f\n", alumno.dni, alumno.nota);
}
datos cambiar(datos a)
{
      a.dni=23789345;
      a.nota=8.0;
      return(a);
}
```

El prototipo y la primera línea de la definición de `cambiar` deben especificar el tipo `datos` como tipo de salida. La llamada a la función `cambiar` debe escribirse en la parte derecha de una expresión de asignación. La instrucción `return` dentro de la definición de `cambiar` debe devolver la estructura `a`.

La salida de este programa sería:

```
70534213 7.5
23789345 8.0
```

♦

La mayoría de los compiladores de C permiten transferir estructuras de datos complejas libremente entre funciones. Sin embargo, algunos compiladores pueden tener dificultades al ejecutar programas que involucran transferencias de estructuras de datos complejas, debido a ciertas restricciones de memoria.

1.8.3 Devolución de un puntero por una función

Una función puede devolver un puntero de cualquier tipo de dato a la parte del programa que hizo la llamada a una función. Esta característica puede ser útil, por ejemplo, cuando se pasan varias estructuras a una función y sólo una de ellas es devuelta.

♦ **Ejemplo 1.41:**

El prototipo

```
char *func(char *s);
```

declara una función llamada `func` que acepta un puntero `s` a carácter y devuelve un puntero a carácter.

El prototipo

```
int *p(int b, float *C);
```

declara una función llamada `p` que acepta un entero `b` y un puntero `C` a un número en coma flotante. La función devuelve un puntero a un número entero.

♦

♦ **Ejemplo 1.42:**

Considérese el siguiente programa escrito en C:

```
#include <stdio.h>
#define T 3
#define NULL 0
typedef struct {
                  long int dni;
                  float nota;
            } datos;
datos *buscar(datos a[], long int b);
main()
{
      datos alumnos[T]={
                        {70534213, 7.5},
                        {33356897, 8.5},
                        {85963472, 7.0}
                        };
      long int id;
      datos *pr;
      printf("\nIntroduzca el DNI del alumno: ");
      scanf("%ld", &id);
      pr=buscar(alumnos, id);
      if (pr!=NULL)
            printf("Nota[%ld]= %.1f\n", pr->dni, pr->nota);
      else
            printf("\nDNI no encontrado");
}
datos *buscar(datos a[], long int b)
```

```
{
        int h;
        for(h=0;h<T;++h)
        if(a[h].dni==b)
                return(&a[h]);
        else
                return(NULL);
}
```

En este programa dentro de la función `main` se define el array `alumnos` de tres estructuras del tipo `datos` cuyos miembros `dni` y `nota` tienen asignados unos valores iniciales. También se define un entero largo `id` y un puntero a una estructura del tipo `datos`. Cuando el programa se ejecuta en primer lugar se visualiza en la pantalla el mensaje

```
Introduzca el DNI del alumno:
```

En segundo lugar el programa se queda a la espera de que el usuario introduzca un entero largo y pulse la tecla de salto de línea. El valor introducido se asigna a la variable `id`. En tercer lugar, se llama a la función `buscar` pasándole como argumentos reales la dirección del array de estructuras `alumnos` y el valor de la variable `id`. Comienza por lo tanto a ejecutarse la función `buscar` que lo que hace es recorrer todo el array de estructuras hasta encontrar alguna cuyo campo `dni` coincida con `id`. Si se produce alguna coincidencia, la función devuelve a `main` un puntero a dicha estructura del array, se muestra en la pantalla el mensaje

```
Nota[dni]= nota
```

y el programa finaliza.

Si no se produce ninguna coincidencia después de buscar en todo el array, entonces la función `buscar` devuelve el valor `NULL` (cero) a `main`, se muestra por pantalla el mensaje

```
DNI no encontrado
```

y el programa finaliza.

♦

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. Una aplicación típica del paso de punteros a funciones es el tratamiento de las interrupciones. Usando punteros a funciones se puede capturar una interrupción y tratarla usando una función determinada.

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.

La declaración de la función anfitriona se puede escribir de forma general de la siguiente forma:

```
tipo_a nombre_fun_a (puntero_fun_h, otros)
```

donde `tipo_a` es el tipo de dato que devuelve la función anfitriona cuyo nombre es `nombre_fun_a`. Esta función recibe como argumentos formales un puntero a la función huésped (`puntero_fun_h`). También puede recibir, como cualquier otra función, varios argumentos formales de diferentes tipos (`otros`).

La declaración de un puntero a una función huésped como argumento formal (`puntero_fun_h`) se puede escribir cómo:

```
tipo_h(*nombre_fun_h)(tipo1 arg1, tipo2 arg2,..., tipoN argN)
```

donde `tipo_h` es el tipo de dato de la cantidad devuelta por la función huésped, `nombre_fun_h` es el nombre de la función huésped, `tipo1, tipo2,..,tipoN` se refieren 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.

La declaración del argumento formal `puntero_fun_h` también se puede escribir omitiendo el nombre de los argumentos formales:

```
tipo_h(*nombre_fun_h)(tipo1, tipo2,..., tipoN);
```

Para acceder a la función huésped dentro de la función anfitriona el operador `*` debe preceder al nombre de la función huésped y ambos deben estar encerrados entre paréntesis, es decir:

```
(*nombre_fun_h)(argumento1, argumento2,..., argumentoN);
```

donde `argumento1, argumento2,..., argumentoN` son los argumentos reales de la llamada a la función.

♦ **Ejemplo 1.43:**

Considérese el siguiente programa escrito en C:

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

```
float a1(float (*b)(float a));
float a2(float a);
main(){
      int u[3]={501,501,0};
      float z=0;
      z=a1(a2);
      printf("\n[%d, %d, %d, %.2f]\n",u[0],u[1],u[2],z);
}
float a1(float (*b)(float a))
{
      float u=30.4;
      u=(*b)(u);
      return(u);
}
float a2(float a){
      int h;
      for(h=-1;h<3;h++)
            a=0.5*a;
      return(a);
}
```

Cuando se ejecuta este programa en primer lugar se invoca a la función anfitriona `a1` pasándole como argumento real la dirección de memoria de la función huésped `a2`.

La primera acción que se realiza al ejecutar la función `a1` es invocar a la función huésped `a2` pasándole como argumento real el valor inicial de la variable `u`, que es 30.4. Se comienza a ejecutar la función `a2`. La primera acción asociada a `a2` es ejecutar un bucle 4 veces dentro del cual se multiplica el valor de la variable `a` por 0.5. El resultado de la multiplicación se asigna a la variable `a`. En la primera ejecución del bucle (`h=-1`) `a`=0.5*30.4=15.2, en la segunda ejecución (`h=0`) `a`=0.5*15.2=7.6, en la tercera ejecución (`h=1`) `a`=0.5*7.6=3.8 y en la cuarta ejecución (`h=2`) `a`=0.5*3.8=1.9.

Finalizado el bucle se ejecuta la instrucción `return(a)` con lo que la función `a2` finaliza devolviendo 1.9 como valor de salida que es asignado a la variable `u` de la función `a1`. A continuación se ejecuta la instrucción `return(u)` de la función `a1` con lo que finaliza devolviendo el valor 1.9 como valor de salida que es asignado a la variable `z` del código principal.

Por último se imprime en pantalla el mensaje

```
[501, 501, 0, 1.90]
```

y el programa finaliza su ejecución.

Comentar que el prototipo de la función anfitriona `a1` se podría haber escrito omitiendo el nombre de la función huesped y el nombre de su argumento:

```
float a1(float (*)(float));
```

Asimismo la primera línea de la declaración de la función anfitriona se podría haber escrito omitiendo el nombre del argumento de la función huésped:

```
float a1(float (*b)(float))
```

♦

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 primera línea de la definición de `main` es:

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

El significado de los argumentos formales de `main` es el siguiente:

- `int argc`. Es un número entero que contiene el número de argumentos de la línea de comandos. Como mínimo este argumento puede valer uno, puesto que el nombre del programa cuenta como primer argumento.

- `char *argv[]`. Es un array de punteros a caracteres. Cada puntero 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. Por ejemplo, la función `atoi` incluida en el fichero de cabecera `stdlib.h` convierte una cadena de caracteres numéricos en un número entero.

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 1.44:**

Considérese 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);
            }
      }
```

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 ASIGNACIÓN DINÁMICA DE MEMORIA

Se denomina *asignación dinámica de memoria* a la acción de guardar un espacio de memoria de tamaño variable durante la ejecución de un programa. Un ejemplo típico donde es necesario realizar la asignación dinámica de memoria es cuando se usa un array de punteros para implementar un array multidimensional.

Las funciones `malloc` y `free` definidas en el fichero de 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 1.45:**

Considérese 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));
}
```

Este programa muestra por pantalla el número de bytes asignados por el compilador a los tipos de datos `int`, `float`, `double` y `char`:

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

Es importante observar que el número de bytes asignado a cada tipo de datos puede variar dependiendo del compilador utilizado.

♦

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

```
free(ptr);
```

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

♦ **Ejemplo 1.46:**

Considérese el siguiente programa escrito en C:

```
#include <stdio.h>
#include <stdlib.h>
void fun1(float a[],int b[][3]);
void fun2(float *a, int *b[2]);
main()
{
        float vector[]={1.5,2.5,3.5};
        int matriz[2][3]={
                        {2, 1, 3},
                        {4, 5, 6}
                };
        int *d[2];
        d[0]=(int *) malloc(3*sizeof(int));
        d[1]=(int *) malloc(3*sizeof(int));
        *(d[0])=2;
        *(d[0]+1)=1;
        *(d[0]+2)=3;
        *(d[1])=4;
        *(d[1]+1)=5;
        *(d[1]+2)=6;
        fun1(vector, matriz);
        fun2(vector, d);
```

```
        free(d[0]);
        free(d[1]);
}
void fun1(float a[],int b[][3])
{
  printf("\n----Fun1----\nVector: %g  %g  %g\n",a[0],a[1],a[2]);
  printf("Fila 1: %d  %d  %d\n",b[0][0],b[0][1],b[0][2]);
}
void fun2(float *a, int *b[2])
{
  printf("\n----Fun2----\nVector: %g  %g  %g\n",*a,*(a+1),*(a+2));
  printf("Fila 1: %d  %d  %d\n",*b[0],*(b[0]+1),*(b[0]+2));
  printf("Fila 1: %d  %d  %d\n",*(*b),*(*(b)+1),*(*(b)+2));
  printf("Fila 2: %d  %d  %d\n",*b[1],*(b[1]+1),*(b[1]+2));
  printf("Fila 2: %d  %d  %d\n",*(*(b+1)),*(*(b+1)+1),*(*(b+1)+2));
}
```

En este programa ilustra como es posible usar un array de punteros para implementar un array multidimensional y como en dicho caso es necesario realizar una asignación dinámica de memoria. Se tiene un array `d` de dos punteros a números enteros con el que se desea implementar un array bidimensional de dos filas y tres columnas. En consecuencia es necesario reservar memoria para los tres enteros de la primera fila cuyo elemento 0 (primer elemento) es apuntado por `d[0]`y los tres enteros de la segunda fila cuyo primer elemento es apuntado por `d[1]`. Este programa muestra la siguiente traza de ejecución en pantalla:

```
----Fun1----
Vector: 1.5  2.5  3.5
Fila 1: 2  1  3
----Fun2----
Vector: 1.5  2.5  3.5
Fila 1: 2  1  3
Fila 1: 2  1  3
Fila 2: 4  5  6
Fila 2: 4  5  6
```

♦

COMPLEMENTO 1.A

Forma alternativa del uso de punteros para referirse a un array multidimensional

Los arrays multidimensionales pueden implementarse alternativamente como un puntero a un grupo de arrays unidimesionales contiguos en vez de como un array de punteros. Así la declaración de un array multidimensional de orden N

```
tipo_array nombre_array[rango1][rango2]...[rangoN];
```

puede sustituirse equivalentemente por:

```
tipo_array (*nombre_puntero)[rango2]...[rangoN];
```

Obsérvese que el nombre del puntero al array y el asterisco que le precede van entre paréntesis. Esto no es algo arbitrario sino que realmente la escritura de estos paréntesis es necesaria, ya que en caso contrario se estará definiendo un array de punteros en vez de un puntero a un grupo de arrays. Los corchetes y el asterisco se evalúan normalmente de derecha a izquierda. Además el índice `[rango1]` ya no se escribe.

Al igual que ocurría cuando se utilizaba un array de punteros para implementar un array multidimensional, si se utiliza un puntero a un grupo de arrays unidimesionales contiguos la reserva de memoria la debe realizar el programador de forma explícita en el código del programa mediante el uso de la función `malloc`.

Se puede acceder a un elemento individual de un array multidimensional mediante la utilización repetida del operador `*`.

♦ **Ejemplo 1A.1:**

Considérese que `z` es un array bidimensional de números en coma flotante con 3 filas y 4 columnas. Se puede declarar `z` como

```
float z[3][4];
```

o equivalentemente como

```
float (*z)[4];
```

En el segundo caso, `z` se define como un puntero a un grupo de arrays unidimensionales consecutivos de 4 elementos en coma flotante. Así `z` apunta al primero de los arrays de 4 elementos, que es en realidad la primera fila (fila 0) del array bidimensional original. Análogamente (`z+1`) apunta al segundo array de 4 elementos, que es la segunda fila (fila 1) del array bidimensional original, y así sucesivamente.

Para acceder al elemento de la fila 2 situado en la columna 3 (`z[2][3]`) se puede escribir

```
*(*(z+2)+3)
```

El significado de esta expresión (ver Figura 1A.1) es el siguiente: `(z+2)` es un puntero a la fila 2. Como la fila 2 es un array unidimensional `*(z+2)` es realmente un puntero al primer elemento de la fila 2. Se le suma 3 a ese puntero, Por lo tanto, `(*(z+2)+3)` es un puntero al elemento 3 (el cuarto elemento) de la fila 2. Luego `*(*(z+2)+3)` se refiere al elemento en la columna 3 de la fila 2, que es `z[2][3]`.

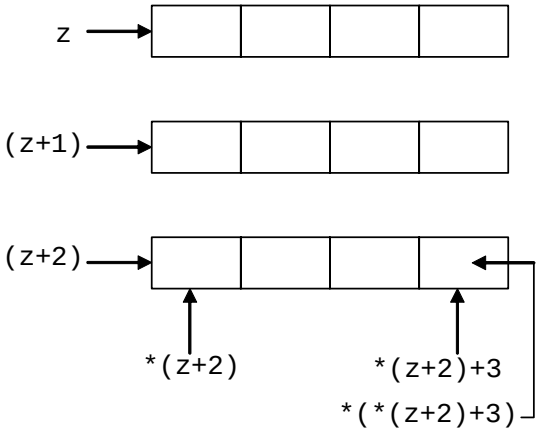


Figura 1A.1: Forma alternativa del uso de punteros para referirse a un array bidimensional de 3 filas y 4 columnas.

♦

COMPLEMENTO 1.B
Macros

La directiva del preprocesador `#define` aparte de para definir constantes 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 1B.1:**

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);
}
```

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");
        }
}
```

♦

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).

COMPLEMENTO 1.C
Principales archivos de cabecera

El lenguaje C dispone de un gran número de *funciones de biblioteca* que realizan varias operaciones y cálculos de uso frecuente. Las funciones de biblioteca de propósitos relacionados se suelen encontrar agrupadas en programas objeto en *archivos de biblioteca o librerías* separados. Estos archivos de biblioteca se proporcionan como parte de cada compilador de C. Las prototipos de todas las funciones que forman parte de una misma librería se encuentran agrupados en un archivo denominado *archivo de cabecera* que se denota con la extensión `.h`. En este archivo también se pueden incluir: declaraciones de constantes, declaraciones de tipos de datos y macros. Las principales archivos de cabecera incluidos en la mayoría de los compiladores de C son:

- `<alloc.h>`. Contiene los prototipos de funciones para obtener y liberar memoria.
- `<ctype.h>`. Contiene los prototipos de funciones que indican características de los caracteres, por ejemplo, si está en mayúscula o en minúscula.
- `<errno.h>`. Contiene la definición de varias constantes y variables, entre ellas la variable global `errno`. Esta variable contiene el identificador numérico del error que se ha producido durante la ejecución de una llamada al sistema.

- `<fcntl.h>`. Define los indicadores o flags para los modos de apertura de un fichero.
- `<float.h>`. Establece algunas propiedades de las representaciones del tipo coma flotante.
- `<limits.h>`. Contiene macros que determinan varias propiedades de las representaciones de tipos enteros.
- `<math.h>`. Contiene los prototipos de funciones matemáticas elementales, entre ellas, las funciones trigonométricas, exponenciales y logarítmicas.
- `<stdarg.h>`. Contiene los prototipos de funciones que permiten acceder a los argumentos adicionales sin nombre en una función que acepta un número variable de argumentos.
- `<stdio.h>`. Incluye macros y los prototipos de funciones para realizar operaciones de entrada y salida sobre ficheros y flujos de datos.
- `<stdlib.h>`. Contiene los prototipos de funciones estándar, por ejemplo para convertir números a cadenas de caracteres o para realizar la asignación dinámica de memoria. (algunas de éstas también están declaradas en `alloc.h`.
- `<string.h>`. Contiene los prototipos de funciones para manejar cadenas de caracteres.
- `<time.h>`. Contiene los prototipos de funciones para manejar fechas.

En el Apéndice B se incluye un listado con las funciones de bibliotecas de uso más frecuente.

COMPLEMENTO 1.D

Compilación con gcc de un programa que consta de varios ficheros

Es una práctica frecuente en programación el descomponer la escritura de un programa en varios ficheros con objeto de tener una visión más clara del mismo. En el caso del compilador `gcc` de C bajo UNIX para compilar un programa que consta de varios ficheros se debe teclear la siguiente orden desde la línea de comandos:

```
$ gcc fichero1.c fichero2.c   ficheroN.c –o nombre_ejecutable
```

♦ **Ejemplo 1D.1:**

Supóngase que un programa se ha escrito en tres ficheros: `ejemplo.h`, `parte1.c` y `parte2.c`. El código del fichero de cabecera `ejemplo.h` es:

```
#define T 3
#define NULL 0
typedef struct {
                  long int dni;
                  float nota;
            } datos;
datos *buscar(datos a[], long int b);
```

El código del fichero `parte1.c` es:

```
#include <stdio.h>
#include "ejemplo.h"
main()
{
      datos alumnos[T]={
                        {70534213, 7.5},
                        {33356897, 8.5},
                        {85963472, 7.0}
                        };
      long int id;
      datos *pr;
      printf("\nIntroduzca DNI del alumno: ");
      scanf("%ld", &id);
      pr=buscar(alumnos, id);
      if (pr!=NULL)
            printf("Nota[%ld]= %.1f\n", pr->dni, pr->nota);
      else
            printf("\nDNI no encontrado\n");
}
```

El código del fichero `parte2.c` es:

```
#include "ejemplo.h"
datos *buscar(datos a[], long int b)
{
      int h;
      for(h=0;h<T;++h)
            if(a[h].dni==b)
                  return(&a[h]);
            else
                  return(NULL);
}
```

Si se desea compilar con `gcc` este programa, al que se le va a llamar `busca_notas`, se debe escribir la siguiente orden desde la línea de comandos:

```
$ gcc parte1.c parte2.c -o busca_notas
```

Obsérvese que no hace falta incluir en la orden el nombre fichero de cabecera ya que está incluido dentro de cada fichero `.c`. Asimismo nótese que el fichero de cabecera está escrito entre comillas (""). Originalmente esto se hacía para indicarle al compilador que se trata de un archivo de cabecera de usuario y diferenciarlo así de los archivos de cabecera del sistema que están escritos entre los signos menor y mayor (<>). La mayoría de los compiladores de C y los entornos de desarrollo actuales permiten especificar donde se encuentran los distintos archivos de cabecera. Sin embargo se sigue recomendando usar la misma nomenclatura por cuestiones de claridad en el código.

♦

CAPÍTULO 2

Consideraciones generales del sistema operativo UNIX

2.1 INTRODUCCIÓN

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 la 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 una interfaz de usuario sencilla 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 pueden 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 Sotfware Foundation, *SunOS* y *Solaris* de Sun Microsystems, etc. Además dentro de cada distribución existen diferentes versiones.

En este capítulo en primer lugar se detalla la historia del sistema UNIX, su lectura aclarará, sin duda, el porqué de la existencia de tantas distribuciones. En segundo lugar se describe la arquitectura de UNIX. En tercer lugar se enumeran los principales servicios prestados por el núcleo. En cuarto lugar 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 incluye una primera introducción

a dos de los principales eventos que son atendidos en modo núcleo: las interrupciones y las excepciones. En quinto lugar se describe la estructura del sistema operativo UNIX. En sexto lugar se introduce la interfaz de usuario para el sistema de ficheros. El capítulo finaliza con un par de complementos, el primero está dedicado a comentar las funciones más importantes de la librería estándar de funciones de entrada/salida de C. El segundo complemento relata, a modo de curiosidad, el origen del término *proceso demonio*.

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, comenzó 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 (s5fs). 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 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 universitaria 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 desarrolló 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 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 adaptaciones 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 la 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), al 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, una de las 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 de la 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 numerosos y diversos como las versiones de UNIX. Finalmente, la mayoría de los vendedores se puso 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 BSD4.3. 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 de la interfaz entre los programadores y el sistema operativo y no de cómo 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 anunció 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 Fundation*) 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 una interfaz de usuario gráfico llamada *Motif*, que fue muy bien recibida. 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 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 intérprete 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 (en el momento de editar este libro).

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.

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.

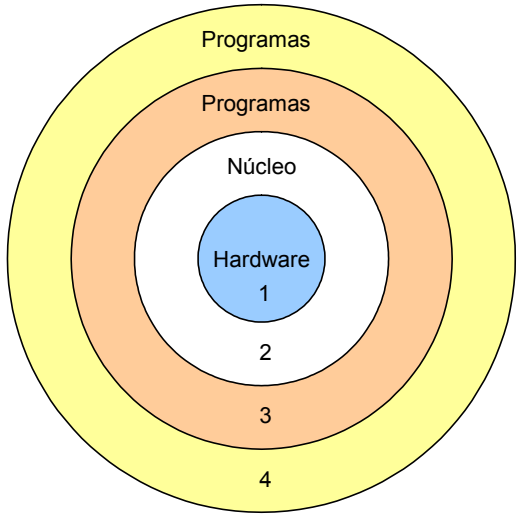


Figura 2.1: Arquitectura del sistema operativo UNIX

En definitiva, las *llamadas al sistema* son el mecanismo que los programas utilizan para solicitar el 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 por qué 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 NÚCLEO

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 al 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 EJECUCIÓN

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 proceso 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 necesario 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 esta 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 0[1].

♦ **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 los demás tipos de interrupciones estarán permitidas.

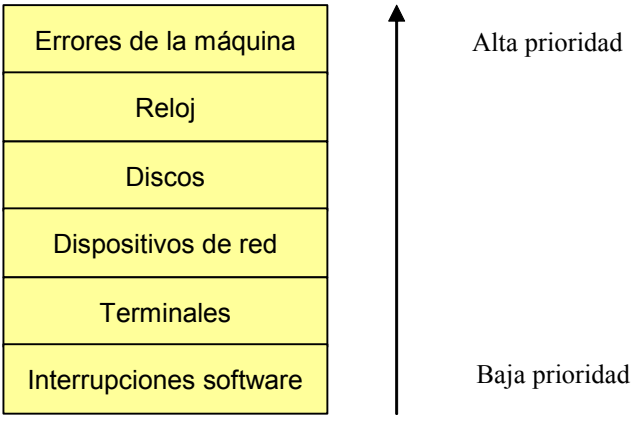


Figura 2.2: Niveles de prioridad de interrupción típicos

♦

[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

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.

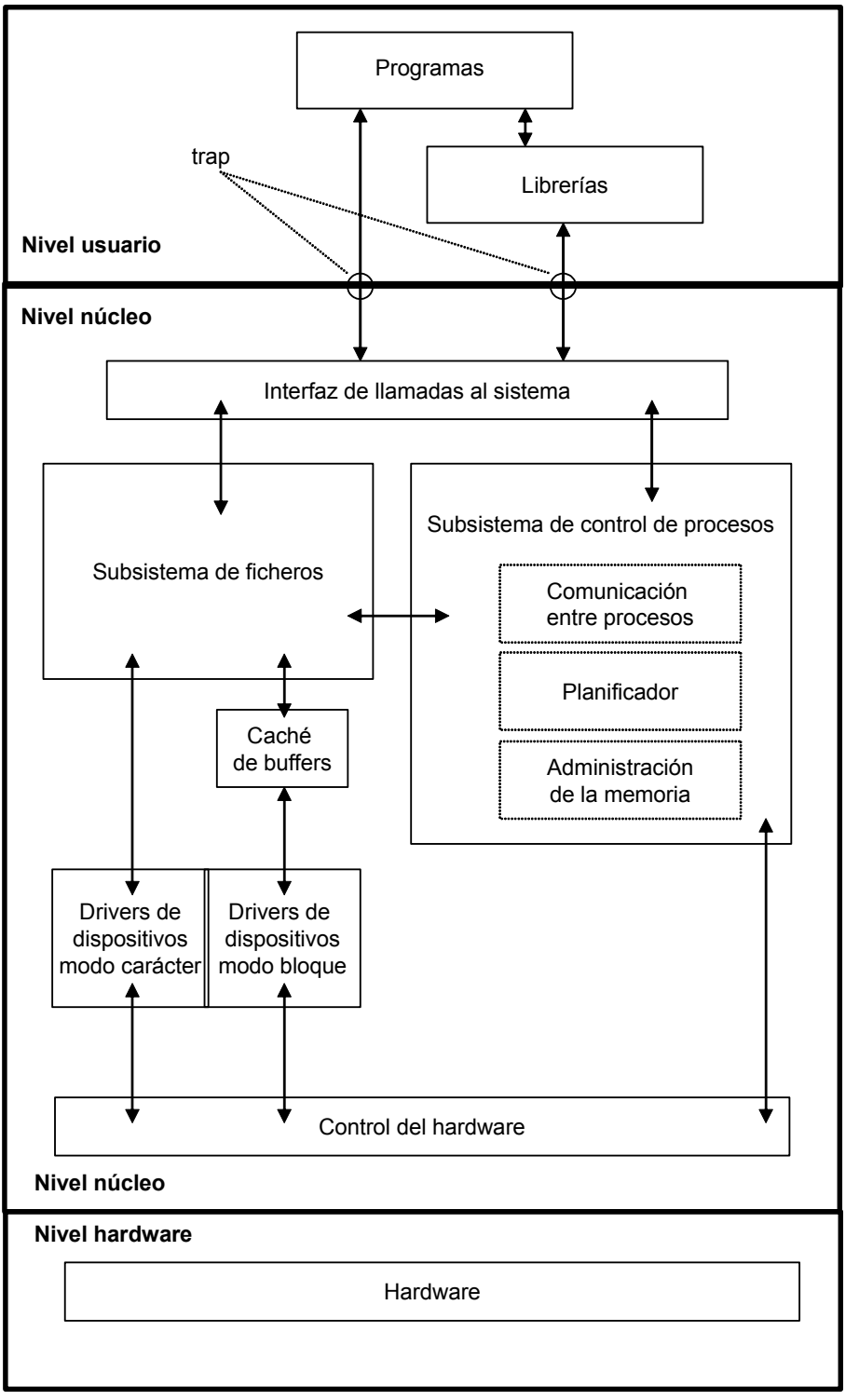


Figura 2.3: Estructura del sistema operativo UNIX

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.

La 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 una interfaz bien definido, que encapsula la visión que tiene el usuario del sistema de ficheros. Además esta 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. La 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 la *interfaz nodo-v/sfv* que permite a UNIX soportar diferentes tipos de sistemas de ficheros tanto UNIX (sf5s, FFS, etc) como DOS (fat). Esta interfaz será objeto de estudio en el Capítulo 8.

En UNIX hay diferentes tipos de ficheros: ordinarios (también denominados regulares o de datos), directorios, enlaces simbólicos, tuberías, 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 FIFO (First In First Out)) 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 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, 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 procesos, 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 LA 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.

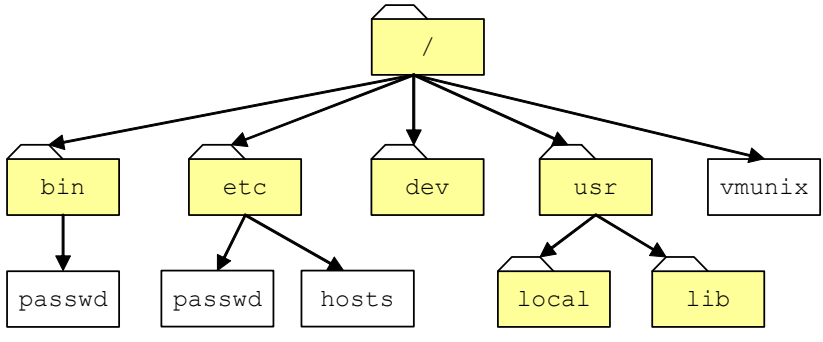


Figura 2.4. Un ejemplo de árbol de directorios

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 por 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 es interpretada 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, 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 por qué 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 cuál 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).

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 debe asignar al identificador de usuario efectivo *euid* del proceso 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.

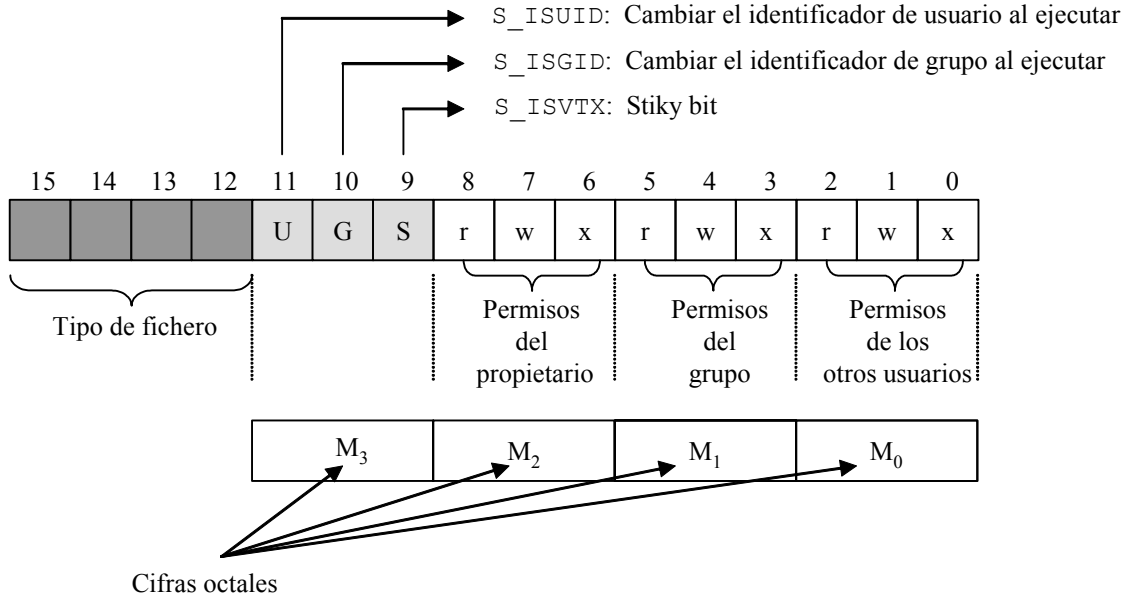


Figura 2.5: Máscara de modo de un fichero

- *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 y listar el contenido del mismo.

    - 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 un shell script. Para directorios, el permiso de ejecución permite al usuario cambiar al directorio en cuestión.

De los 16 bits de la máscara de modo de un fichero, el propietario del fichero o el superusuario (ver sección 3.4.1) 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 x *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 x 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 para cada máscara de modo expresada 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`. Los bits `S_ISUID`, `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 (`stout`), 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.

UNIX proporciona un sencillo mecanismo denominado *redirección de entrada/salida* (E/S) para cambiar la entrada y la salida estándar (ver sección 3.6.3).

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.

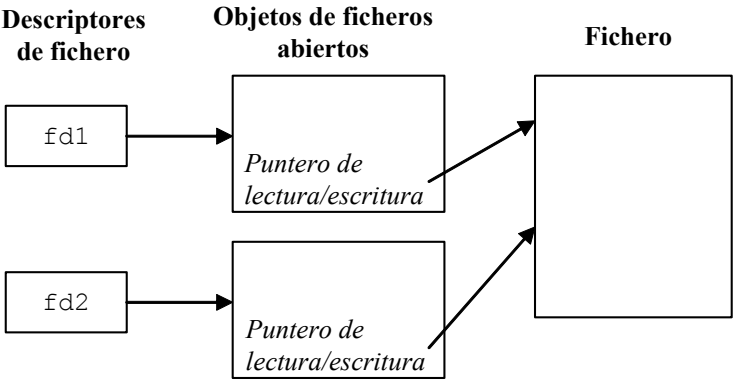


Figura 2.6: Un fichero es abierto dos veces

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 descriptores comparten la misma sesión para el fichero, ambos ven el mismo fichero y usan el mismo puntero de lectura/escritura.

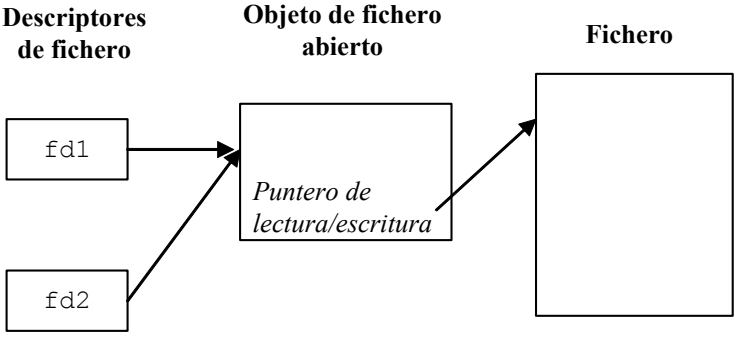


Figura 2.7: Descriptor clonado mediante las llamadas `dup`, `dup2` *o* `fork`.

De forma similar, la llamada al sistema `fork` que permite a un proceso (padre) crear otro proceso (hijo) duplica todos los descriptores 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 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 entrada/salida sobre un fichero

2.7.5.1 Apertura y cierre de un fichero

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

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

donde `ruta` 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 cero 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.

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

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

donde `path` es la ruta del fichero y `flags` puede ser o una máscara de modo octal o una máscara de bits que permiten especificar los permisos de apertura de dicho fichero. 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.

Cuando el argumento `flags` se especifica mediante una máscara de bits, ésta típicamente se implementa como una combinación de constantes enlazadas con el operador OR a nivel de bit ('`|`'). Estas constantes se encuentran definidas en el fichero de cabecera `<fcntl.h>`. En la Tabla 2.2 se muestran algunas de las constantes utilizadas más frecuentemente.

Constante	Significado
`O_RDONLY`	Abrir en modo sólo lectura
`O_WRONLY`	Abrir en modo sólo escritura
`O_RDWR`	Abrir para leer y escribir
`O_CREAT`	Crear el fichero si no existe
`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.

Tabla 2.2: Constantes definidas en el fichero de cabecera `<fcntl.h>` *utilizadas con más frecuencia*

De las constantes `O_RDONLY`, `O_WRONLY` y `O_RDWR` solo una de ellas debe estar presente al componer la máscara `flags`, de lo contrario, el modo de apertura quedaría indefinido.

Otra utilidad que presenta la llamada al sistema `open` es la de crear un fichero antes de abrirlo si éste no existe previamente. En dicho caso su sintaxis toma la siguiente forma:

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

En este caso `flags` debe incluir una de las constantes `O_RDONLY`, `O_WRONLY` y `O_RDWR` junto con la constante `O_CREAT`. Obsérvese que aparece un tercer argumento `modo` que es la máscara de modo octal $M_3M_2M_1M_0$ que especifica los permisos de acceso que serán asociados al fichero cuando sea creado.

♦ **Ejemplo 2.3:**

A continuación se incluyen algunos ejemplos de escritura de la llamada al sistema `open`

- `fd=open("texto.txt", 0666);` abre el fichero `texto.txt` con permisos de lectura y escritura para todos los usuarios.
- `fd=open("texto.txt",O_RDONLY);` abre el fichero `texto.txt` en modo sólo lectura.
- `fd=open("texto.txt",O_RDWR|O_APPEND);` abre el fichero `texto.txt` para leer y escribir. Además sitúa el puntero de lectura/escritura al final del fichero.
- `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. Además los bits `S_ISUID`, `S_ISGID` y `S_ISVTX` están desactivados.

Obsérvese que en los tres primeros ejemplos para que la llamada al sistema `open` se ejecute con éxito el fichero `texto.txt` ya debe estar creado en el directorio de trabajo actual. En caso contrario se almacenará el valor -1 en `fd`.

♦

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. Cuando se intenta leer más allá del final del fichero, `read` devuelve el valor 0.

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:**

Considérese el siguiente programa escrito en C que permite añadir una línea a un fichero ya existente en el directorio de trabajo actual.

```
      #include <fcntl.h>
      #include <stdio.h>
      main()
      {
            int ident,h,cont=0;
            char buffer[100];
[1]         for (h=0;h<100;h++)
[2]               buffer[h]='\0';
[3]         if((ident=open("escribir.txt",O_WRONLY|O_APPEND))==-1)
            {
```

```
[4]               printf("\nError de apertura\n");
[5]               exit(1);
            }
[6]         while (cont<98)
            {
[7]               buffer[cont]=getchar();
[8]               if (buffer[cont]=='\n') break;
[9]               cont=cont+1;
            }
[10]        if (cont==98) buffer[cont]='\n';
[11]        write(ident,buffer,100);
[12]        close(ident);
      }
```

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 ejecuta un bucle `for` 100 veces para asignar **[2]** a cada elemento de la cadena de caracteres `buffer` el carácter nulo '`\0`'. A continuación **[3]** se invoca a la llamada al sistema `open` para abrir el fichero `escribir.txt` en modo sólo escritura y situar el puntero de lectura/escritura al final del fichero para poder añadir datos. Si 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 **[4]** en la pantalla el mensaje

```
Error de apertura
```

e invoca **[5]** 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 **[6]** se entra en un bucle `while` cuya condición de ejecución es que la variable `cont` sea menor que `98`. En cada ejecución del bucle en primer lugar se ejecuta **[7]** la función `getchar` que solicita al usuario la introducción de un carácter por el teclado que se almacenará en un elemento de la cadena `buffer`. En segundo lugar se comprueba **[8]** si el carácter introducido es un salto de línea '`\n`'. En caso afirmativo se ejecuta una instrucción `break` y sale del bucle `while`. En caso contrario se incrementa **[9]** en una unidad la variable `cont`. Cuando se sale del bucle `while` se comprueba **[10]** si la variable `cont` es igual a `98`, en caso afirmativo se asigna a `buffer[98]` el carácter '`\n`'.

Acto seguido **[11]** se invoca a la llamada al sistema `write` para escribir el contenido de `buffer` en el fichero. Finalmente **[12]** se invoca a la llamada al sistema `close` para cerrar el fichero 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 especifico. 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 constantes definidos en el fichero de cabecera `<stdio.h>`:

- `SEEK_SET`. El puntero avanza `offset` bytes con respecto al inicio del fichero. El valor de esta constante es `0`.
- `SEEK_CUR`. El puntero avanza `offset` bytes con respecto a su posición actual. El valor de esta constante es `1`.
- `SEEK_END`. El puntero avanza `offset` bytes con respecto al final del fichero. El valor de esta constante es `2`.

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:**

Considérese el siguiente programa escrito en C que permite visualizar una determinada línea de un fichero supuesto que la longitud de las líneas de ese fichero es de 41 caracteres.

```
      # include <stdio.h>
      # define LL 41
      main(int np, char* a[])
      {
            int b, id, h;
            char alm[LL+1];
[1]         if (np==3)
```

```
            {
[3]               b=atoi(a[2]);
[4]               id=open(a[1],0600);
[5]               if ((id!=-1)&&(lseek(id,(b-1)*LL,0)!=-1)
                       &&(read(id,alm,LL)>0))
                  {
[6]                          printf("\n");
[7]                          for(h=0;h<LL;h=h+1) printf("%c",alm[h]);
[8]                          printf("\n");
                  }
                  else
[9]                     printf("\nError\n");
            }
            else
[10]              exit(1);
      }
```

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

```
$ exlseek errores.dat 3
```

En el *Cuadro 2.1* se muestran las cuatro primeras líneas del fichero `errores.dat`. Puesto que la definición hecha de `main` permite pasar parámetros al ejecutable, en primer lugar **[1]** se comprueba que la invocación del programa se ha realizado con tres parámetros (recordar que el nombre del ejecutable se considera un parámetro más). En caso negativo se invoca **[10]** a la llamada al sistema `exit` para terminar el programa.

```
Mensaje A: Error en buffer de E/S.......
Mensaje B: Error en sector del disco....
Mensaje C: Violación del segmento.......
Mensaje D: Sector defectuoso............
```

Cuadro 2.1: Cuatro primeras líneas del fichero `errores.dat`

En caso positivo, en primer lugar **[3]** se ejecuta la función de librería `atoi` para convertir el número de línea `3` introducido como parámetro de `main` de tipo carácter a tipo entero. A continuación se invoca **[4]** a la llamada al sistema `open` para abrir con permisos de lectura y escritura el fichero `errores.dat` pasado como argumento de `main`. Acto seguido **[5]**, se comprueba si la llamada `open` se ha ejecutado con éxito. Recuérdese que en dicho caso en la variable `id` se almacena el descriptor del fichero que es un número entero distinto de `-1`. También se invoca a la llamada al sistema `lseek` para posicionar el puntero de lectura/escritura al principio de la tercera línea del fichero. Asimismo se comprueba si durante la ejecución de `lseek` se ha producido algún error. Además se invoca a la llamada al sistema `read` para leer dicha línea del

fichero y almacenarla en la variable `alm`. También se comprueba si dicha llamada ha logrado leer la línea pedida. Si alguna de las comprobaciones anteriores da un resultado negativo entonces **[9]** se imprime por pantalla el mensaje

```
Error
```

y el programa finaliza.

Si las tres comprobaciones realizadas en **[10]** son positivas se muestra por pantalla: un salto de línea **[6]**, la línea pedida escribiéndola carácter a carácter a través de un bucle `for` **[7]** y otro salto de línea **[8]**. Luego en pantalla aparece

```
Mensaje C: Violación del segmento.......
```

y el programa finaliza.

♦

COMPLEMENTO 2.A
Librería estándar de funciones de entrada/salida

La librería estándar de funciones de entrada/salida, que forma parte de la definición del ANSI C, hace uso de las llamadas al sistema para presentar una interfaz de alto nivel que permite al programador trabajar con los ficheros desde un punto de vista más abstracto.

Por otra parte, con esta librería cada acceso al disco se gestiona de una forma más eficiente, ya que las funciones manejan memorias intermedias para almacenar los datos y se espera hasta que estas memorias están llenas antes de transferirlas al disco o a la caché de buffers de bloques de disco.

Se considera que un fichero es un *flujo de bytes o flujo de datos*[2] cuya información de control se encuentra almacenada en una estructura predefinida de tipo `FILE`. La definición de este tipo de estructura así como los prototipos de las funciones de entrada/salida se encuentran en el fichero de cabecera `<stdio.h>`.

A esta librería pertenecen las funciones de entrada/salida (`getchar`, `putchar`, `scanf`, `printf`, `gets` y `puts`) descritas en el Capítulo 1. Otras funciones importantes pertenecientes a esta librería son: `fopen`, `fread`, `fwrite`, `fclose`, `feof`, `fgets`, `fgetc`, `fputs`, `fputc`, `fscanf` y `fprintf`.

Algunas de las constantes más importantes definidas en esta librería son: `SEEK_SET`, `SEEK_CUR`, `SEEK_END` y `EOF`. El significado de las tres primeras constantes fue explicado en la sección 2.7.5.3. Por su parte, la constante `EOF` cuyo valor es `-1` se utiliza como valor de retorno de algunas funciones para señalar que se ha alcanzado el final de un fichero.

2.A.1 Función fopen

La declaración de la función de librería `fopen` es:

```
#include <stdio.h>
FILE *fopen(const char *nombre_fichero, const char *modo);
```

Esta función abre el fichero de nombre `nombre_fichero`, lo asocia con un flujo de datos y devuelve un puntero al mismo. Si falla, devuelve un puntero nulo. El argumento `modo` es una cadena de caracteres que le indica a `fopen` el modo de acceso al fichero.

[2] Este es la traducción al castellano que se ha elegido para el término anglosajón *stream*.

Entre otros puede tomar los siguientes valores:

- `"r"`. Abrir un fichero existente para lectura.
- `"w"`. Abrir un fichero existente para escritura.
- `"a"`. Abrir un fichero para escribir al final del mismo o crear el fichero, sino existe, para escribir en él.
- `"r+"`. Abrir un fichero existente para lectura y escritura.
- `"w+"`. Abrir un fichero existente para leer y escribir, pero truncando primero su tamaño a 0 bytes. Si el fichero no existe, se crea para leer y escribir en él.

2.A.2 Función fread

La declaración de la función de librería `fread` es:

```
#include <stdio.h>
size_t fread (char *p, size_t longitud, size_t nelem, FILE *flujo);
```

En esta declaración `size_t` es un tipo de dato predefinido de tipo entero sin signo que se declara en `<stdio.h>` para almacenar el resultado del operador `sizeof`.

La función `fread` lee `nelem` bloques de datos procedentes del fichero apuntado por `flujo` y los copia en el array apuntado por `p`. Cada bloque de datos tiene un tamaño de `longitud` bytes. La operación de lectura finaliza cuando se encuentra el final del fichero, se da una condición de error o se ha leído el total de bloques pedidos. Si la lectura se realiza con éxito, `fread` devuelve el total de bloques leídos. Si el valor devuelto es 0, significa que se ha encontrado el final del fichero.

2.A.3 Función fwrite

La declaración de la función de librería `fwrite` es:

```
#include <stdio.h>
size_t fwrite(const char *p, size_t longitud, size_t nelem,
              FILE *flujo);
```

Esta función escribe en el fichero apuntado por `flujo` `nelem` bloques de datos de tamaño `longitud` procedentes del array apuntado por `p`. Si la escritura se realiza con éxito, `fwrite` devuelve el total de bloques escritos. Si se da una condición de error, el número devuelto por `fwrite` será distinto del número de bloques que se le pasó como parámetro.

2.A.4 Función fclose

La declaración de la función de librería `fclose` es:

```
#include <stdio.h>
int fclose (FILE *flujo);
```

Esta función cierra el fichero asociado con `flujo`. `fclose` también hace que la memoria intermedia de datos asociada a `flujo` sea escrita en el disco, que dicha memoria intermedia sea liberada y que el flujo de datos `flujo` sea cerrado. En caso de éxito devuelve 0 y `EOF` en caso contrario.

2.A.5 Función feof

La declaración de la función de librería `feof` es:

```
#include <stdio.h>
int feof (FILE *flujo);
```

Esta función devuelve un valor distinto de 0 si el indicador de final de fichero está a 1, es decir, si se ha alcanzado el final del fichero.

2.A.6 Función fgets

La declaración de la función de librería `fgets` es:

```
#include <stdio.h>
char *fgets (char *p, int n, FILE *flujo);
```

Esta función lee caracteres en el fichero asociado con `flujo` y los almacena en elementos sucesivos del array apuntado por `p`. La función para de leer cuando almacena `n-1` caracteres o almacena un carácter de nueva línea `'\n'`. En cualquier de los dos casos `fgets` almacena un carácter nulo `'\0'`en el siguiente elemento del array. En caso de éxito la función devuelve un puntero al array. En caso de error o de alcanzar el final del fichero `fgets` devuelve un puntero nulo.

2.A.7 Función fgetc

La declaración de la función de librería `fgetc` es:

```
#include <stdio.h>
int fgetc (FILE *flujo);
```

Esta función lee un carácter en el fichero asociado con `flujo`. En caso de éxito devuelve el carácter leído convertido en un entero sin signo. En caso de error o de alcanzar el final del fichero `fgets` devuelve `EOF`.

2.A.8 Función fputs

La declaración de la función de librería `fputs` es:

```
#include <stdio.h>
int fputs (const char *p, FILE *flujo);
```

Esta función escribe los caracteres de la cadena apuntada por `p` en el fichero asociado con `flujo`. No escribe el carácter nulo de terminación. En caso de éxito, devuelve un valor no negativo; en caso de error devuelve `EOF`.

2.A.9 Función fputc

La declaración de la función de librería `fputs` es:

```
#include <stdio.h>
int fputc(int c, FILE *flujo);
```

Esta función escribe el carácter `c` en el fichero asociado con `flujo`. En caso de éxito, devuelve el carácter `c`; en caso de error devuelve `EOF`.

2.A.10 Función fscanf

La declaración de la función de librería `fscanf` es:

```
#include <stdio.h>
int fscanf(FILE *flujo, const char *formato[, dir, ...]);
```

Esta función lee de uno en uno una serie de campos de entrada procedentes del fichero asociado con `flujo`, los convierte al tipo de datos y da formato de acuerdo con los especificadores de tipo de datos y formato (similares a los utilizados en la función `scanf`) incluidos en la cadena `formato`. Luego almacena las entradas formateadas en las direcciones pasadas como argumentos (una dirección `dir` por cada campo). En caso de éxito devuelve el número de campos de entrada leídos, formateados y almacenados. Si `fscanf` intenta leer el final de un fichero, devuelve `EOF` como valor de retorno. Si no se almacenó ningún campo devuelve el valor 0.

2.A.11 Función fprintf

La declaración de la función de librería `fprintf` es:

```
#include <stdio.h>
int fprintf(FILE *flujo, const char *formato[, argumento, ...]);
```

Esta función genera texto formateado. Acepta una serie de argumentos a los que da formato de acuerdo con los especificadores de formato (similares a los utilizados en la función `printf`) incluidos en la cadena `formato` y escribe la salida formateada en el fichero asociado con `flujo`. Debe existir un especificador de formato para cada argumento. En caso de éxito devuelve el número de caracteres escritos o `EOF` en caso de error.

COMPLEMENTO 2.B
Origen del término proceso demonio

El término *proceso demonio* o *demonio* (daemon) fue acuñado por los programadores del proyecto MAC (Multiple Access Computer) del MIT. Ellos tomaron el nombre del *demonio de Maxwell* un ser imaginario de un famoso experimento pensado por el físico escocés J. M. Maxwell para probar posibles violaciones de la segunda ley de la termodinámica. El demonio de Maxwell trabajaba continuamente sin ser visto ordenando moléculas. Maxwell se inspiró en los demonios de la mitología griega, algunos de los cuales se encargaban de hacer aquellas tareas que los dioses no querían realizar.

Los sistemas UNIX heredaron esta terminología para designar a un tipo especial de proceso que se ejecuta de forma continua en segundo plano y que no puede ser controlado directamente por el usuario ya que no está asociado con una terminal o consola. Al igual que los proceso 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.

Un proceso demonio no hace uso de los dispositivos de entradas y salida estándar para comunicar errores o registrar su funcionamiento, sino que usa archivos del sistema en zonas especiales o se utilizan otros demonios especializados en dicho registro.

CAPÍTULO

3

Administración básica del sistema UNIX

3.1 INTRODUCCIÓN

Una descripción completa de la administración de un sistema UNIX daría, sin duda, para escribir un libro entero, de hecho existen en el mercado una gran variedad de libros centrados en este tema. Este capítulo tiene un objetivo más modesto, en él se describen de forma básica algunas de los principales trabajos asociados con la 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. Se deja para el capítulo 8 la creación de enlaces a ficheros y el montaje de sistemas de ficheros.

En este capítulo en primer lugar se incluyen una serie de consideraciones iniciales tales como la entrada al sistema UNIX y los conceptos de consola virtual e intérprete de comandos (shell). En segundo lugar se describen los comandos de UNIX más comunes. Así se describen los comandos que se utilizan para el manejo de directorios y ficheros, la ayuda de UNIX, la edición de ficheros y los comandos para salir del sistema. En tercer lugar se explica la gestión de usuarios en UNIX. En cuarto lugar se analiza la configuración de los permisos de acceso a un fichero. En quinto lugar, se realizan una serie de consideraciones generales sobre los intérpretes de comandos. Así se describen los diferentes tipos de intérpretes existentes, el uso de caracteres comodines, la redirección de la entrada y de la salida, el encadenamiento de órdenes, la asignación de alias a comandos, los shell scripts, las variables del intérprete de comandos y las variables de entorno. En sexto lugar se explica el control de tareas

El capítulo finaliza con cuatro complementos. En el primer complemento se describen otros comandos existentes en UNIX. En el segundo complemento se incluyen ejemplos adicionales de shell scripts. El tercer complemento está dedicado a los ficheros de arranque de un intérprete de comandos. Finalmente en el cuarto complemento se describe la función de librería `system`.

Los ejemplos que se incluyen en este capítulo (y en los demás capítulos) han sido realizados sobre una distribución de un 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 GPL[1], que es la licencia pública del Proyecto GNU[2]. En el Apéndice A se incluyen las principales consideraciones que se deben tener en cuenta antes de instalar Linux y se enumeran las principales distribuciones de Linux existentes actualmente.

3.2 CONSIDERACIONES INICIALES

3.2.1 Acceso 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.

Es importante recordar que UNIX distingue entre letras mayúsculas y letras minúsculas, así por ejemplo la contraseña `RTF2007A` es distinta de la contraseña `rtf2007a`.

♦ **Ejemplo 3.1:**

En el momento de entrar en el sistema aparecerá la siguiente línea de comandos en la pantalla:

```
ORION login:
```

donde `ORION` es el nombre del ordenador (*hostname*).

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

```
ORION login: darkseid
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 un mensaje de aviso. En ese caso, deberá intentarse de

[1] GPL es el acrónimo derivado del término inglés "General Public License" (Licencia pública general)
[2] GNU es el acrónimo recursivo para "GNU No es Unix".

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

En las distribuciones modernas de UNIX cuando un usuario accede al sistema lo hace sobre una interfaz gráfica del tipo X Windows, la cual permite al usuario comunicar órdenes al sistema mediante el uso del ratón y el teclado. Con esta interfaz el usuario trabaja de forma similar a cómo lo hace en el conocido sistema operativo Windows de Microsoft.

Otra interfaz posible de comunicación con el sistema UNIX, que era la única forma posible de comunicación en las distribuciones clásicas, es la que ofrece los intérpretes de comandos. Un *intérprete de comandos* (shell) es simplemente un programa de utilidad que permite al usuario comunicarse con el sistema. Básicamente lo que hace es leer las *órdenes o comandos* que teclea el usuario en la línea de órdenes, buscar los programas ejecutables asociados a las mismas y ejecutarlos.

El formato general de una orden o comando de UNIX es el siguiente

```
nombre_orden -opciones parámetro1 parámetro2 ... parámetroN
```

Cuando el intérprete de comandos lee una línea de comandos, extrae la primera palabra `nombre_orden`, asume que ésta es el *nombre de un programa ejecutable*, lo busca y lo ejecuta. El intérprete de comandos suspende su ejecución hasta que el programa termina, tras lo cual intenta leer la siguiente línea de órdenes.

Las *opciones* modifican el comportamiento por defecto de la orden y se suelen indicar justo después del nombre de la orden y con el prefijo `-`. Si existen varias opciones se pueden agrupar. Por ejemplo, las orden `ls -a -l -s` es equivalente a `ls -als`.

Los *parámetros o argumentos adicionales* aportan información adicional que será usada por la orden. Los más habituales son nombres de ficheros y nombres de directorios, pero no tiene por qué ser siempre así. La mayoría de las órdenes aceptan múltiples opciones y argumentos.

♦ **Ejemplo 3.2:**

Supóngase que el usuario `darkseid` acaba de entrar en el sistema, entonces en la pantalla aparece el marcador (prompt) del intérprete de comandos

```
$
```

Un ejemplo de orden sería:

```
$ cp fich1 fich2
```

Esta orden copia el contenido del fichero `fich1` en el fichero `fich2` De acuerdo con la estructura general de una orden en UNIX, se observa que el nombre de la orden es `cp`, que no se han especificado opciones y que tiene dos parámetros o argumentos adicionales `fich1` y `fich2`. Realmente el número total de parámetros es tres ya que el nombre de la orden se considera como un parámetro más.

♦

Puede ser bastante útil tener presente que si se desea borrar todo el contenido de la línea de órdenes una forma de hacerlo es pulsando la combinación de teclas [`control`+`u`]. Si únicamente se desea borrar una palabra de la línea de órdenes se puede pulsar [`control`+`w`]. Asimismo la combinación de teclas [`control`+`c`] interrumpe la orden actualmente en ejecución, mientras que [`control`+`z`] la suspende.

3.3 COMANDOS DE UNIX MÁS COMUNES

3.3.1 Manejo de directorios y ficheros

3.3.1.1 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` que tiene el mismo nombre que el usuario.

Las órdenes que se teclean en el intérprete de comandos toman como referencia el directorio de trabajo actual. Para cambiar el directorio de trabajo y moverse en la estructura de directorios se utiliza la orden `cd`**.** La sintaxis más habitual 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.

Para conocer la ruta absoluta del directorio de trabajo actual se puede usar el comando `pwd`.

♦ **Ejemplo 3.3:**

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

```
$ cd CANCIONES
```

o la orden:

```
$ cd /home/darkseid/CANCIONES
```

En el primer caso se está indicando la ruta relativa mientras que en el segundo se está indicando la ruta absoluta. Para conocer la ruta absoluta del directorio de trabajo actual se puede teclear la orden

```
$ pwd
```

En este ejemplo, en pantalla aparecería como respuesta

```
/home/darkseid/CANCIONES
```

Para volver al directorio anterior, se puede teclear la orden:

```
$ cd ..
```

o la orden

```
$ cd /home/darkseid
```

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

```
$ cd
```

♦

3.3.1.2 Obtener información de un directorio o de un fichero

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

```
ls [-opciones][fichero(s)]
```

Los opciones más frecuentes son:

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

Si se incluyen como parámetros de la orden los nombres de determinados ficheros entonces la orden `ls` solamente mostrará información sobre dichos ficheros y no sobre todos los ficheros contenidos en el directorio de trabajo actual.

♦ **Ejemplo 3.4:**

Supóngase que un usuario teclea la orden

```
$ ls
```

y en pantalla aparece la siguiente respuesta:

```
Fotos
películas
CANCIONES
```

Esta respuesta indica que el directorio de trabajo actual tiene tres entradas `Fotos`, `películas` y `CANCIONES`. Hoy en día como los monitores son de color es posible saber si una entrada se corresponde con un fichero (ordinario o ejecutable) o con un directorio, ya que se utilizan colores distintos para cada una. Hace años cuando sólo había monitores monocolor la información contenida en esta respuesta del comando era claramente insuficiente, ya que no se podía saber si las entradas eran ficheros o directorios. Para obtener esta información se tenía que teclear la siguiente orden:

```
$ ls -F
```

Supóngase que en pantalla aparece la siguiente respuesta:

```
Fotos/
películas/
CANCIONES/
```

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.

Supóngase ahora que en el directorio de trabajo actual, entre otros ficheros, existe uno llamada `prueba` del que se desea conocer cual es su número de nodo-i. Para ello simplemente habría que escribir la siguiente orden:

```
$ ls -i prueba
```

Supóngase que en pantalla aparece la siguiente respuesta:

```
12500 prueba
```

En dicho caso `12500` es el número de nodo-i asignado al fichero `prueba`.

♦

3.3.1.3 Crear directorios nuevos

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

```
mkdir dir1 dir2 ...dirN
```

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

♦ **Ejemplo 3.5:**

Supóngase que dentro del directorio de trabajo actual existen los directorios `Fotos`, `peliculas` y `CANCIONES`. La orden

```
$ mkdir correo
```

crearía dentro del directorio de trabajo actual el subdirectorio `correo`. Esto se puede comprobar escribiendo la siguiente orden

```
$ ls -F
```

En pantalla aparecería la siguiente respuesta:

```
Fotos/
películas/
CANCIONES/
correo/
```

♦

3.3.1.4 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

```
$ cp /etc/host.conf .
```

copia en el directorio de trabajo actual '`.`' el fichero `host.conf` (que está dentro del directorio `/etc/`).

♦

3.3.1.5 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 cambiar el nombre de un fichero.

♦ **Ejemplo 3.7:**

Supóngase que en el directorio de trabajo actual `/home/darkseid` se encuentran entre otros los ficheros `prueba.txt` y `lista.dat`. En ese caso la orden

```
$ mv lista ..
```

mueve el fichero `lista.dat` al directorio `/home`.

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

```
$ mv prueba.txt prueba2.txt
```

♦

3.3.1.6 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

```
$ rm prueba2.txt
```

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

♦

3.3.1.7 Acceder al contenido de los ficheros

Para ver el contenido de un fichero se pueden usar las órdenes `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 la tecla `[espacio]` para avanzar a la página siguiente y la tecla `[b]` para volver a la página anterior. Si se pulsa la tecla `[q]` finalizará la ejecución de `more`.

♦ **Ejemplo 3.9:**

La orden

```
$ cat /etc/host.conf
```

mostraría por pantalla el contenido del fichero `host.conf` situado dentro del directorio `/etc`. El problema es que se visualizaría tan rápido que no sería posible leerlo. Para visualizarlo pantalla a pantalla se debe teclear la orden:

```
$ more /etc/host.conf
```

♦

3.3.2 La ayuda de UNIX

3.3.2.1 Páginas del manual de ayuda de UNIX

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.

En UNIX puede suceder que con un mismo nombre se esté designando a un comando y a un recurso del sistema. En este caso el manual de ayuda de UNIX dispone de una página para cada caso. A las distintas páginas del manual de ayuda asociadas a un mismo nombre se les denomina *secciones*. Para especificar una determinada sección el comando `man` se debe invocar de la siguiente forma:

```
man sección nombre
```

donde `sección` es un número entero positivo que designa la sección del manual asociada a `nombre` a la que se desea acceder.

♦ **Ejemplo 3.10:**

Como se explicará en el Capítulo 8 en UNIX se utiliza la palabra `mount` para designar tanto a una orden como a una llamada al sistema que permite montar un sistema de ficheros. La orden

```
$ man mount
```

mostrará por pantalla la página del manual de ayuda asociada al comando `mount`. Mientras que la orden

```
$ man 2 mount
```

mostrará por pantalla la segunda sección del manual de ayuda asociada al nombre `mount` que describe a la llamada al sistema que tiene ese mismo nombre.

♦

Todas las páginas del manual de ayuda de UNIX tienen la siguiente estructura: nombre de la orden o función (nº sección), breve descripción de la orden, descripción detallada de las órdenes y sus opciones, ficheros que necesita la orden para funcionar, referencia a documentos afines (incluyendo otras páginas del manual), comentarios de errores detectados y/o detalles sorprendentes y autor del programa (nombre y dirección de contacto).

La visualización en pantalla de una página del manual de ayuda de UNIX se va realizando por partes (conjunto de líneas) comenzando por la primera. Si se pulsa la tecla `[espacio]` se muestra la siguiente parte. Si se pulsa la tecla `[b]` se vuelve a la parte anterior. También es posible ir avanzando línea a línea para ello se debe ir pulsando la tecla `[intro]`. Para finalizar la ejecución del comando `man` se debe pulsar la tecla `[q]`.

Es importante recordar que no todos los comandos disponen de página de ayuda. En dicho caso cuando se invoca el comando `man` con el nombre de un comando sin página de ayuda aparecerá un mensaje en pantalla avisando de esta circunstancia.

3.3.2.2 Manuales info

Los *manuales info* son libros o manuales que documentan algunos programas. Para consultarlos en pantalla se puede emplear el comando `info` seguido del nombre del manual que se desea consultar. Si se emplea el comando `info` sólo, se accede a un índice de los manuales disponibles.

3.3.3 Edición de ficheros

Para editar un fichero se puede invocar desde la línea de comandos a alguno de los editores disponibles en las diferentes distribuciones de UNIX, por ejemplo, el editor `vi`. Para conocer el funcionamiento de este o de otro editor se recomienda consultar su página del manual de ayuda de UNIX.

Las distribuciones modernas de UNIX suelen incluir editores de texto a los que se puede acceder y trabajar a través de la interfaz X Windows. Obviamente el uso de estos editores es mucho más sencillo y cómodo.

3.3.4 Salir del sistema

Para salir del sistema UNIX de forma segura se recomienda usar el comando `shutdown`. En la invocación de este comando se puede especificar que tras salir del sistema la computadora se apague o se reinicie.

Si se desea salir del sistema UNIX más rápidamente y apagar la computadora se pueden teclear los comandos `halt` o `poweroff`. Si se desea reiniciar la computadora se puede usar el comando `reboot`.

Si únicamente se desea cerrar el intérprete de órdenes se puede teclear el comando `exit` o pulsar `[control+d]`. Antiguamente cuando el intérprete de comandos era la única interfaz de que disponía el usuario para comunicarse con el sistema, la invocación de este comando provocaba no sólo el cierre del intérprete de comandos sino también la salida del sistema UNIX.

3.4 GESTIÓN 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. Es usual encontrar en este campo simplemente el carácter ‘`x`‘. Esto significa que la contraseña se encuentra almacenada dentro del archivo `/etc/shadow` que únicamente puede leer el superusuario. De esta forma obtener la contraseña de un usuario es mucho más difícil.

- *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 3.11:**

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

```
darkseid:Ay7W352a52eDF:201:200:Pablo Marcos:/home/darkseid:/bin/bash
```

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

- `darkseid` es el nombre de usuario.
- `Ay7W352a52eDF` es la clave encriptada.
- `201` es el *uid*.
- `200` es el *gid*.
- `Pablo Marcos` es el nombre completo del usuario.
- `/home/darkseid` es el directorio de trabajo inicial.
- `/bin/bash` es el intérprete de comandos inicial.

♦

Si se abre el fichero `/etc/passwd` y se lee su contenido se puede observar que existen cuentas asociadas a usuarios “extraños”. Pues bien estas cuentas de usuario son creadas por el sistema cuando se instalan ciertos programas y son necesarias para la ejecución de los mismos por lo que no conviene manipularlas.

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 qué 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 3.12:**

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

```
darkseid:*Ay7W352a52eDF:201:200:Pablo Marcos:/home/darkseid:/bin/bash
```

se evitará que `darkseid` 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 qué 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, *gid* y el login de otros usuarios miembros al grupo.

La clave de acceso a un grupo no se suele utilizar, por eso en dicho campo aparece el carácter '`*`'. En dicho campo también puede aparecer el carácter '`x`' que significa que la contraseña se encuentra almacenada dentro del archivo `/etc/shadow`.

♦ **Ejemplo 3.13:**

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

```
root:*:0:

users:*:100:PROFESOR,darkseid

invitados:*:200:

otros:x: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 `darkseid`.

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`. Su contraseña de acceso se encuentra almacenada en el fichero `/etc/shadow`. 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 manipular el fichero `/etc/group`. Si se desea añadir un usuario a un cierto grupo basta con añadir el login del usuario al final de la línea correspondiente a dicho grupo en el fichero `/etc/group`.

Si se desea añadir un nuevo grupo se debe añadir una nueva línea al 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`.

Si se abre el fichero `/etc/group` y se lee su contenido se puede observar que existen líneas asociadas a grupos “extraños”. El significado de estos grupos es análogo al descrito para las cuentas “extrañas” que aparecen en el fichero `/etc/passwd`.

3.5 CONFIGURACIÓN 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:

$$S_9S_8S_7S_6S_5S_4S_3\ S_2S_1S_0$$

- El carácter s_9, indica el tipo de fichero: ordinario (`-`), directorio (`d`), especial modo carácter (`c`), especial modo bloque (`b`), FIFO o tubería (`p`), enlace simbólico (`l`) 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_ISGID` 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 s_2, indica si se encuentra habilitado el permiso de lectura (`r`) para el resto de usuarios, en caso negativo tomará el valor (`-`).

- El carácter s_1, indica si se encuentra habilitado el permiso de escritura (`w`) para el resto de usuarios, en caso negativo tomará el valor (`-`).

- El carácter s_0, 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 3.14:**

Supóngase que en el directorio de trabajo actual existe únicamente el fichero `notas` y que la orden:

```
$ ls -l
```

muestra en pantalla la siguiente línea:

```
-rw-r--r--  1 darkseid  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, `darkseid` 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, `darkseid`. Luego `darkseid` tiene permisos de lectura y escritura para el fichero `notas`. Además como s_6=`-`, significa que `darkseid` 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 3.15:**

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

a) `- 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.

b) `- 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.

c) `- 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 qué 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 qué tipo de permiso se hace referencia lectura (r), escritura (w) o ejecución (x). También se pueden 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 [$M_3M_2M_1M_0$] es el modo del fichero expresado en octal

♦ **Ejemplo 3.16:**

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

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

3.6.1 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`[3]. 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*[4]. *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 otro es cuestión de gustos. Algunas personas prefieren la sintaxis del intérprete *Bourne* con las características avanzadas que proporciona el intérprete *Bash* y otras prefieren el más estructurado intérprete C. En lo que respecta a los comandos usuales (`cp`, `ls`, ...) 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.

---

[3] *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.

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

3.6.2 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 3.17:**

Supóngase que en el directorio de trabajo actual existen únicamente los ficheros `lista` y `fotos`. Para ver un listado de los ficheros que poseen alguna letra 'o' en su nombre, se puede usar la orden:

```
$ ls *o*
```

En pantalla aparecería la siguiente respuesta:

```
fotos
```

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

```
$ ls *
```

mostraría la siguiente respuesta en la pantalla

```
lista fotos
```

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.

3.6.3 Redirección de entrada/salida

UNIX proporciona un mecanismo sencillo para cambiar la entrada y la salida estándar. Este mecanismo se denomina *redirección de entrada/salida*.

Si el último parámetro de una orden es un nombre de fichero precedido por el carácter '>' (mayor que) entonces la salida estándar de esa orden se redirige hacia ese fichero en vez de aparecer en pantalla. Si el fichero no existe, se crea. Si ya existe, se eliminará su contenido y se reemplazará por la salida de la orden. Para evitar este último caso, se puede redireccionar mediante los caracteres >>, de forma que la salida de la orden se concatena con el contenido anterior del fichero.

Para redirigir la entrada estándar se usa el carácter '<' (menor que) seguido del nombre del fichero de entrada.

♦ **Ejemplo 3.18:**

La orden

```
$ ls -l > listado
```

almacena en el fichero `listado` el resultado de la orden `ls -l`. Si el fichero no existe se crea. Si el fichero ya existe, se eliminará su contenido y se reemplazará por la salida de la orden. Para evitar esto último se debe teclear la orden

```
$ ls -l >> listado
```

Ahora el resultado de la orden se concatena con el contenido anterior del fichero.

La orden

```
$ cat capitulo1 capitulo2 capitulo3 > libro
```

concatena el contenido de los ficheros `capitulo1`, `capitulo2` y `capitulo3` y lo guarda en el fichero `libro`.

La orden

```
$ cat < /etc/host.conf
```

mostraría por pantalla el contenido de `host.conf`. En este caso se obtendría el mismo resultado si se teclease la orden

```
$ cat /etc/host.conf
```

ya que la orden `cat` muestra por pantalla el contenido de los ficheros que se le pasan como argumentos.

La orden

```
$ sort < borrador > definitivo
```

ordena (`sort`) por orden alfabético las líneas del fichero `borrador` y escribe el resultado en `definitivo`.

♦

3.6.4 Encadenamiento de órdenes

Se pueden escribir varias órdenes en una misma línea escribiendo un punto y coma entre ellas. De esta forma, primero se ejecutara la orden que está escrita en primer lugar, luego la orden que está escrita en segundo lugar y así sucesivamente.

También es posible conseguir que la salida de una orden se convierta en la entrada de otra orden. Para ello se debe emplear una tubería cuyo símbolo es '`|`'. Las tuberías son un mecanismo de comunicación entre procesos que será estudiado en detalle en el Capítulo 7.

♦ **Ejemplo 3.19:**

La orden `ls` muestra por la salida estándar el contenido de un directorio. Por otra parte, la orden `sort -r` lee líneas de la entrada estándar y las muestra por la salida estándar en orden alfabético inverso. Con la orden

```
$ ls | sort -r
```

se está haciendo uso de una tubería '`|`' para conseguir que la salida de `ls` se redirija hacia la entrada de `sort`. En consecuencia el resultado de esta orden será presentar en pantalla un listado del directorio de trabajo actual en orden alfabético inverso.

♦

3.6.5 Asignación de alias a comandos

En ciertas ocasiones se suelen utilizar comandos que son difíciles de recordar o que son demasiado extensos. En UNIX existe la posibilidad de dar un *nombre alternativo o alias* a un comando con el fin de que cada vez que se quiera ejecutar, sólo se use el nombre alternativo. Obviamente se debe procurar que el alias que se utilice sea corto y fácil de recordar.

Para definir un alias sólo se necesita usar el comando `alias`, su sintaxis es:

```
alias nombre='definición'
```

donde `nombre` es el nuevo nombre o alias que se va asignar al comando y `definición` es el comando o los comandos que se desean que se ejecuten que se ejecuten cuando se teclee el alias. Obsérvese que `definición` debe escribirse entre `' '`. Si se desean conocer todos los alias existentes en el intérprete se debe escribir el comando `alias` sin argumentos.

Para borrar un alias anteriormente definido se debe usar el comando `unalias`, su sintaxis es:

```
unalias nombre_alias
```

donde `nombre_alias` es el nombre del alias que se desea borrar.

♦ **Ejemplo 3.20:**

La orden

```
$ alias comprobar='pwd; ls -F'
```

crea un alias denominado `comprobar` para las órdenes `pwd` y `ls -F`. De esta forma si se teclea el alias `comprobar` se ejecutarán en su lugar las órdenes `pwd` y `ls -F`.

Para borrar el alias anteriormente definido se debe teclear la orden:

```
$ unalias comprobar
```

♦

Se debe tener presente que un alias únicamente existirá mientras continúe abierta la sesión de trabajo del usuario en el intérprete de comandos (salvo, claro está, que sea borrado con el comando `unalias`), es decir, si se cierra el intérprete de comandos y posteriormente se vuelve abrir el alias definido ya no existirá. Si se desea que la definición de un determinado alias sea persistente, es decir, que ya esté definido cada vez que se abra el intérprete de comandos, entonces se debe incluir dicha definición en los ficheros de arranque del intérprete (ver Complemento 3C).

3.6.6 Shell scripts

En un intérprete de comandos aparte de ejecutar órdenes también es posible definir variables y funciones, así como usar sentencias del tipo `if`, `while`, `for`, etc. En consecuencia, el intérprete de comandos dispone de un completo lenguaje de programación.

Los *shell scripts* son ficheros de texto que contienen programas escritos en el lenguaje del intérprete de comandos. Para poder ejecutar un shell script es necesario que su fichero de texto asociado tenga activados, al menos, los permisos de lectura y ejecución.

La principal ventaja que presenta un shell script frente a un programa compilado es la portabilidad. Un shell script puede ser ejecutado en una máquina UNIX u otra sin necesidad de retocar nada, salvo que se utilicen llamadas a órdenes muy concretas específicas de una determinada distribución de UNIX. Mientras que un programa desarrollado en C, Pascal, etc. debe ser recompilado, pues el código se genera en función del microprocesador de cada máquina. Otra ventaja es la facilidad de lectura e interpretación.

El principal inconveniente que presenta un shell script respecto a un programa compilado es la lentitud de ejecución. Otro inconveniente es que el código resulta visible a cualquier usuario que lo pueda ejecutar.

Los shell scripts suelen encabezarse con comentarios que indican el nombre de archivo y lo que hace el shell script. Además se colocan comentarios de documentación en diferentes partes del shell script para mejorar la comprensión y facilitar el mantenimiento. Los comentarios se insertan anteponiendo el carácter "`#`" al comentario, que se extenderá hasta el final de la línea.

Un caso especial es el uso de "`#`" en la primera línea, seguido del carácter admiración "`!`" y la ruta de acceso del nuevo intérprete de comandos con que se ejecutará el script. Debe tenerse en cuenta que cuando se invoca a un shell script desde la línea de órdenes de un intérprete de comandos, este intérprete invoca usualmente a otro intérprete de comandos, al cual se le denomina *subintérprete* (subshell).

A la hora de programar un shell script conviene saber que muchos comandos devuelven un valor después de ejecutarse, el cual indicará si la ejecución ha sido correcta o si se ha producido algún fallo y que tipo de fallo se ha producido. Para conocer si un comando devuelve o no un valor y qué es lo que devuelve en cada caso se debe consultar el manual de UNIX. Por lo general si el comando se ejecuta correctamente devolverá el valor 0 y en caso de fallo devolverá otro número, positivo o negativo.

♦ **Ejemplo 3.21:**

Supóngase el shell script llamado `primer_script` cuyo código es el siguiente

```
#!/bin/bash
# Este programa concatena el contenido de los ficheros
# datos1, datos2 y datos3 en el fichero temp
cat datos1 datos2 datos3 > temp
# Ordena alfabétioamente todas las líneas de temp
# y las almacena en el fichero resultado.dat
sort < temp > resultado.dat
# Visualiza el contenido de resultado.dat
more resultado.dat
# Finalmente cuenta el número de líneas de resultado.dat
wc -l resultado.dat
```

Este es un shell script sencillo que únicamente contiene comentarios y órdenes del intérprete. No se utilizan elementos tales como variables, funciones y sentencias de programación del tipo `if`,

`for`, `while`, etc. Ejemplos de shell scripts que utilizan estos elementos pueden encontrarse en el Complemento 3.B.

Obsérvese que la primera línea del shell script indica la ruta de acceso (`/bin/bash`) del nuevo intérprete de comandos (`bash`) con que se ejecutará el shell script

Es importante recordar que el fichero de texto donde se escriba este shell script debe tener permiso de ejecución para el usuario de lo contrario no se ejecutará. Para saber si dispone de permiso de ejecución se debe consultar la máscara de modo simbólica tecleando la orden:

```
$ ls -l primer_script
```

Si el fichero no dispone de permiso de ejecución se le puede otorgar escribiendo la orden

```
$ chmod u+x primer_script
```

Finalmente la invocación de este shell script se realizaría con la orden:

```
$ primer_script
```

♦

3.6.7 Funcionamiento de un intérprete de comandos

Un intérprete de comandos es un fichero ejecutable. El proceso que se crea asociado a la ejecución de dicho fichero en primer lugar lee y ejecuta las órdenes establecidas en los diferentes ficheros de arranque del intérprete para configurar sus variables internas, a continuación muestra el marcador en pantalla y se queda a la espera de recibir órdenes del usuario. De forma general, las órdenes que se pueden ejecutar en un intérprete de comandos pueden ser de dos tipos:

- *Órdenes internas* (*builtin commands*). Son aquellas órdenes cuyo código de ejecución se encuentra incluido dentro del propio código del intérprete. Así la ejecución de una orden interna no supone la creación de un nuevo proceso. Ejemplos de órdenes internas son `cd` y `pwd`. En el manual de ayuda de UNIX en la entrada asociada al intérprete de comandos que se esté utilizando se puede encontrar información sobre todas sus órdenes internas.
- *Órdenes externas*. Son aquellas órdenes que para poder ser ejecutadas por el intérprete requieren de la búsqueda y ejecución del fichero ejecutable asociado a cada orden. Típicamente son programas ejecutables o shell scripts incluidos en la distribución del sistema o creados por el usuario. La ejecución de una orden externa supone la creación de al menos un nuevo proceso con la llamada al sistema `fork` (ver sección 5.2) y la invocación del programa ejecutable con la llamada al sistema `exec` (ver sección 5.7). Ejemplos de órdenes externas son `ls` y `mkdir`.

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. En caso afirmativo la ejecuta. En caso contrario considera que es una orden externa por lo que debe buscar su programa ejecutable asociado. Si lo encuentra lo ejecuta. En el caso de que no se pueda encontrarlo mostrará un mensaje de aviso por la pantalla.

3.6.8 Variables del intérprete de comandos y variables de entorno

Cuando se ejecuta un intérprete de comandos se crean, con los valores iniciales establecidos en diferentes ficheros de arranque, un conjunto de variables denominadas *variables del intérprete de comandos*. Estas variables son locales al intérprete y pueden ser utilizadas por todas sus órdenes internas. Por ejemplo, la variable `SECONDS` contiene el número de segundos transcurridos desde que se arrancó la ejecución del intérprete de comandos. Para ver todas las variables disponibles se puede teclear el comando `set`. En el manual de ayuda de UNIX en la entrada asociada al intérprete de comandos que se esté utilizando se explica el significado de cada una de estas variables.

Si únicamente se desea visualizar el valor de una determinada variable se debe usar el comando `echo` con la siguiente sintaxis:

```
echo $variable
```

donde `variable` es el nombre de la variable cuyo valor se desea visualizar. Nótese que está precedido por el símbolo `$` (no debe confundirse con el marcador del intérprete), ésta es la forma de acceder al valor contenido en la variable.

Por otra parte un usuario puede cambiar el valor de algunas de las variables del intérprete y definir nuevas variables. La forma de cambiar el valor de una variable ya existente o de definir una variable nueva es escribir su nombre, el signo igual y su valor.

♦ **Ejemplo 3.22:**

La orden

```
$ PRUEBA=0
```

Crea una variable del intérprete de comandos denominada PRUEBA y le asigna el valor `0`.

Por su parte la orden

```
$ echo $PRUEBA
```

Muestra por pantalla el valor de dicha variable

```
0
```

Si se teclea la orden

```
$ set
```

se mostrarán en la pantalla todas las variables del intérprete, incluida la variable `PRUEBA`.

♦

Se denominan *variables de entorno* a aquellas variables del intérprete de comandos que proceden del *entorno* del proceso (ver sección 5.7) asociado al intérprete. Por ejemplo, la variable `UID` es una variable del intérprete de comandos que es también una variable de entorno. Esta variable es de sólo lectura y especifica el identificador de usuario real (ver sección 4.3) del usuario. Para visualizar un listado de las variables de entorno del intérprete se puede usar el comando `env`.

Los cambios que se hacen en una variable de entorno no se pasan al entorno del proceso hasta que el usuario no exporta explícitamente dicha variable al entorno. Para ello debe usar el comando `export` seguido del nombre de la variable. La importancia de exportar variables al entorno radica en que cuando se ejecuta una orden externa desde el intérprete se crea un nuevo proceso asociado a la ejecución de dicha orden que recibe una copia del entorno del proceso asociado al intérprete. En conclusión, las órdenes externas tienen acceso únicamente a las variables de entorno almacenadas en dicha copia del entorno pero no a las variables locales del intérprete que no hayan sido exportadas al entorno.

Algunas variables de entorno están configuradas por defecto en los ficheros de arranque del sistema o del intérprete para ser exportadas automáticamente. Un usuario también puede añadir esta propiedad a una variable usando el comando `set` con la opción `-a` seguido del nombre de la variable.

También es posible exportar al entorno una variable del intérprete que no fuese de origen una variable de entorno.

♦ **Ejemplo 3.23:**

Considérese el shell script `ver` cuyo código es:

```
#!/bin/bash
echo $DATO
```

La función de este shell script es mostrar en la pantalla el valor de la variable `DATO`. Supóngase que se define la siguiente variable del intérprete

```
$ DATO=0
```

Si a continuación se lanza el shell script

```
$ ver
```

En pantalla no aparece ningún valor para `DATO` puesto que dicha variable no ha sido exportada al entorno del intérprete y el shell script no tiene acceso a ella. Para exportarla se debe teclear la siguiente orden:

```
$ export DATO
```

Ahora si se invoca al shell script

```
$ ver
```

en pantalla aparece

```
0
```

Es decir, ahora el shell script si que tiene acceso a la variable.

La definición y exportación de la variable DATO podría haberse realizado alternativamente usando una única orden:

```
$ export DATO=0
```

♦

También es posible borrar variables previamente definidas para ello se puede usar el comando `unset`.

Los cambios, las definiciones y las exportaciones de variables del intérprete desaparecen al cerrar el intérprete de comandos. Si se desea que las acciones realizadas sobre variables sean persistentes, entonces se deben incluir en los ficheros de arranque del intérprete de comandos.

3.6.9 La variable de entorno `PATH`

La variable `PATH` es una variable de entorno que especifica las rutas de acceso a los directorios donde el intérprete debe buscar el fichero ejecutable asociado a una determinada orden externa. Esta variable no es usada, sin embargo, en la localización de los ficheros ordinarios.

Al entrar en el sistema, la variable `PATH` al igual que el resto de variables de entorno es inicializada con un valor por defecto que se encuentra definido en los ficheros de arranque del sistema o del intérprete.

La variable `PATH` contiene una cadena de caracteres con el siguiente formato:

```
ruta1:ruta2:...:rutaN
```

donde `ruta1`, `ruta2`,..., `rutaN` son rutas de acceso a directorios.

Para añadir una nueva ruta `ruta_nueva` al principio de la cadena contenida en `PATH` se debe usar la siguiente orden

```
PATH=ruta_nueva:$PATH
```

Si se desea añadir una nueva ruta `ruta_nueva` al final de la cadena contenida en `PATH`, entonces la orden que hay que usar es:

```
PATH=$PATH:ruta_nueva
```

♦ **Ejemplo 3.24:**

La orden

```
$ echo $PATH
```

mostraría en la pantalla el contenido de la variable `PATH`. Supóngase que el contenido es el siguiente

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

Se observa que la cadena contenida en `PATH` contiene tres rutas. Así cuando se teclea una orden externa, el intérprete buscará el fichero ejecutable asociada a dicha orden primero en `/bin`, luego en `/usr/bin` y finalmente en `/usr/local/bin`.

Supóngase que el directorio de trabajo actual es `/home/darkseid` y que en él existe un programa ejecutable llamado `asd`. Si se invoca desde el intérprete

```
$ asd
```

aparecería un mensaje en pantalla avisando de que el intérprete no ha sido capaz de encontrar el fichero `asd`. Esto es debido a que el directorio de trabajo actual no está incluido en la variable `PATH`. Para incluirlo una forma de hacerlo es escribiendo la orden

```
$ PATH=$PATH:/home/darkseid
```

o alternativamente la orden

```
$ PATH=$PATH:.
```

Ya que "`.`" hace referencia al directorio de trabajo actual.

Se puede comprobar que el directorio de trabajo actual ha sido añadido a `PATH` escribiendo la orden

```
$ echo $PATH
```

En pantalla aparecería

```
/bin:/usr/bin:/usr/local/bin:/home/darkseid
```

o

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

♦

3.7 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.

3.7.1 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 3.25:**

Supóngase que la ejecución de la orden

```
$ 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 primera 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 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.

3.7.2 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 ejecuta 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 `[control+c]`, el proceso muere por lo que 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 esta 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 3.26:**

La orden

```
$ 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 [`control+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:

```
$ 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 [`control`+`c`].

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

```
$ 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
```

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:

```
$ 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 3.27:**

La orden

```
$ 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 [control+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

```
$ 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.

♦

3.7.3 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 3.28:**

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

```
$ kill %1
```

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

```
$ 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* del proceso es 324, así que la orden

```
$ kill 324
```

es equivalente a

```
$ kill %1
```

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

♦

COMPLEMENTO 3.A
Otros comandos de UNIX

3.A.1 Información del sistema

Algunos comandos útiles para obtener información del sistema son:

- `date`. Muestra la fecha y la hora actuales.
- `cal`. Muestra el calendario del mes actual.
- `uptime`. Muestra el tiempo que lleva encendida la máquina.
- `w`. Muestra los usuarios conectados a la máquina.
- `whoami`. Muestra el nombre de mi usuario.
- `finger usuario`. Muestra información sobre `usuario`.
- `uname -a`. Muestra información sobre el núcleo.
- `cat /proc/cpuinfo`. Muestra información sobre la CPU.
- `cat /proc/meminfo`. Muestra información sobre la memoria.
- `df`. Muestra el espacio libre en los discos.
- `du`. Muestra el espacio usado por los directorios
- `free`. Muestra información sobre el uso de la memoria principal y el espacio de intercambio.
- `whereis app`. Localiza el fichero ejecutable, el fichero fuente y la página de manual de `app`
- `which app`. Localiza el comando `app`

3.A.2 Búsqueda de ficheros y texto

El comando `find` busca un fichero en un directorio y sus subdirectorios. Si no se especifica uno en particular, busca desde el directorio actual. Su sintaxis básica es:

```
find archivo
```

El comando `grep` busca una cadena de texto dentro de uno o varios ficheros. Su sintaxis básica es

```
grep patrón fichero1 fichero2    ficheroN
```

3.A.3 Compresión de ficheros

El comando `tar` empaqueta archivos y directorios en un solo archivo. Algunas de sus formas de invocación más habituales son:

- `tar cf fichero.tar f1 f2 fN`. Empaqueta los ficheros `f1`, `f2`,..,`fN` en el fichero `fichero.tar`.
- `tar xf fichero.tar`. Extrae el contenido de `fichero.tar`.

El comando `tar` no realiza compresión de datos, por lo que no reduce el tamaño de los archivos. Sin embargo se puede combinar el uso de `tar` con el algún programa compresor como `gzip` o `bzip2`. Así por ejemplo:

- `tar czf file.tar.gz f1 f2 fN`. Empaqueta y comprime con el programa `gzip` los ficheros `f1`, `f2`,..,`fN` en el fichero `file.tar.gz`.
- `tar czf file.tar.bz2 f1 f2 fN`. Empaqueta y comprime con el programa `bzip2` los ficheros `f1`, `f2`,..,`fN` en el fichero `file.tar.gz`.
- `tar xzf file.tar.gz`. Extrae y descomprime usando el programa `gzip` el contenido de `file.tar.gz`.
- `tar xzf file.tar.bz2`. Extrae y descomprime usando el programa `bzip2` el contenido de `file.tar.bz2`.

Obviamente pueden usarse los programas compresores independientemente del comando `tar`. Por ejemplo, algunas formas de invocar a `gzip` son:

- `gzip fichero`. Comprime `fichero` y lo renombra como `fichero.gz`.
- `gzip -d fichero.gz`. Descomprime `fichero.gz` a `fichero`.

3.A.4 Instalación de software

Si un determinado programa tiene una licencia de software libre, significa que de alguna manera es posible llegar a obtener el código fuente de ese programa.

Normalmente el código fuente se encuentra comprimido en el formato `.tar.gz`. Una vez que se hayan descomprimido los archivos, se debe ejecutar la acción que corresponda según el lenguaje en el que esté desarrollada la aplicación.

La mayor parte del software libre está desarrollado en lenguaje C y pensado para ser compilada con el compilador `gcc`. El software que sigue los directrices de empaquetamiento de GNU, dispone de un comando `configure` que detecta una gran

variedad de datos acerca del sistema utilizado (el procesador, el sistema operativo, el compilador C, las bibliotecas necesarias para compilar y todos los recursos necesarios para compilar el código fuente en cuestión). En el caso en que falte un determinado recurso (bibliotecas, programas, etc) avisará al usuario de ello.

Una vez que todos los recursos han sido detectados correctamente, es necesario ejecutar el comando `make`, para que compile el código fuente. A continuación se debe ejecutar el comando `make install` que instala el programa en el sistema, dejándolo listo para usar.

Para el caso de los programas que no usen los comandos `configure` y `make` será necesario leer la documentación que acompañe al código fuente para saber cómo realizar la compilación.

Si bien todo programa que sea software libre da la posibilidad de ser compilado por el usuario, esto requiere mucho tiempo. Por ello muchas veces es preferible utilizar el código binario (programa ejecutable) que ya ha sido compilado por otras personas, para la plataforma que se esté utilizando.

La gran mayoría de los programas, se distribuyen también en forma binaria compilada por su creador. Normalmente este código binario se encontrará en formato `.tar.gz`, al igual que el código fuente. O incluso en archivos ejecutables que pueden instalarse directamente.

Por lo general basta con descomprimir el archivo y luego agregar el directorio correspondiente a la variable `PATH`, o bien ejecutarlo directamente desde el directorio.

Tanto la instalación desde el código fuente como la instalación a partir del código binario permiten que un usuario instale una aplicación en su directorio de trabajo sin tener que pedirle permiso al administrador del sistema.

Cuando la instalación es realizada por el administrador del sistema, es recomendable colocar los programas en la ruta `/usr/local` o bien `/opt` (estas rutas pueden variar según la distribución de UNIX o Linux utilizada). De forma que todos los usuarios del sistema puedan acceder a estos programas.

Otra forma bastante habitual de presentación de un software es en forma de paquete. Se denomina *paquete* al conjunto formado por el código binario de una aplicación, los scripts necesarios para configurar, instalar y desinstalar esta aplicación,

los datos acerca de otros programas y bibliotecas que son necesarios para su correcto funcionamiento (dependencias) y algunos otros datos adicionales relacionados con la aplicación en cuestión.

Existen varios formatos de paquetes en Linux, por ejemplo:

- *Paquetes de la distribución Red Hat y derivados*. Se identifican porque su nombre termina con la extensión `.rpm`. La herramienta para manejar estos paquetes se llama `rpm`. Para instalar un cierto paquete llamado `paquete.rpm` se debe ejecutar la orden `rpm -i paquete.rpm`.

- *Paquetes de la distribución Debian*. Se identifican porque su nombre termina con la extensión `.deb`. Existen dos herramientas para manejar estos paquetes: `apt` y `dpkg`. Para instalar con `dpkg` un cierto paquete llamado `paquete.deb` se debe ejecutar la orden `dpkg -i paquete.deb`.

- *Paquetes de la distribución Slackware*. Se identifican porque su nombre termina con la extensión `.tgz`. Las herramientas para manejar estos paquetes son dos: `pkgtool` para seleccionar los paquetes a instalar desde un menú amigable y `installpkg` para instalar los paquetes.

3.A.4 Trabajo en red

En ocasiones es necesario acceder a un equipo remoto al que el usuario no se encuentra directamente conectado, o bien acceder a un archivo ubicado en otro sistema diferente al del usuario. La forma más sencilla de resolver este problema es mediante la conexión de los equipos a una red. Si el sistema está integrado en una red, se puede transferir archivos entre los equipos, iniciar una sesión de forma remota y ejecutar aplicaciones o comandos en equipos remotos.

Es posible acceder de una máquina a otra siempre y cuando exista un camino físico por el que hacerlo, un cable directo, una red, etc. Las redes pueden ser de área local (LAN) o de área extensa (WAN) dependiendo de la extensión de la misma.

Para que un usuario de un sistema local determinado pueda acceder a un sistema remoto será necesario que tenga una cuenta de usuario en dicho sistema, y que los accesos remotos estén permitidos.

Cuando se accede a un sistema remoto, los comandos se ejecutarán en la máquina o host remoto, si bien las entradas y salidas estándar estarán en el sistema local. Se

puede decir que el sistema local se limita a interconectar al usuario con el sistema remoto, de forma transparente. Esto no es del todo cierto, pues quedan registros en el sistema local de las conexiones establecidas en los sistemas remotos.

Para acceder a un sistema habrá que conocer su dirección IP (AAA.BBB.CCC.DDD), donde cada uno de los 4 valores tomará valores entre 0 y 255) y/o nombre. Para saber si un sistema es accesible por su nombre se puede consultar el archivo `/etc/hosts`, en el que se identifican los nombres de los sistemas con su dirección IP. Este archivo solo puede ser modificado por el superusuario.

Una forma de conocer si un sistema es accesible es usando el comando `ping`, mediante el cual el sistema local envía mensajes al sistema remoto y espera respuesta. Si se recibe respuesta y no se pierde ningún paquete, es que el sistema está accesible.

Para conocer información sobre un sistema se puede utilizar el comando `nslookup` lo cual de paso servirá para saber si el sistema remoto es accesible desde el sistema local.

El comando por excelencia usado para transferir archivos entre dos sistema es `ftp`. Con él, en principio, se puede acceder a cualquier a cualquier archivo del sistema remoto. Por motivos de seguridad, lo más habitual es que el acceso esté limitado a una serie de directorios definidos por el administrador. El sistema puede admitir o no conexiones remotas de tipo anónimo.

Para copiar archivos entre sistemas se puede usar el comando `rcp`. Se pueden copiar archivos o directorios entre el sistema local y un sistema remoto o copiarlos entre sistemas remotos. Para utilizar `rcp`, el usuario debe tener permiso de lectura para los archivos que desee copiar. Así como permisos de lectura y ejecución en todos los directorios que se encuentren en la ruta del directorio. Además debe disponer de una cuenta (de inicio de sesión) en el sistema remoto.

Quizás el método más extendido para conectarse a sistemas remotos es usar el comando `telnet`. Su utilización es muy sencilla, una vez el usuario accede al sistema remoto puede trabajar en él como si estuviera conectado directamente en modo local en él. Si se dispone de cuenta en un sistema remoto, se puede utilizar el comando `rlogin` para iniciar una sesión en dicho sistema y trabajar en él como si se estuviera en modo local.

El comando `remsh` permite ejecutar uno o varios comandos en un sistema remoto sin iniciar una sesión en dicho sistema. Para ello, el usuario local y el sistema local deben estar definidos en el archivo `.rhosts` del sistema remoto.

Se puede enviar y recibir correo de usuarios del sistema o de otros sistemas, para ello, existen diferente comandos como `mail`, `mailx` o `elm`. Para enviar un mensaje a un usuario, por lo general, habrá que indicar el nombre del usuario de destino y el nombre del sistema. Se usa la nomenclatura típica de Internet: `usuario@host`.

COMPLEMENTO 3.B
Ejemplos adicionales de shell scripts

A continuación se incluyen varios ejemplos que ilustran el lenguaje de programación de los shell scripts. Se deja como tarea para el lector el descubrir el significado de cada sentencia.

♦ **Ejemplo 3B.1:**

Supóngase un shell script con el siguiente código:

```
# Programa que cambia de directorio y muestra el directorio
# actual.
echo SUBSHELL.
DIRACT=`pwd`
echo Directorio actual $DIRACT
echo Cambiando directorio en el subshell...
cd /etc
echo Ahora en directorio `pwd`
cd
echo Ahora estoy en mi directorio, que es `pwd`
```

La ejecución de este shell script tendría la siguiente salida por pantalla, supuesto que el directorio de trabajo desde donde se invoca es `/home/darkseid`:

```
SUBSHELL.
Directorio actual /home/darkseid
Cambiando directorio en el subshell...
Ahora en directorio /etc
Ahora estoy en mi directorio, que es /home/darkseid
```

♦

♦ **Ejemplo 3B.2:**

El siguiente shell script muestra la hora del sistema cada segundo durante 1 minuto.

```
Cont=0
```

```
while [$Cont -le 60 ]
 do
  date
  Cont=`expr $Cont + 1`
 sleep 1
 done
```

♦

♦ **Ejemplo 3B.3:**

El siguiente shell script muestra por pantalla el día de la semana que fue ayer.

```
# En la variable HOY se almacena el numero de dia
# para hoy.
HOY=`date +%w`
AYER=`expr $HOY - 1` # y en ayer, el valor de HOY menos 1

# date +%w devuelve el día de la semana en formato numérico,
# con valores comprendidos entre 0 (domingo) y 6 (sábado).
# En este caso, ayer tomará valores entre -1 y 5.

echo "Ayer fue \c"
case $AYER in
 -1) echo "Sabado";;
  0) echo "Domingo";;
  1) echo "Lunes";;
  2) echo "Martes";;
  3) echo "Miercoles";;
  4) echo "Jueves";;
  5) echo "Viernes";;
  *) echo "un dia raro, pues no existe";;
esac
```

♦

♦ **Ejemplo 3B.4:**

Supóngase un shell script con el siguiente código:

```
#!/bin/sh
#
echo "Introduce una cadena: \c"
read NOMBRE
LONGITUD=`echo $NOMBRE | wc -c`
while [ $LONGITUD -gt 0 ]
 do
  NOMBREALREVES="$NOMBREALREVES"`echo $NOMBRE | cut -c$LONGITUD`
```

```
  LONGITUD=`expr $LONGITUD - 1`
 done
echo "\n$NOMBRE\n$NOMBREALREVES"
```

La ejecución de este shell script pide al usuario que introduzca una cadena de caracteres y la muestra, por pantalla del derecho y del revés. Un ejemplo de ejecución sería:

```
Introduce una cadena: Hola que tal?
Hola que tal?
¿lat euq aloH
```

♦

♦ **Ejemplo 3B.5:**

El siguiente shell script simula una papelera de reciclaje. Así en lugar de borrar un archivo, lo que se hace es guardarlo en un subdirectorio, con el fin de evitar borrados accidentales.

```
#!/bin/ksh
#
if [ $# -gt 0 ]
 then
  for i in $*
   do
    echo "Moviendo $i al directorio $HOME/borrados"
    if [ `mv $i $HOME/borrados 2> /dev/null` ¡= 0 ]
     then
      echo "Error, no se puede mover $i"
    fi
  done
 else
  echo "Error: hay que especificar argumentos"
  echo "$0 archivo1 [archivo2] ..."
  return 1
fi
return 0
```

♦

♦ **Ejemplo 3B.6:**

El siguiente shell script ejecuta un proceso largo y mientras tanto va mostrando un punto en la pantalla, simulando que el proceso va progresando.

```
function puntos ()
 {
 if [ $1 ]      # Si se ha pasado algun argumento
  then
   INTERVALO=$1  # Considerar el intervalo como tal
```

```
  else
    INTERVALO=5   # Si no se ha pasado, considerar 5
 fi
 while true    # Ejecutar siempre, sin fin.
  do
   echo ".\c"  # Imprimir un punto si salto de linea
   sleep $INTERVALO # Esperar el intervalo especificado
  done
 }
# Programa principal
# Lo que sea
puntos 2 &  # Se llama a la funcion puntos, con intervalos de 2 sg
[ programa que tarda mucho ]
kill -9 $!  # Se mata el ultimo proceso lanzado en background
# Lo que sea
# Fin del programa
```

♦

♦ **Ejemplo 3B.7:**

El siguiente shell script comprueba la hora de los diferentes sistemas conectados a un sistema determinado.

```
#
# En la variable SISTEMAS_TOT se definen los sistemas de los que
# se intentara obtener la hora.

SISTEMAS_TOT="maquina1 maquina2 maquina3 maquina4 maquina5"

#
# La funcion hora se encarga de pedir la hora al sistema
# especificado, filtrarla y mostrarla por pantalla.
#
hora()
      {
      hora=´telnet $1 daytime 2> /dev/null | grep ":" ´
      echo "$hora -----> $1"
      }
#
# Comprobar si el sistema esta accesible, y si lo esta,
# añadirlo a la variable SISTEMAS, que será la que contiene
# los sistemas accesibles.
#
```

```
for SISTEMA in $SISTEMAS_TOT
      do
            /usr/sbin/ping $SISTEMA -n 1 | grep " 0% packet loss" >
            /dev/null
            if [ $? = 0 ]
                  then
                        SISTEMAS="$SISTEMAS $SISTEMA"
                  else
                        echo "$SISTEMA no esta accesible"
            fi
      done

# Para los sistemas accesibles comprobar la hora de los mismos en
# background para minimizar diferencias.

for SISTEMA in $SISTEMAS
      do
            hora $SISTEMA &
      done

#
# Esperar a que finalice la ejecucion de todas las tareas en
# background.
#

wait

#
#Fin de programa
```

♦

COMPLEMENTO 3.C
Ficheros de arranque de un intérprete de comandos

Un intérprete de comandos es un programa ejecutable que puede ser invocado con diferentes opciones y argumentos. De forma general en función de su forma de invocación se distinguen tres tipos de intérpretes de comandos:

- *Intérprete de entrada* (*login shell*). Es aquél cuyo primer carácter del argumento cero es un `'-'`, o uno que ha sido llamado con la opción `--login`. Cuando un usuario entra en el sistema se suele invocar a un intérprete de entrada que entre otras funciones configura el valor de ciertas variables de entorno e invoca a un intérprete interactivo.

- *Intérprete interactivo* (*interactive shell*). Es uno cuya entrada y salida estándares están conectadas a terminales, es decir el usuario puede interactuar con ellos, o uno que ha sido llamado con la opción `-i`.
- *Intérprete no interactivo* (*non-interactive shell*). Es uno que no cumplen ninguna de las condiciones para ser un intérprete interactivo. Un ejemplo típico de intérprete no interactivo es un subintérprete invocado para ejecutar un shell script.

Supóngase que el intérprete de comandos utilizado es el intérprete `bash` (para conocer los ficheros de arranque de otros tipos de intérpretes se puede consultar su página correspondiente en el manual de ayuda de UNIX). Los párrafos que se incluyen a continuación describen como el intérprete `bash` ejecuta sus ficheros de arranque. Si cualquiera de los ficheros existe pero no puede ser leído, entonces `bash` informa de un error.

Cuando un usuario entra en el sistema se invoca un intérprete de entrada `bash` que en primer lugar lee y ejecuta los comandos del fichero `/etc/profile`, si este fichero existe. Después busca los ficheros `~/.bash_profile`, `~/.bash_login`, y `~/.profile` en ese orden. Nótese que el carácter tilde '~' que aparece en la rutas de acceso es una forma abreviada de designar al directorio de trabajo inicial del usuario. Mientras que el carácter '`.`' al comienzo del nombre de un fichero indica que se trata de un fichero oculto. `Bash` lee y ejecuta las órdenes del primero de estos ficheros que exista y que se pueda leer. La opción `--noprofile` puede emplearse cuando se llame al intérprete para inhibir este comportamiento.

Cuando el intérprete de entrada termina lee y ejecuta las órdenes del fichero `~/.bash_logout` si éste existe.

Cuando se arranca un intérprete interactivo `bash` que no es un intérprete de entrada, éste lee y ejecuta órdenes desde `~/.bashrc`, si existe. En algunas distribuciones también existe un fichero `/etc/bashrc` que es leído y ejecutado antes `~/.bashrc`. Esto puede evitarse mediante la opción `--norc`. La opción `--rcfile fichero` forzará a `bash` a leer y ejecutar órdenes desde un fichero predeterminado en vez desde `~/.bashrc`.

Cuando `bash` se arranca como un intérprete no interactivo busca la variable de entorno `BASH_ENV`, expande su valor si está definida, y utiliza el valor expandido como el nombre de un fichero a leer y ejecutar. `Bash` se comporta como si se ejecutaran las

siguientes órdenes:

```
if [ -n "$BASH_ENV" ]; then . "$BASH_ENV"; fi
```

Debe tenerse en cuenta que el valor de la variable `PATH` no es utilizado para buscar el nombre del fichero.

De los ficheros enumerados en los párrafos anteriores comentar que aquellos que llevan en el nombre la palabra `profile` o `login` contienen datos de inicialización que son utilizados en el momento de entrar un usuario en el sistema. El fichero `/etc/profile` pertenece al superusuario, mientras que los ficheros `~/.bash_profile`, `~/.bash_login`, y `~/.profile` pertenecen al usuario.

Asimismo los ficheros que terminan en `rc` contienen datos de inicialización del intérprete que son consultados cada vez que se arranca dicho intérprete. El fichero `/etc/bashrc` es propiedad del superusuario, mientras que el fichero `~/.bashrc` pertenece al usuario.

El usuario puede utilizar estos ficheros de arranque para configurar a su gusto el valor de ciertas variables de entorno, definir nuevas variables, definir alias, etc.

♦ **Ejemplo 3C.1:**

Si se desea que una cierta ruta `ruta_deseada` se encuentre añadida en la variable `PATH` en el momento de arrancar un intérprete de comandos se pueden incluir en alguno de sus ficheros de arranque el siguiente par de líneas:

```
PATH=$PATH:/ruta_deseada
export PATH
```

♦

♦ **Ejemplo 3C.2:**

Si se desea conocer si un intérprete de comandos es interactivo se pueden incluir en alguno de sus ficheros de arranque las siguientes líneas:

```
if [ -z "$PS1" ]; then
      echo Intérprete no interactivo
else
      echo Intérprete interactivo
fi
```

En ellas se comprueba el valor de la variable del intérprete `PS1`, si está desactivada el intérprete es no interactivo, en caso contrario es interactivo. Otra forma alternativa de saber si un intérprete es interactivo consiste en comprobar el valor del parámetro especial `-`, con las siguientes líneas

```
case "$-" in
```

```
        *i*)  echo Intérprete interactivo
        *)    echo Intérprete no interactivo
        esac
```

Este parámetro contiene el valor `i` si el intérprete es interactivo.

♦

COMPLEMENTO 3.D
La función de librería system

En ocasiones puede resultar útil poder ejecutar desde un programa ejecutable una orden del intérprete de comandos. Para ello se usa la función de librería `system` cuya declaración es

```
#include <stdlib.h>
int system (const char *orden);
```

Esta función ejecuta el comando `orden` invocando al intérprete de comandos `/bin/sh` y regresa después de que la orden se haya terminado de ejecutar. Si la invocación a `/bin/sh` falla, el valor devuelto por `system` es `127`. Si se puede invocar al intérprete de comandos pero se produce algún error en la ejecución de la orden el valor devuelto es `-1`. Si el comando se ejecuta correctamente, la función devuelve la salida de la orden. Conviene recordar que muchas órdenes devuelven un valor después de ejecutarse y que este valor indicará si la ejecución ha sido correcta o si se ha producido algún fallo y que tipo de fallo se ha producido. Por lo general en caso de una ejecución correcta devolverán el valor 0, y en caso de fallo otro número, positivo o negativo.

♦ **Ejemplo 3D.1:**

Considérese el siguiente programa escrito en lenguaje C:

```
#include <stdlib.h>
main()
{
      int salida;
      salida=system("\n echo $PATH \n");
      printf("\n Salida=%d \n",salida);
}
```

Supóngase que al fichero ejecutable que resulta de compilar este programa se le denomina `ver_path`. Cuando se ejecute `ver_path` desde un intérprete de comandos se visualizará en pantalla el valor de la variable `PATH` (sino ocurre ningún error) y a continuación el valor contenido en la variable `salida`.

♦

CAPÍTULO 4

Estructuración de los procesos en UNIX

4.1 INTRODUCCIÓN

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 capítulo 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 qué 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 capítulo, 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 la interfaz de las llamadas al sistema. La parte final del capítulo se dedica a enumerar y describir los posibles estados de un proceso, haciéndose especial hincapié en el estado dormido.

En la explicación de este capítulo 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

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 al núcleo identificar 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. Entre otras informaciones contienen 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.

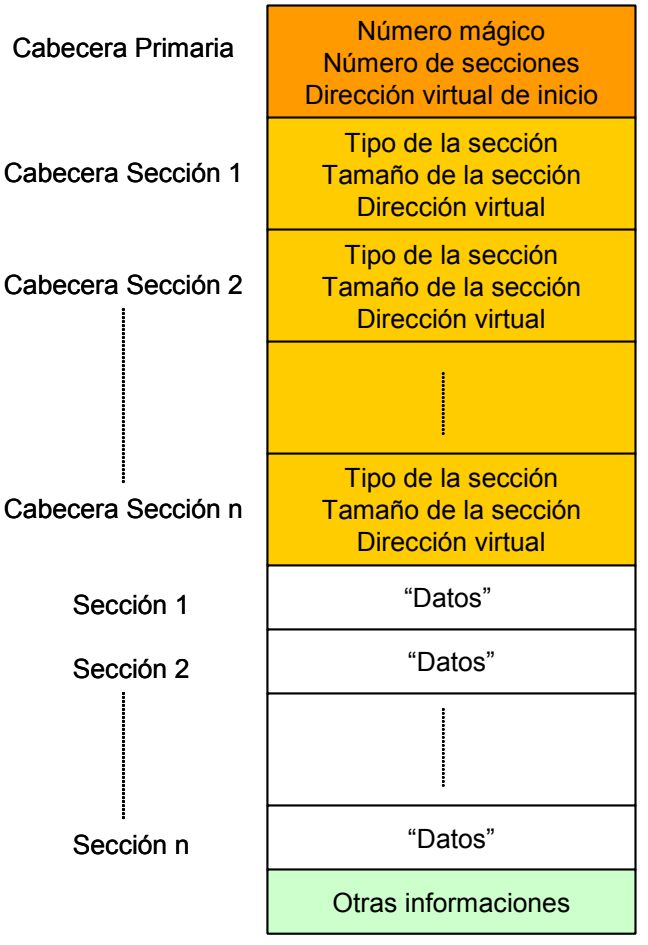


Figura 4.1: Estructura de un archivo ejecutable

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

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

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. 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 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.

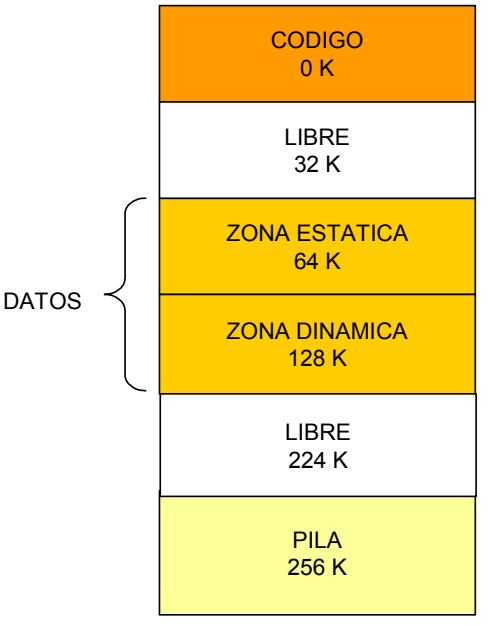


Figura 4.2: Diagrama del espacio de direcciones de memoria virtual de un proceso

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 datos 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`.
- *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 NUMÉRICOS 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

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

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

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

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 análoga a la explicada 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:**

Considérese el programa ejecutable `ejemident` que tiene el siguiente código fuente escrito en C:

```
      #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());
      }
```

Supóngase que en un cierto directorio, se tienen los siguientes archivos:

- El programa ejecutable `ejemident` 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.

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 *euid* se mantiene siempre con el valor 501, puede abrir `fichero1.txt` (`fd1≠-1`) ya que posee permiso de lectura y su *euid* coincide con el *uid* del propietario, que es él, o sea *euid*=*uid*=501. Sin embargo, no puede abrir `fichero2.txt` (`fd2=-1`) ya que éste pertenece a USUARIO2 cuyo *uid*=503, es decir *euid*≠*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 `ejemident` tiene su bit `S_ISUID` activado, al ejecutarlo el USUARIO2 su *euid* se hace igual al *uid* del propietario de este fichero que recuérdese es USUARIO1. Luego en **[1]** *uid*=503 y *euid*=501. Al cambiar su *euid* 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 *euid*, como se pone de manifiesto en **[2]**. Tras ejecutar la sentencia de `setuid(503)`, el USUARIO2 pasa a tener **[3]** su *euid*=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 *euid*=501.

* **CASO 3**: El USUARIO2 ejecuta el archivo `ejemident`, 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 *euid* se mantiene siempre con el valor 503, puede abrir `fichero2.txt` (`fd2≠-1`) ya que posee permiso de lectura y su *euid* coincide con el *uid* del propietario, que es él, o sea *euid*=*uid*=503. Sin embargo, no puede abrir `fichero1.txt` (`fd1=-1`) ya que éste pertenece a USUARIO1 cuyo *uid*=501, es decir *euid*≠*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 Capítulo 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:**

Considérese el siguiente programa escrito en lenguaje C que crea una copia del contenido de un fichero en otro fichero nuevo:

```
[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);
      }
```

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, `arg[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`). 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.

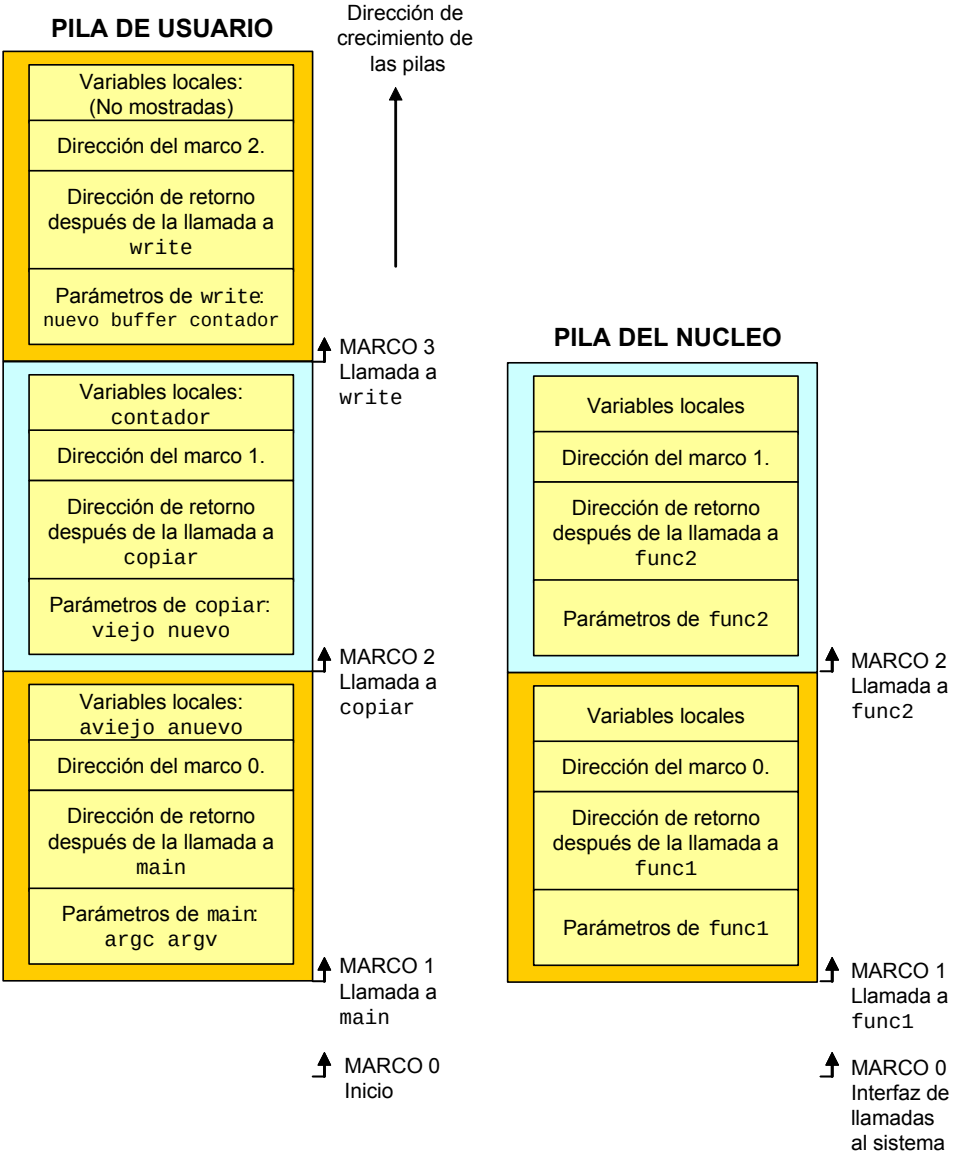


Figura 4.3: Pila del usuario y pila del núcleo en un cierto instante de la ejecución de la llamada al sistema `write`

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. Obviamente 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. 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, salvo si se captura una señal (ver sección 5.3.1).

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. A partir de ese momento 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 pila 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*, las cuales permiten al núcleo controlar a los procesos, como por ejemplo, la lista de procesos planificados para ejecución, la lista de procesos dormidos, etc.
- *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 qué 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, etc.
- *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 Área 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 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*.
- El *terminal de acceso* asociada con el proceso, si existe alguno.
- Estadísticas de uso de 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 un 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 asociada a dicho proceso, o puede implementarse dentro del área U de dicho proceso. También puede implementarse en un 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 referencias, 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.

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.

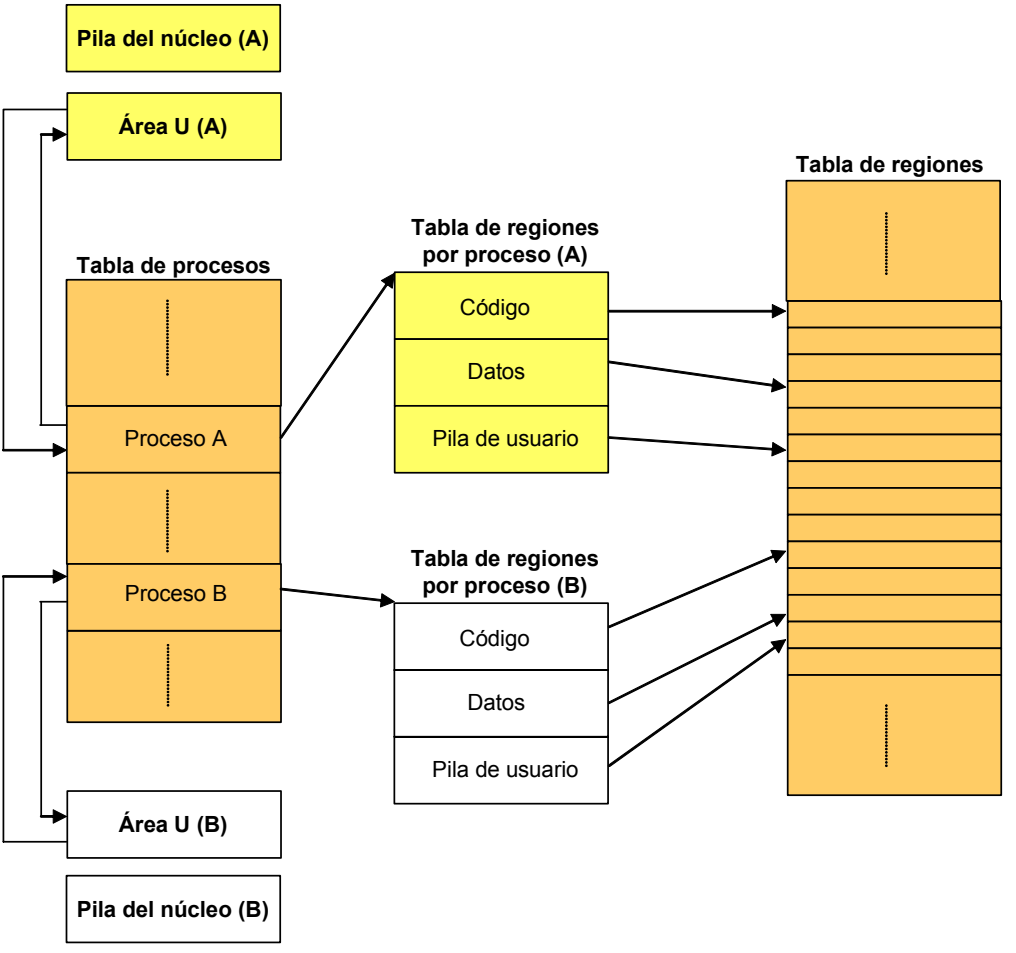


Figura 4.4. Estructura de datos del núcleo asociadas a los procesos A y B

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 contendrá el valor 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 *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.

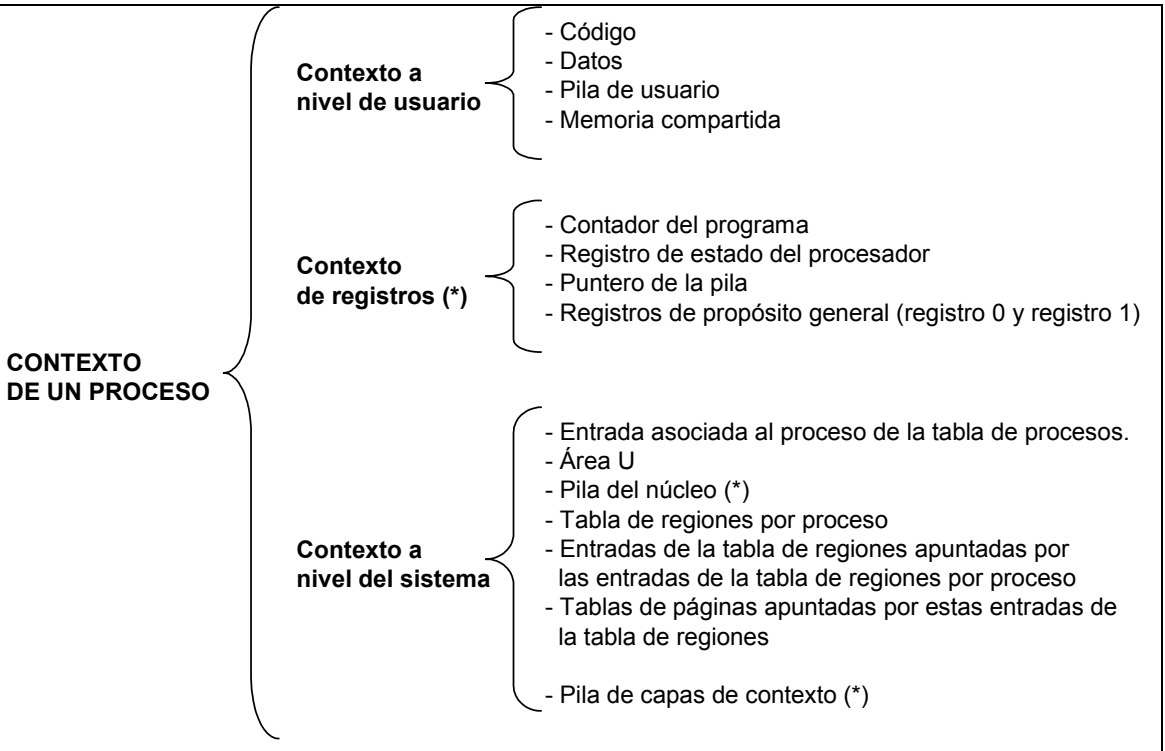
El *contexto a nivel del sistema* de un proceso está formado por: la entrada de la *tabla de procesos* asociada a dicho proceso, su *área U,* su *pila del núcleo*, su *tabla de regiones por proceso*, las entradas de la *tabla de regiones* apuntadas por las entradas de su tabla de regiones por proceso 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 formada por el *contexto a nivel de registros* y su *pila del núcleo*. Los restantes elementos del contexto de un proceso constituyen su *parte estática*.



Nota: Los elementos marcados con (*) constituyen la parte dinámica del contexto

Cuadro 4.1: Contexto de un proceso

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.

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 dónde debe almacenar una nueva capa de contexto o dónde 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 *t0* 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 va a denotar como *capa 0* y su contenido será únicamente RE_{t0} 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 *t1* 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 va a denotar como *capa 1* y su contenido será RE_{t1} y PN_{t1}. 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 *t2* 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 le va a denotar como *capa 2* y su contenido será RE_{t2} y PN_{t2}. 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 *t3* 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_{t2} y PN_{t2}, respectivamente.

5) En el instante de tiempo *t4* 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_{t1} y PN_{t1}, respectivamente.

6) En el instante de tiempo *t5* 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_{t0} 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.

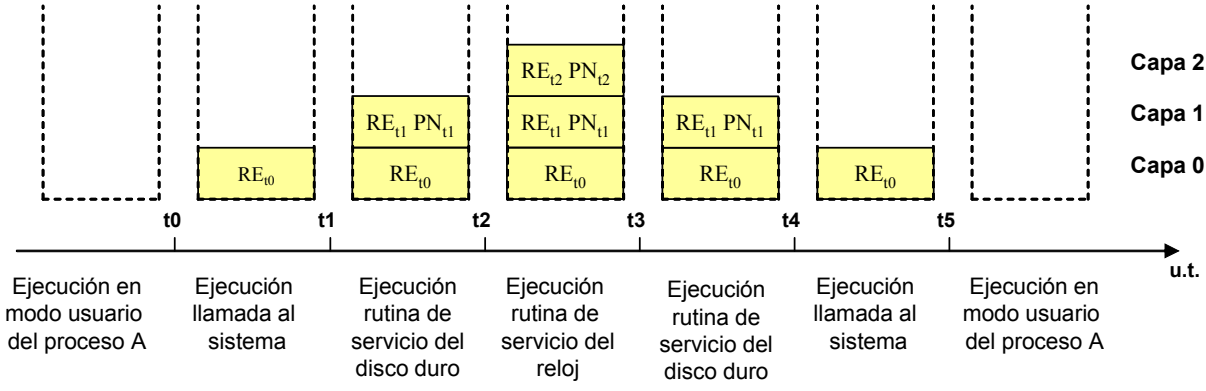


Figura 4.5: Configuración de la pila de capas de contexto del proceso A durante los distintos sucesos considerados

♦

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.* En esta circunstancia, de acuerdo con el tipo de algoritmo de planificación utilizado en UNIX, 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 basta 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 van a utilizar, por simplificar, las siguientes abreviaturas:

- *npi*, es el *nivel de prioridad de interrupción* actual almacenado en el registro de estado del procesador.

- *npii*, 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 *npii>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=npii*. 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 *npii* 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 apropiada.
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 interrupciones. En las máquinas que no disponen de un 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 atender 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.

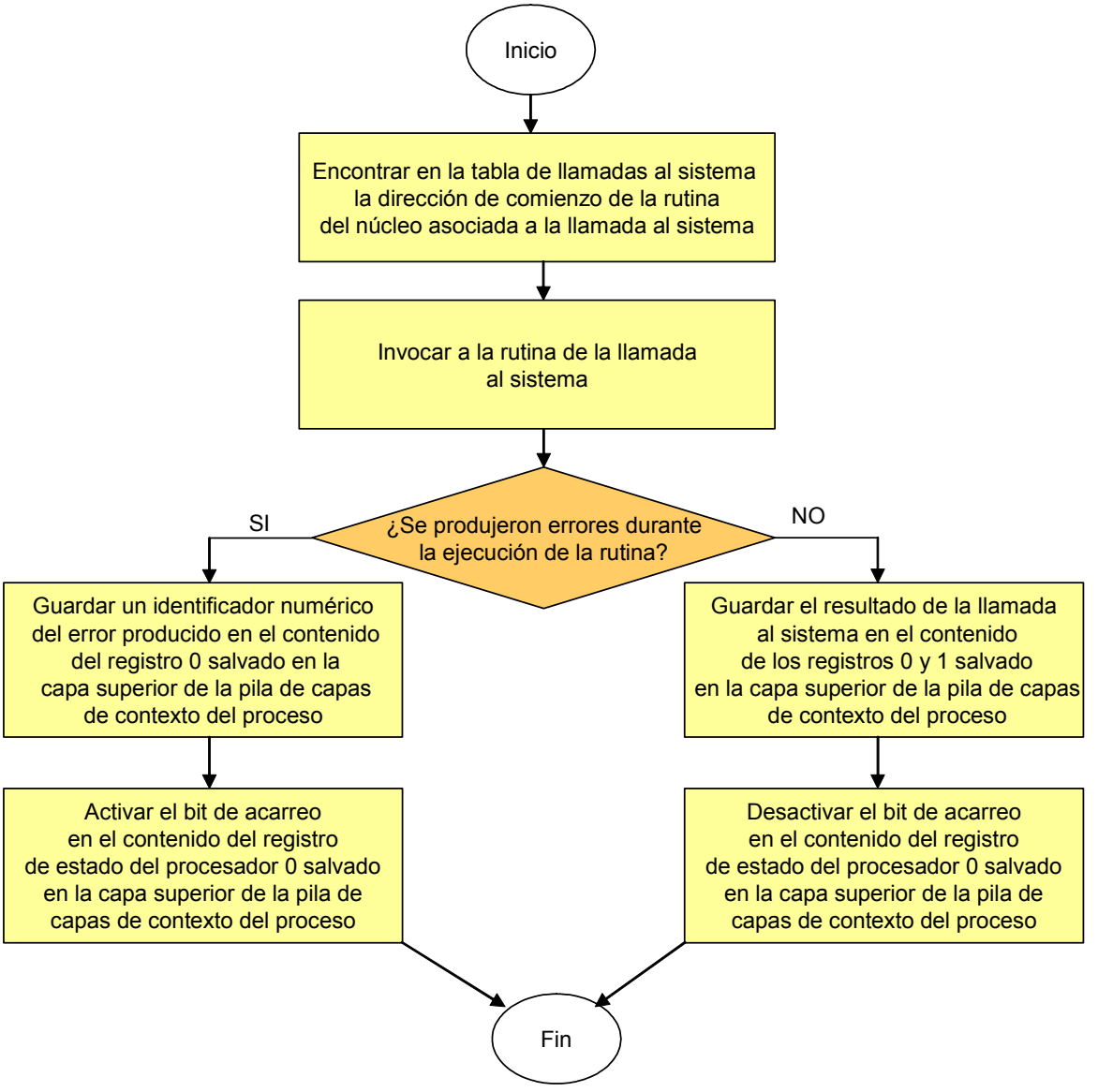


Figura 4.6: Principales acciones realizadas por el algoritmo `syscall()`

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.

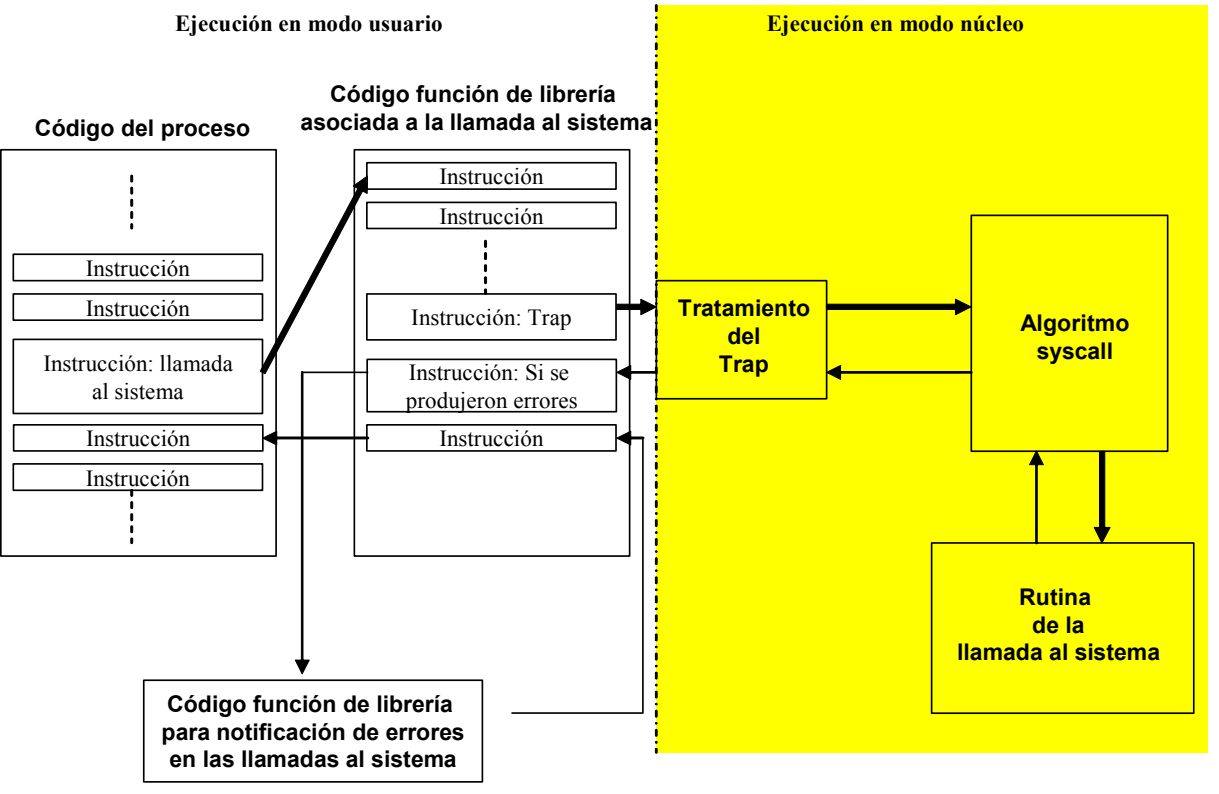


Figura 4.7: Interfaz de las llamadas al sistema

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.

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 la interfaz de las llamadas al sistema que se acaba de describir.

En dicha figura se observa cómo 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 siguiente programa 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);
}
```

Supóngase que el programa 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 del ejemplo 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, entre otras acciones, la invocación del algoritmo del núcleo `syscall()` para el tratamiento de 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 la 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 asociada 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:**

Considérese el siguiente programa escrito en lenguaje C:

```
#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]);
        {
  }
```

Este programa es un ejemplo de cómo 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:**

Considérese el siguiente programa escrito en C:

```
main()
{
      char buffer[100];
      int iden=20;
      if(read(iden,buffer,100)==-1);
      {
            perror("Mensaje");
      {
}
```

Se trata de una variación del programa del ejemplo anterior. 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*. Cada interrupción del reloj del sistema se comprueba si el proceso ejecutándose en modo usuario ha finalizado su cuanto. En caso afirmativo el proceso será expropiado de la CPU y otro proceso B pasará a 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* cuando vuelva a ser planificado para ejecución pasará directamente al estado ejecución en modo usuario.
- *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.

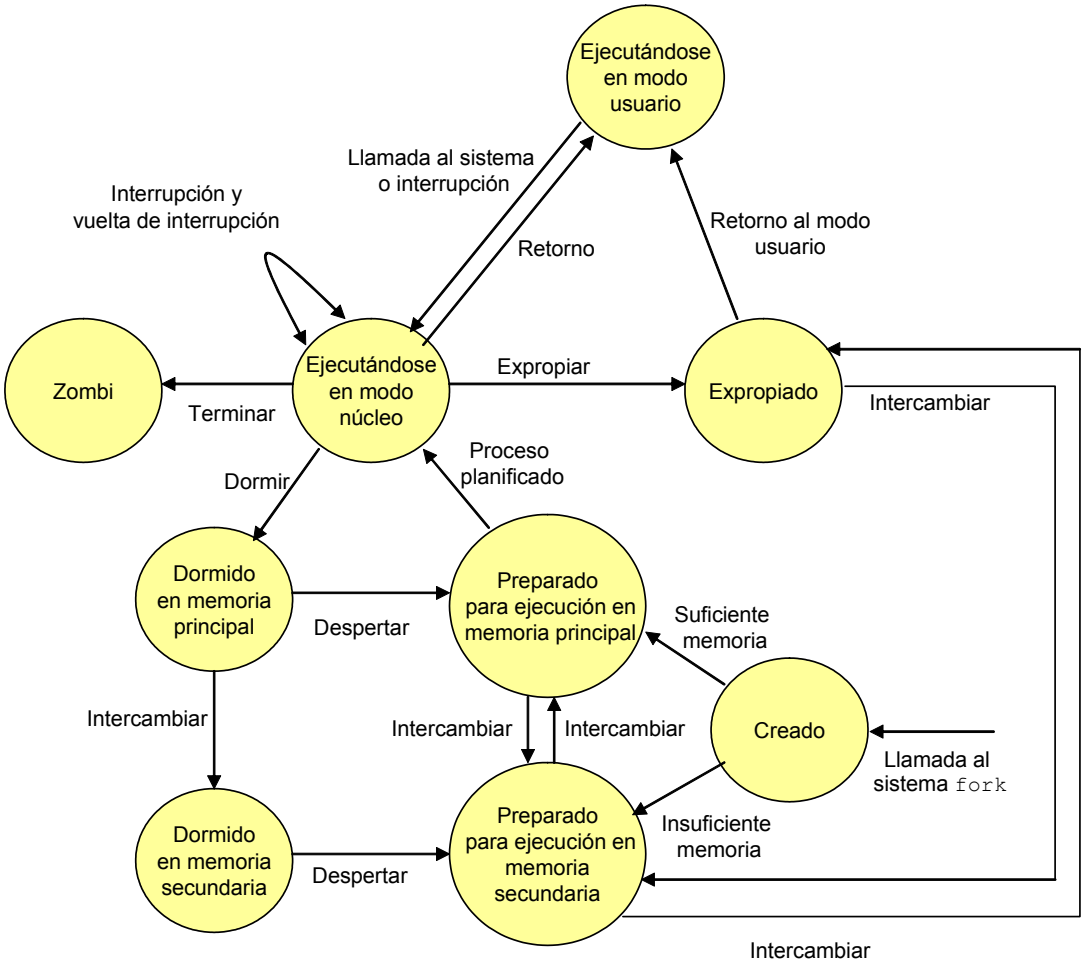


Figura 4.8: Diagrama de transiciones de estado en un sistema UNIX (SVR2 o SVR3)

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` realizada 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 comenzarán 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 las 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 esta 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 quien decide en qué momento se realizará 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 sigue 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

En el UNIX BSD4 se definieron algunos estados adicionales que no son soportados en SVR2 ni SVR3, pero sí 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 cómo 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, 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 los 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 cómo 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).

Se observa cómo 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.

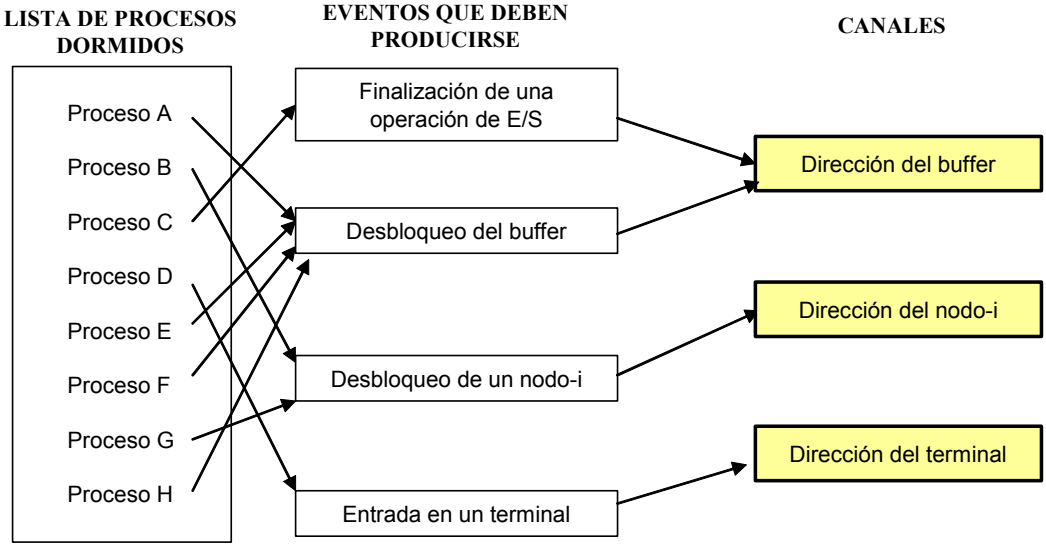


Figura 4.9: Lista de procesos dormidos, eventos que deben producirse y canales asociados

♦

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 tendrá 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 despertara 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 asociado 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 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.

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	*Evento* : Desbloqueo de un nodo-i
PRIBIO	20	*Evento*: Finalización de una operación de E/S en disco
PRIBIO+1	21	*Evento*: Desbloqueo del buffer
PZERO	25	Prioridad umbral
TTIPRI	28	*Evento*: Entrada en un terminal
TTOPRI	29	*Evento*: Salida en un terminal
PWAIT	30	*Evento*: Terminación de un proceso hijo
PLOCK	35	*Evento*: Bloqueo de un recurso
PSLEP	40	*Evento*: Recepción de una señal

Tabla 4.1: Prioridades para dormir en el UNIX BSD4.3

Como se estudiará en el capítulo 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 que 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. En ella las prioridades más altas corresponden a los valores numéricos más bajos.

CAPÍTULO 5

Control de procesos en UNIX

5.1 INTRODUCCIÓN

Este capítulo 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 la explicación de las llamadas al sistema que se tratan en este capítulo se va a tomar como referencia principalmente el núcleo de una distribución clásica como SVR3. 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 capítulo.

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 incluye un complemento dedicado a las *hebras* que son unidades computacionales utilizadas en las distribuciones modernas de UNIX.

5.2 CREACIÓN DE PROCESOS

En un sistema UNIX la única forma que tiene un usuario de crear un nuevo proceso es invocando a la llamada al sistema `fork`. El único proceso no creado mediante `fork` es el proceso 0 (*pid*=0) que 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 encuentran 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 cómo 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.

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 descriptores de ficheros del proceso hijo es una copia de la tabla de descriptores 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`.

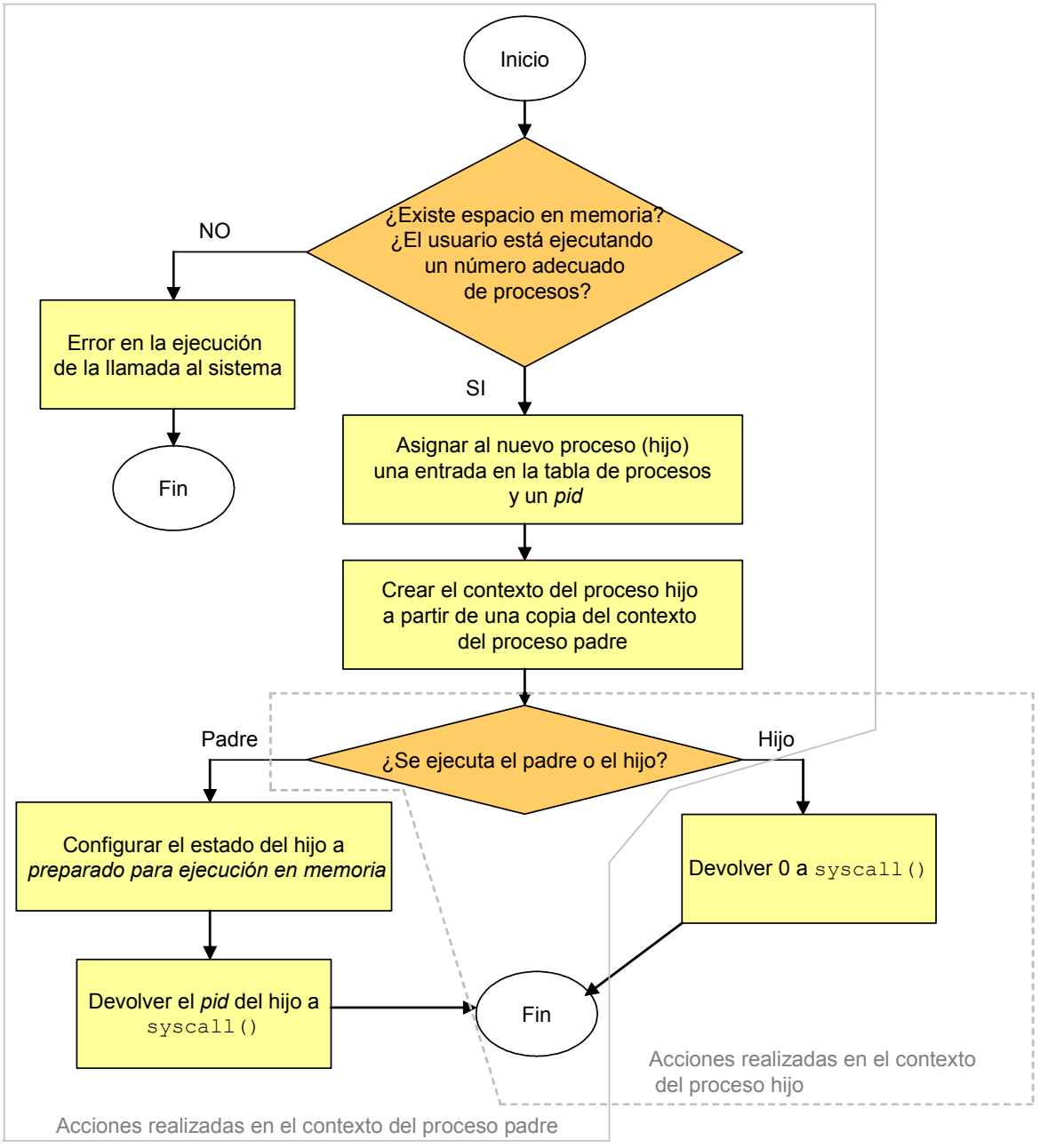


Figura 5.1: Principales acciones realizadas por el núcleo durante la ejecución de la rutina asociada a la llamada al sistema `fork`

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:**

Considérese el siguiente programa escrito en C:

```
      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 padre y el hijo */
[11]        printf("\nFinalizar\n");
      }
```

Al ejecutar este programa, 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 cómo 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 efectúe 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:**

Considérese el siguiente programa escrito en C:

```
[1]   int fichero1, fichero2;
[2]   char caracter;
[3]   main (int argc, char *argv[])
      {
[4]         if (argc!=3)
```

```
[5]               exit(1);

[6]         fichero1=open(argv[1],0444);
[7]         fichero2=creat(argv[2],0666);

[8]         if (fichero1==-1 || fichero2==-1)
[9]               exit(2);

[10]        fork();

[11]        leer_escribir();
[12]        exit(0);
      }

[13]  int leer_escribir()
      {
[14]        for(;;)
            {
[15]              if (read(fichero1, &caracter, 1)!=1)
[16]                    return(0);
[17]              write(fichero2, &caracter,1);
            }
      }
```

Supóngase que el nombre del fichero ejecutable que se crea después de compilar este programa 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 del fichero 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 `caracter`. 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, recuérdese 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 `creat` 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]** `fichero1` o `fichero2` es igual a `-1` entonces se ha producido un error durante la ejecución de la llamada al sistema `open` o `creat`, respectivamente. En dicho caso se invoca **[9]** a la llamada al sistema `exit` para terminar la ejecución del programa.

A continuación **[10]** invoca a la llamada al sistema `fork`, para crear un proceso hijo, llama **[11]** a la función `leer_escribir` e invoca **[12]** a la llamada al sistema `exit` para finalizar el programa. Con esta sentencia finaliza el código de la función `main`.

En **[13]** declara la función `leer_escribir` que no recibe ningún parámetro de entrada y devuelve un número entero como salida. En **[14]** ejecuta un bucle infinito, dentro del cual **[15]** 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 `caracter`. 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 sale del bucle **[16]** y devuelve el valor 0. En caso contrario, invoca **[17]** a la llamada al sistema `write` que escribe en el fichero `file_W` (cuyo descriptor es `fichero2`) el contenido de la variable `caracter`, es decir, 1 byte.

Cuando se ejecuta `ejfork` el sistema le asocia un proceso. A raíz de la sentencia **[10]** se crea otro proceso, hijo del anterior. Cada proceso (padre e hijo) puede acceder a copias privadas de las variables globales `fichero1`, `fichero2` y `caracter` 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 alternen 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 `[control+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.

En la distribución original del UNIX System V únicamente existían definidas 15 señales distintas. Las distribuciones BSD4 y SVR4 aumentaron el número de señales definidas a 32. Posteriormente la distribución de Solaris aumentó el número de señales a 45. En la Tabla 5.1 se muestra una recopilación de 34 señales presentes en diferentes distribuciones.

Cada señal tiene asignado un número entero entre 1 y el número máximo de señales disponible en la distribución(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 o Solaris.

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 [control + t]	ignorar	B	
SIGINT	Interrupción desde el terminal [control + 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 [control +\]	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 [control + 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 — **A** ANSI C
B Señal de BSD4.3 — **S** SVR4
P POSIX.1 — **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: Recopilación de 34 señales presentes en diferentes distribuciones de UNIX

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 usando la llamada al sistema `raise`.

- *Interrupciones del terminal*. La pulsación simultánea por parte de un usuario de teclas, como `[control+c]` o `[control+\]`, 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 dicho 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 se active transcurrido 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

[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.

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 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 un 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).

[2] El bloqueo de señales no estaba implementado en la distribución SVR2 ni en las anteriores

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 los 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 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).

3) 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 continúa 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 reiniciara 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 asociada 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.

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.

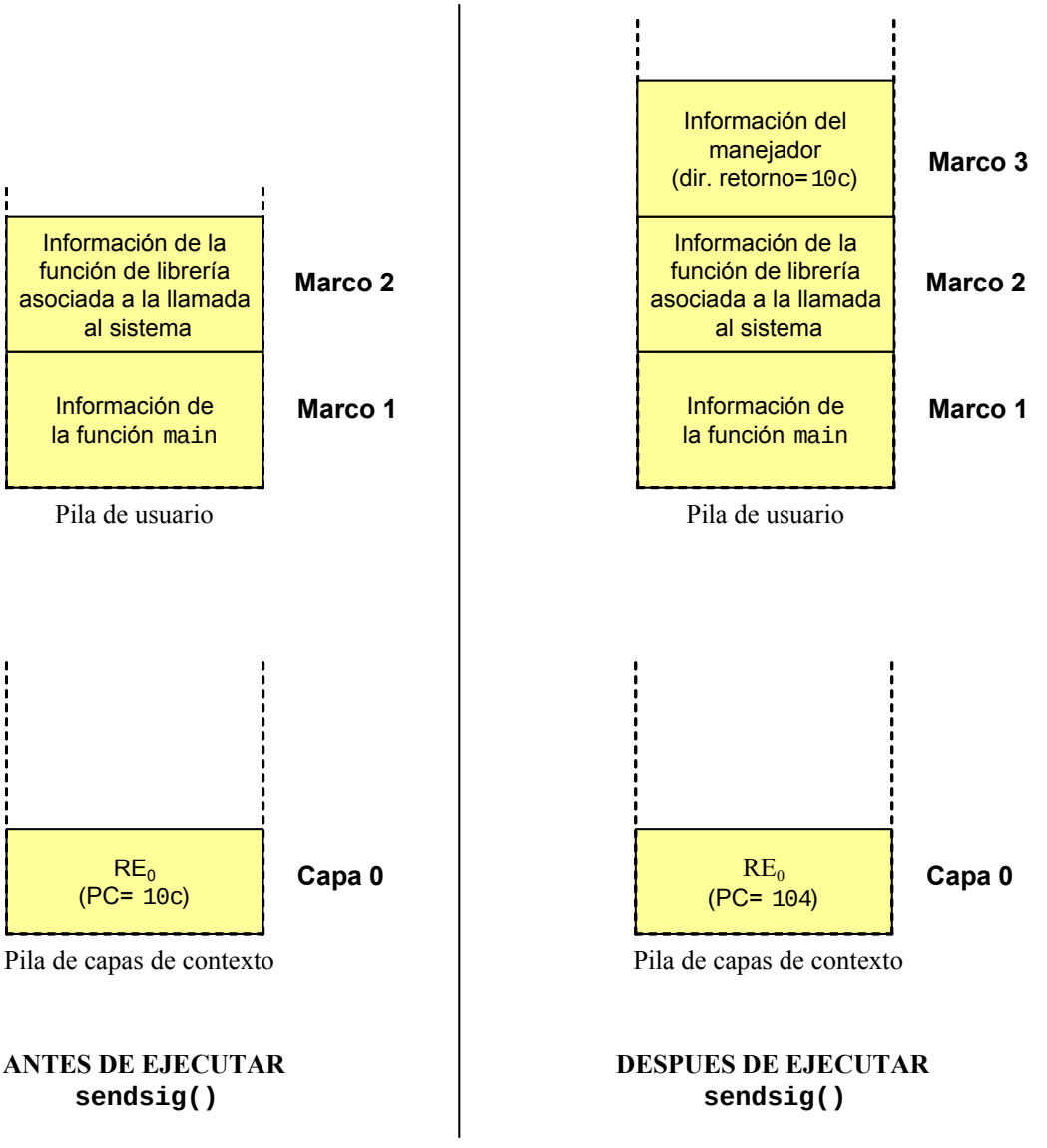


Figura 5.2: Estado de la pila de usuario y de la pila de capas de contexto asociadas a un cierto proceso antes y después de ejecutar el algoritmo `sendsig()`

♦

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()`, quien 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()`, quien 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 `[control+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 permanece 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 al de la distribución BSD4.2, diferenciándose principalmente en los nombres de algunos variables y funciones.

En el área U de un proceso se mantienen 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 bloquedas.
- `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:**

Considérese el siguiente programa escrito en lenguaje C:

```
      #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);
      }
```

El código de la función `main` de este programa 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 3.7.3 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. 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:**

Considérese el siguiente programa escrito en lenguaje C:

```
      #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);
      }
```

En este programa 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 otra 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 ejecutara 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 a no ser que se suspendiese su ejecución pulsando las teclas `[control + 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 ésta 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
```

Nótese que sino se hubiera especificado un manejador para la señal SIGUSR1 se habría ejecutado la acción por defecto que es, de acuerdo con la Tabla 5.1, terminar el proceso.

♦

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:**

Considérese el siguiente programa escrito en lenguaje C:

```
main()
{
      pause();
      printf("\nFinal\n");
}
```

Supóngase que el ejecutable que resulta de ejecutar este programa 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 qué 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.

Otra llamada al sistema para el manejo de la máscara de señales es `sigprocmask` (SVR4).

♦ **Ejemplo 5.7:**

Considérese el siguiente programa escrito en lenguaje C:

```
      #include <signal.h>
      main()
      {
[1]         long mask0;
[2]         mask0=sigsetmask(sigmask(SIGUSR1)| sigmask(SIGUSR2));
[3]         sigblock(sigmask(SIGINT));
[4]         sigsetmask(mask0);
      }
```

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.

♦

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 sí 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 porque 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.

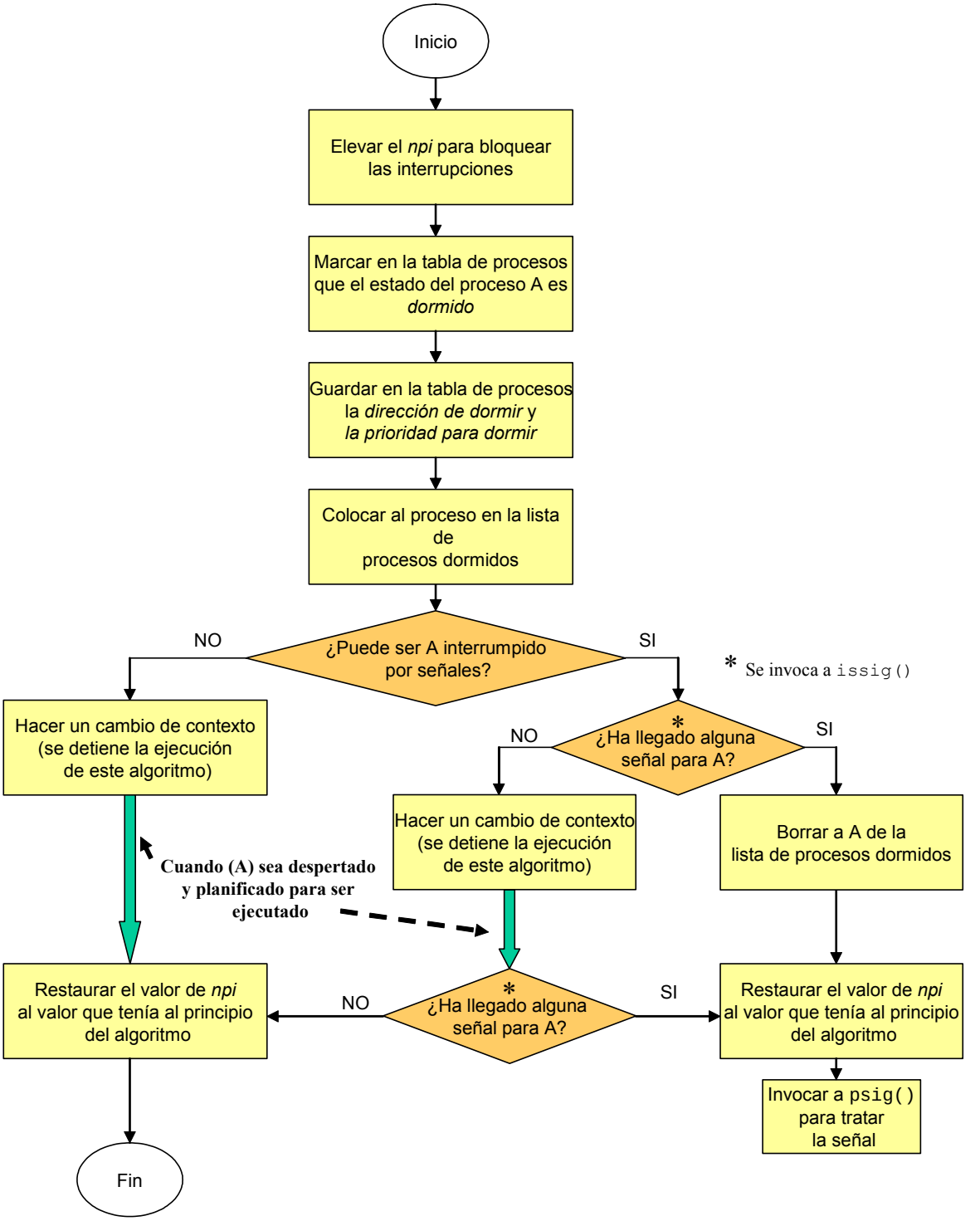


Figura 5.3: Principales acciones realizadas por el núcleo durante la ejecución del algoritmo `sleep()`

- 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 bloqueado 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.

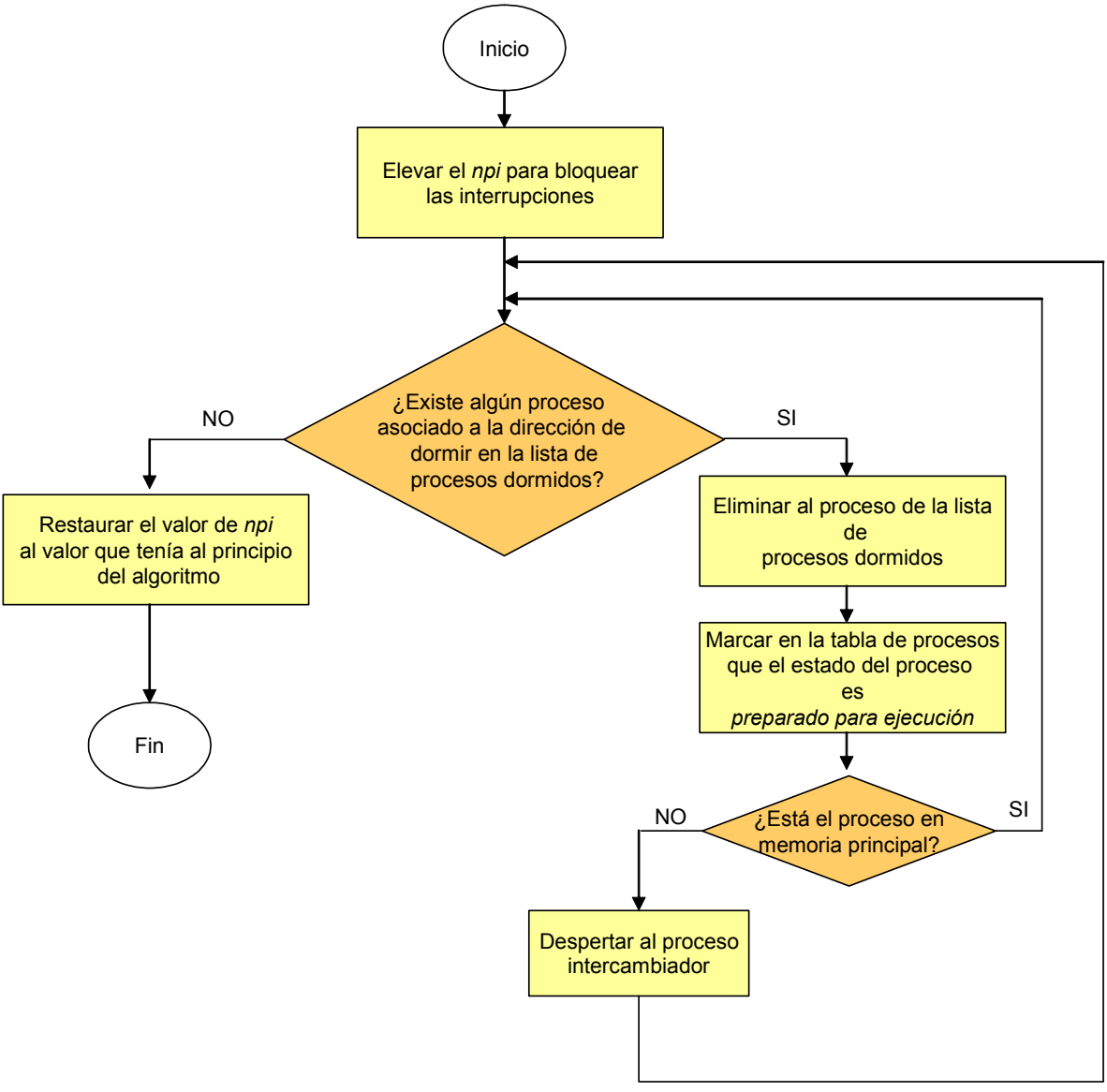


Figura 5.4: Principales acciones realizadas por el núcleo durante la ejecución del algoritmo `wakeup()`

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 tenida en cuenta por el algoritmo de planificación cuando el proceso vaya a retornar al modo usuario.

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 TERMINACIÓN 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.

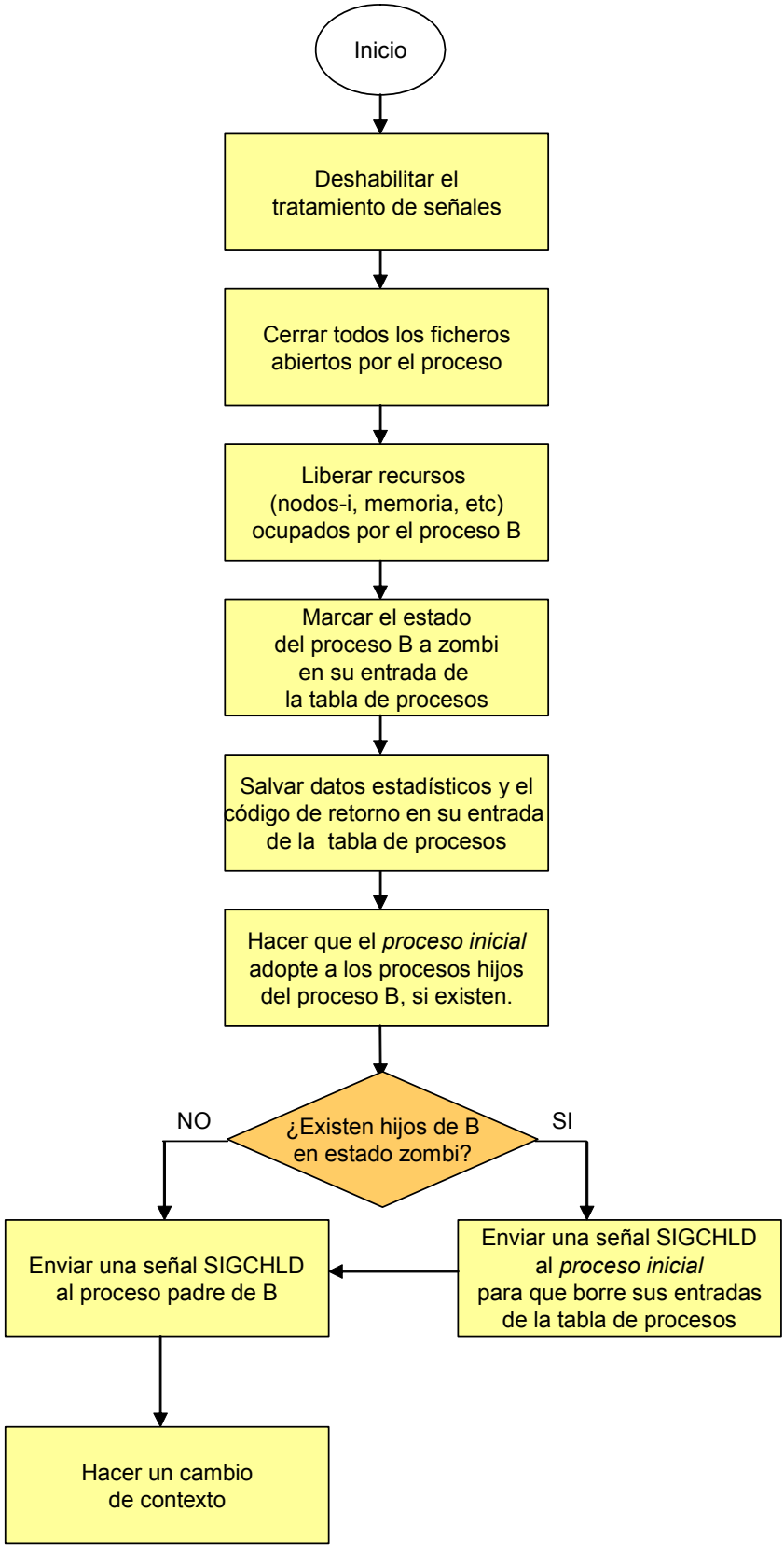


Figura 5.5: Principales acciones realizadas por el núcleo durante la ejecución del algoritmo `exit()`

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.

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 ningún 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.

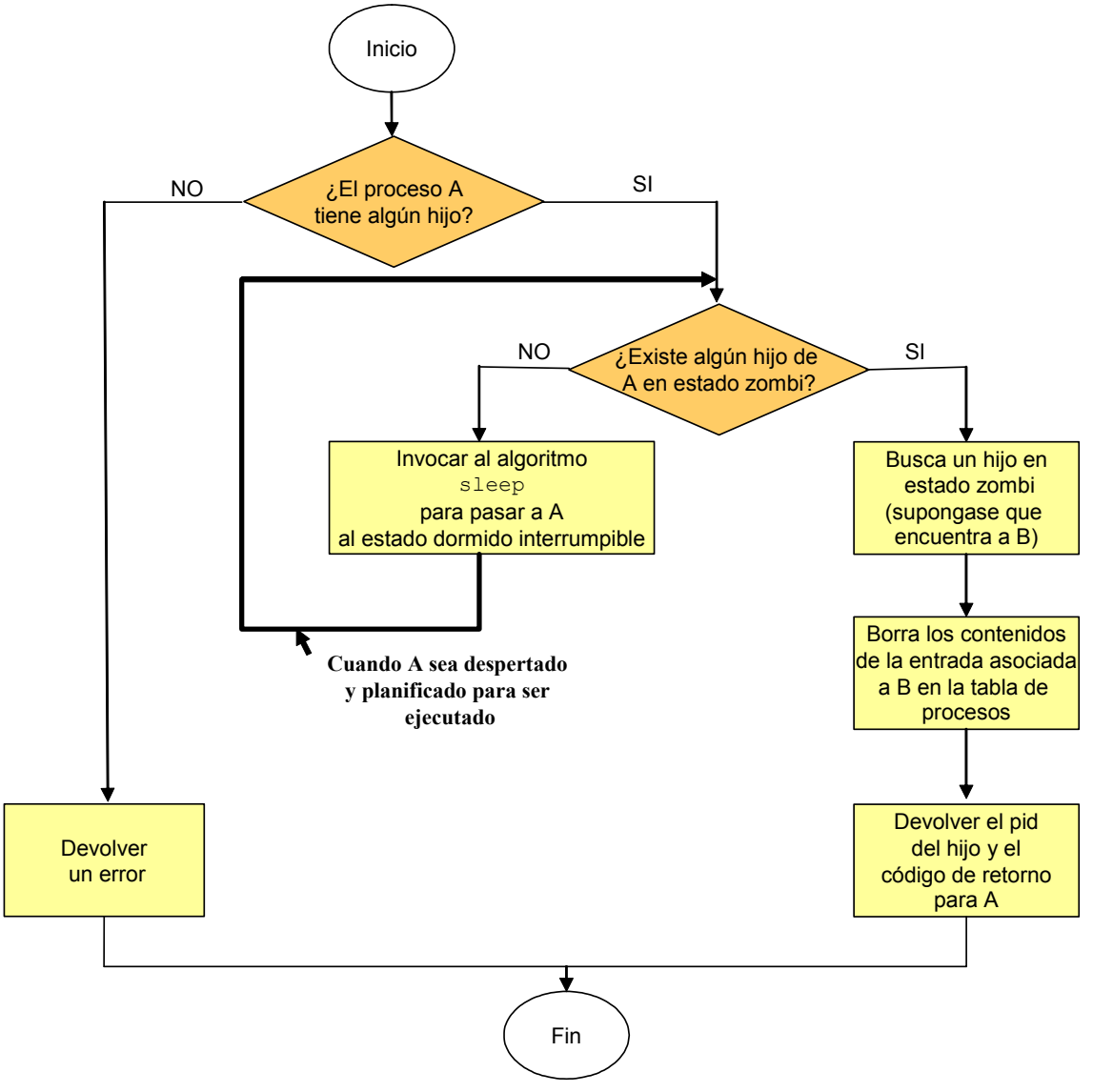


Figura 5.6: Principales acciones realizadas por el núcleo durante la ejecución del algoritmo `wait()`

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.

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 el 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");
            }
      }
```

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 esta llamada al sistema hasta que no finalice el proceso hijo.

♦

5.7 INVOCACIÓN DE OTROS PROGRAMAS

5.7.1 Funciones de librería

La llamada al sistema `exec` sirve para invocar desde un proceso a otro programa ejecutable (programa compilado o shell script). 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.

Existe toda una familia de funciones de librería asociadas a esta llamada al sistema: `execl`, `execv`, `execle`, `execve`, `execlp` y `execvp`. La sintaxis de esta familia de funciones es la siguiente:

```
resultado=execl(ruta, arg0, arg1,..., argN, NULL);
resultado=execv(ruta, argv);
resultado=execle(ruta, arg0, arg1,..., argN, NULL, envp);
resultado=execve(ruta, argv, envp);
resultado=execlp(fichero, arg0, arg1,..., argN, NULL);
resultado=execvp(fichero, argv);
```

En todas estas funciones, `ruta` es la ruta del fichero ejecutable que es invocado. `Fichero` es el nombre de un fichero ejecutable, la ruta del fichero se construye buscando el fichero en los directorios indicados en la variable de entorno `PATH`.

`Arg0`, `arg1`,...,`argN` son punteros a cadenas de caracteres y constituyen la lista de argumentos o parámetros que se le pasa al nuevo programa. Por convenio, al menos `arg0` está presente siempre y apunta a una cadena idéntica a `ruta` o al ultimo componente de `ruta`. Para indicar el final de los argumentos siempre a continuación del último argumento `argN` se pasa un puntero nulo `NULL`.

`Envp` es un array de punteros a cadenas de caracteres terminado en un puntero nulo que constituyen el *entorno* en el que se va ejecutar el nuevo programa. Cuando un programa es invocado con una llamada al sistema `exec`, el sistema pone a su disposición un array de cadenas de caracteres conocido como *entorno*. A cada cadena se le conoce como *variable de entorno* (ver sección 3.6.8). Por convenio, cada una de estas cadenas tiene la forma:

```
VARIABLE_ENTORNO=VALOR_VARIABLE
```

Para obtener el valor de una variable de entorno determinada o para declarar nuevas variables, se pueden usar las funciones `getenv` y `putenv` cuyos prototipos se encuentran en el fichero de cabecera `<stdlib.h>`.

Para los programas que se ejecutan mediante una llamada a `execl`, `execv`, `execlp` o `execvp`, el entorno se encuentra accesible únicamente a través de la variable global `environ` declarada en la librería C. Para los programas que se invocan con las llamadas `execle` y `execve`, el entorno es accesible también a través del parámetro `envp`. El entorno de un proceso es heredado por todos sus procesos hijos.

Si el programa que es invocado es un programa compilado, es decir, tiene un número mágico en su cabecera primaria que lo identifica como directamente ejecutable, entonces recibe los parámetros `arg0`, `arg1`,...,`argN` o `argv` y `envp` a través de la función principal `main`. Para pasar `envp` a `main`, ésta se debe declarar con tres argumentos formales, en vez de con los dos argumentos tradicionales (ver sección 1.8.5):

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

Si el fichero invocado no dispone de un número mágico que lo identifica como un programa compilado directamente ejecutable entonces es considerado como un shell script y se le pasa al intérprete de comandos `/bin/sh` para que lo ejecute.

Si la llamada al sistema `exec` devuelve el control al programa desde que se invoca, es porque no se ha ejecutado correctamente, entonces en `resultado` se almacenará el valor `-1` y en la variable `errno` estará el código del error producido.

5.7.2 El algoritmo exec()

Al tratar la llamada al sistema `exec` desde alguna de sus funciones de librería asociadas, el núcleo invoca a la rutina o algoritmo `exec()`. Este algoritmo requiere como parámetros de entrada, la ruta, la lista de argumentos o 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 e*uid* o el e*gid*, respectivamente, del proceso invocador para que sea igual 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.

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.

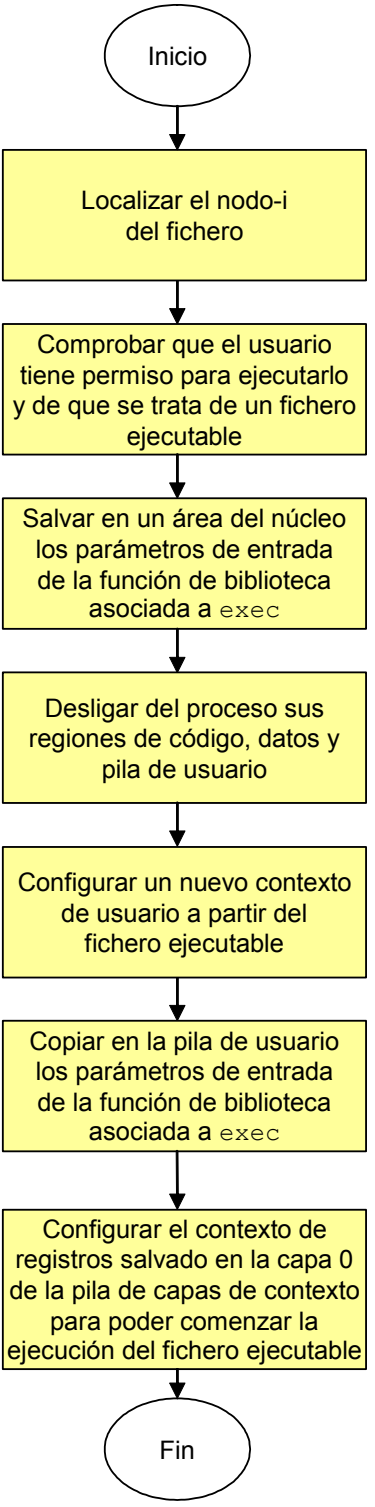


Figura 5.7: Principales acciones realizadas por el núcleo durante la ejecución del algoritmo `exec()`

Es importante darse cuenta de 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");
      }
```

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.

♦

COMPLEMENTO 5.A
Hebras

5.A.1 Motivación

Muchos programas deben realizar varias tareas, en gran medida independientes, que no tienen necesidad de ser ejecutadas secuencialmente. Por ejemplo, un servidor con una base de datos puede recibir y atender numerosas peticiones de clientes. Puesto que las peticiones no tienen porque ser servidas en un orden particular, pueden ser tratadas como unidades de ejecución independientes, las cuales en principio podrían ejecutarse en paralelo. Obviamente, la aplicación se comportaría mejor si el sistema dispusiera de mecanismos para la ejecución en paralelo de las subtareas.

En los sistemas UNIX tradicionales, un programa como el comentado utiliza múltiples procesos. La mayoría de las aplicaciones de un servidor tienen un proceso receptor de escucha que espera por las peticiones de los clientes. Cuando una petición llega, el proceso receptor crea un nuevo proceso con la llamada al sistema `fork` para servir la petición. Puesto que el servicio de una petición a menudo requiere operaciones de E/S que pueden bloquear el proceso, esta aproximación de múltiples procesos posee algunos beneficios de concurrencia incluso en sistemas con un único procesador.

Considérese ahora el caso de una aplicación de tipo científico que calcula los valores de varios elementos de un array, siendo cada elemento independiente de los demás. Se podría crear un proceso diferente para cada elemento del array y conseguir paralelismo encaminando cada proceso hacia diferentes computadoras, o quizás hacia los diferentes CPUs de un sistema multiprocesador. Si un proceso se bloquea debido a que debe esperar por una operación de E/S o por el servicio de un fallo de página, otro proceso podría progresar mientras tanto.

El uso de múltiples procesos para implementar una aplicación tiene algunas desventajas obvias. Crear todos estos procesos añade una sobrecarga (overhead) no despreciable al sistema, ya que `fork` suele ser usualmente una llamada al sistema bastante costosa en el uso de recursos. Además, puesto que cada proceso tiene su propio espacio de direcciones, se deben usar mecanismos de intercomunicación entre procesos como el paso de mensajes o memoria compartida. Asimismo, se requiere un trabajo adicional para: encaminar los procesos hacia diferentes máquinas o procesadores, pasar información entre los procesos, esperar a su finalización y reunir todos los resultados. Finalmente, los sistemas UNIX no tiene entornos de trabajo apropiados para compartir ciertos recursos.

Con estos ejemplos se han ilustrado las deficiencias de la abstracción de proceso y la necesidad de disponer de mejores servicios para el procesamiento paralelo, que pueden resumirse de la siguiente forma:

- Muchas aplicaciones son inherentemente paralelas por naturaleza y requieren un modelo de programación que soporte el paralelismo. Los sistemas UNIX tradicionales fuerzan a tales aplicaciones a ejecutar secuencialmente sus tareas independientes o a idear raros e ineficientes mecanismos para realizar las operaciones múltiples.
- Los procesos tradicionales no pueden aprovecharse de las arquitecturas de multiprocesador, puesto que un proceso sólo puede usar un único procesador a la vez. Una aplicación debe crear varios procesos separados e encaminarlos hacia los procesadores disponibles. Estos procesos deben encontrar la forma de compartir la memoria y los recursos, además de sincronizar sus tareas unos con otros.

Las distribuciones de UNIX modernas resuelven las limitaciones que aparecen con el modelo de proceso proporcionando el modelo de *hebra* (*thread*). La abstracción hebra representa a una unidad computacional que es parte de un trabajo de procesamiento de una aplicación. Estas unidades interaccionan entre si muy poco, por lo que son prácticamente independientes.

De forma general un proceso se puede considerar como una entidad compuesta que puede ser dividida en dos componentes: un conjunto de hebras y una colección de recursos. La hebra es un objeto dinámico que representa un punto de control en el proceso y que ejecuta una secuencia de instrucciones. Los recursos (espacio de direcciones, ficheros abiertos, credenciales de usuario, cuotas,...) son compartidos por todas las hebras de un proceso. Además cada hebra tiene sus objetos privados, tales como un contador de programa, una pila y un contador de registro. Un proceso UNIX tradicional tiene una única hebra de control. Los sistemas multihebras como son SVR4 y Solaris extienden este concepto permitiendo más de una hebra de control en cada proceso.

En función de sus propiedades y usos se distinguen tres tipos diferentes de hebras:

- *Hebras del núcleo*: son objetos primitivos no visibles para las aplicaciones.
- *Procesos ligeros*: son hebras visibles al usuario que son reconocidas por el núcleo y que están basadas en hebras del núcleo.
- *Hebras de usuario*: Son objetos de alto nivel no visibles para el núcleo. Pueden

utilizar procesos ligeros, si éstos son soportados por el núcleo, o pueden ser implementadas en un proceso UNIX tradicional sin un apoyo especial por parte del núcleo.

5.A.2 Hebras del núcleo

Una hebra del núcleo no necesita ser asociada con un proceso de usuario. Es creada y destruida internamente por el núcleo cuando la necesita. Se utiliza para ejecutar una función especifica como por ejemplo, una operación de E/S o el tratamiento de una interrupción. Comparte el código del núcleo y sus estructuras de datos globales. Además posee su propia pila del núcleo. Puede ser planificada independientemente y utiliza los mecanismos de sincronización estándar del núcleo, tales como `sleep()` y `wakeup()`.

Las hebras del núcleo resultan económicas de crear y usar, ya que los únicos recursos que consumen son la pila del núcleo y un área para salvar el contexto a nivel de registros cuando no se están ejecutando. También necesitan de alguna estructura de datos para mantener información sobre planificación y sincronización. Asimismo, el cambio de contexto entre hebras del núcleo se realiza rápidamente.

Las hebras del núcleo no son un concepto nuevo. Los procesos del sistema tales como el ladrón de páginas en los núcleos de UNIX tradicionales son funcionalmente equivalentes a las hebras del núcleo.

5.A.3 Procesos ligeros

Un *proceso ligero* es una hebra de usuario soportada en el núcleo. Es una abstracción de alto nivel basada en las hebras del núcleo; puesto que un sistema debe soportar hebras del núcleo antes de poder soportar procesos ligeros. Cada proceso puede tener uno o más procesos ligeros, cada uno soportado por una hebra del núcleo separada (ver Figura 5A.1).

Los procesos ligeros son planificados independientemente y comparten el espacio de direcciones y otros recursos del proceso. Pueden hacer llamadas al sistema y bloquearse en espera de una operación de E/S o por el uso de algún recurso. En un sistema multiprocesador, un proceso puede disfrutar de los beneficios de un verdadero paralelismo, puesto que cada proceso ligero puede ser encaminado para ser ejecutado en un procesador diferente. Hay ventajas significativas incluso en un sistema monoprocesador, puesto que los bloqueos en espera de un recurso o de una operación de E/S, son para un proceso ligero determinado y no para el proceso entero.

Además de la pila del núcleo y el contexto a nivel de registro, un proceso ligero también necesita mantener el contexto a nivel de usuario, que debe ser salvado cuando el proceso ligero es expropiado. Mientras que cada proceso ligero está asociado con una hebra del núcleo, algunas hebras del núcleo pueden ser dedicadas a tareas del sistema y no tener un proceso ligero.

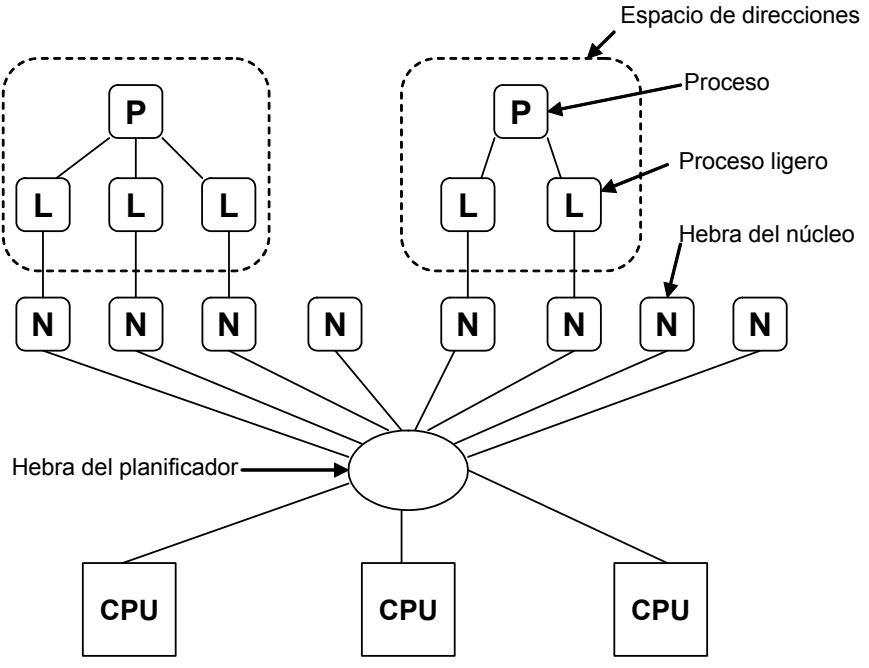


Figura 5A.1: Procesos ligeros

Los procesos multihebras son muy útiles cuando cada hebra es bastante independiente y no interactúa con otras hebras. El código del usuario es completamente expropiable. Todos los procesos ligeros dentro de un proceso comparten un mismo espacio de direcciones. Si cualquier dato puede ser accedido concurrentemente por múltiples proceso ligeros, tales accesos deben estar sincronizados. El núcleo por lo tanto suministra mecanismos de sincronización tales como la exclusión mutua, cerrojos, semáforos y variables de condición para cerrar el acceso a variables compartidas y para bloquear un proceso ligero si intenta acceder a un dato protegido.

Los procesos ligeros tienen algunas limitaciones. La mayoría de las operaciones de los procesos ligeros, tales como creación, destrucción y sincronización, requieren del uso de llamadas al sistema. Cada llamada al sistema requiere de dos cambios de modo, uno de modo usuario a modo núcleo en la invocación y otro de vuelta a modo usuario cuando se completan. En cada cambio de modo, el proceso ligero cruza una frontera de protección. El núcleo debe copiar los parámetros de la llamada al sistema desde el espacio del usuario al espacio del núcleo y validarlos para protegerlos contra procesos

maliciosos. Asimismo, a la vuelta de la llamada al sistema, el núcleo debe copiar los datos de nuevo en el espacio de usuario.

Cuando los procesos ligeros acceden a datos compartidos frecuentemente, la sobrecarga debido a la sincronización puede anular cualquier beneficio que supone el uso de estos procesos.

Cada proceso ligero consume recursos del núcleo de forma significativa, incluyendo memoria física para la pila del núcleo. Por lo tanto un sistema no puede soportar un gran número de procesos ligeros. Además, puesto que el sistema tiene una única implementación de proceso ligero, ésta debe ser lo suficientemente general para soportar la mayoría de las aplicaciones más comunes.

Finalmente, los procesos ligeros deben ser planificados por el núcleo. Aplicaciones que deben transferir a menudo el control de una hebra a otra no pueden hacerlo tan fácilmente usando procesos ligeros.

En resumen, aunque el núcleo suministra los mecanismos para crear, sincronizar y gestionar procesos ligeros, es responsabilidad del programador usarlos juiciosamente. Muchas aplicaciones se pueden atender mejor mediante el uso de hebras a nivel de usuario.

5.A.4 Hebras de usuario

Es posible transferir la abstracción de hebra enteramente al nivel de usuario, sin que el núcleo intervenga para nada. Esto se consigue a través de paquetes de librería tales como *C-threats* de Mach y *pthreads* de POSIX. Estas librerías suministran todas las funciones para crear, sincronizar, planificar y gestionar hebras sin ninguna asistencia especial por parte del núcleo. Las interacciones entre las hebras no involucran al núcleo y por lo tanto son extremadamente rápidas. En la Figura 5A.2 se ilustra esta configuración.

En la Figura 5A.3 se combina el uso de hebras y de procesos ligeros para crear un entorno de programación muy potente. El núcleo reconoce, planifica y gestiona a los procesos ligeros. Una librería a nivel de usuario multiplexa las hebras de usuario encima de los procesos ligeros y suministra los mecanismos para la planificación entre hebras, cambio de contexto y sincronización sin involucrar al núcleo. De hecho, la librería actúa como un núcleo en miniatura para las hebras que controlan.

La implementación de las hebras de usuario es posible porque el contexto a nivel de usuario de una hebra puede ser salvado y restaurado sin la intervención del núcleo. Cada hebra de usuario tiene su propia pila de usuario, un área para salvar el contexto de

registro a nivel de usuario y otras informaciones, como por ejemplo una máscara de señales. La librería planifica y cambia de contexto entre las hebras de usuario, salvando la pila y registros de la hebra actual, después carga los de la nueva planificada.

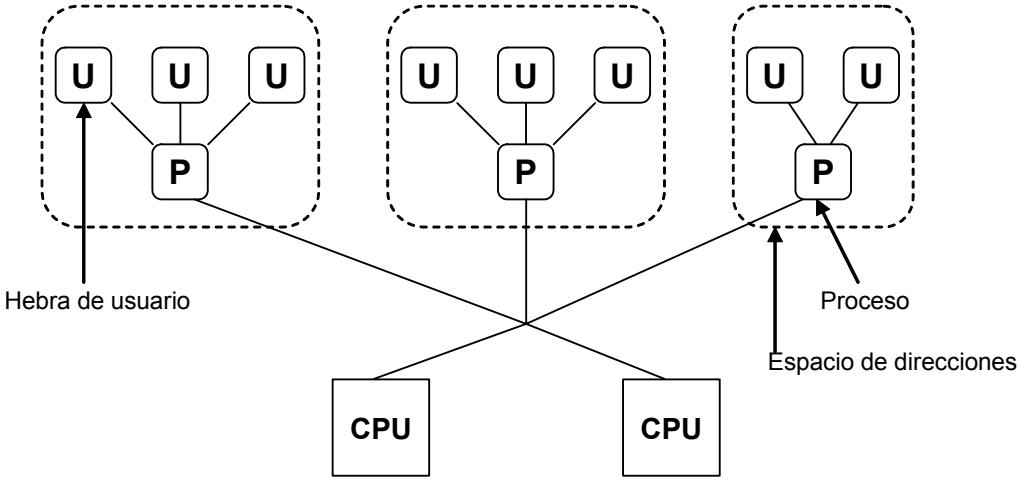


Figura 5A.2: Hebras de usuario encima de los procesos ordinarios

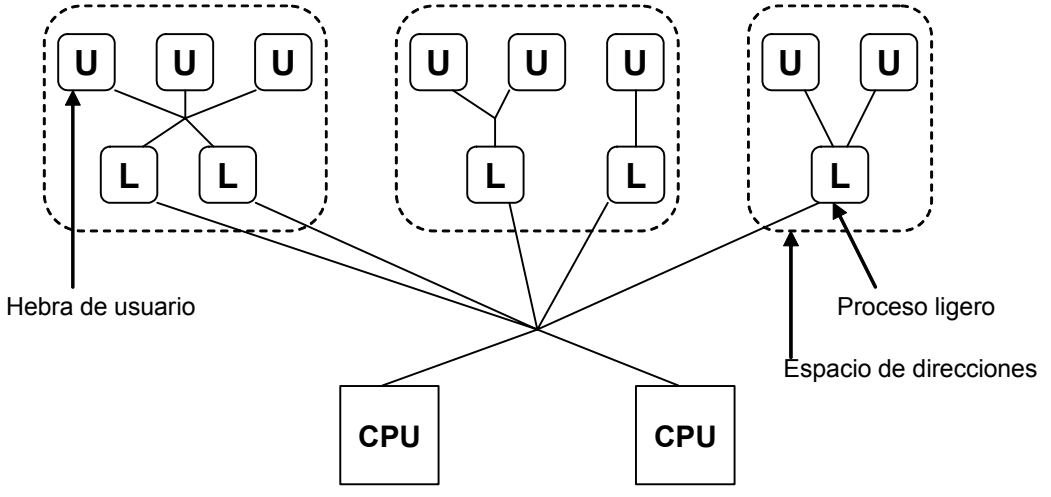


Figura 5A.3: Hebras de usuario multiplexadas en procesos ligeros

El núcleo mantiene la responsabilidad de conmutar entre procesos, ya que sólo él tiene el privilegio de modificar los registros de direcciones de memoria. Las hebras de usuario no son verdaderamente entidades planificables, el núcleo no sabe de su existencia. Éste simplemente planifica al proceso subyacente, es decir, al proceso ligero. Si el proceso tiene un único proceso ligero (o si las hebras de usuario son implementadas en un sistema monohebra), todas sus hebras son bloqueadas.

Las hebras de usuario posee varias ventajas. En primer lugar suministran una forma natural de programar muchas aplicaciones, como por ejemplo todas aquellas gestionadas

mediante ventanas. Además suministran un paradigma de programación síncrona, al ocultar las complejidades de las operaciones asíncronas en la librería de hebras, lo cual las hace muy útiles, incluso aunque el sistema carezca de soporte para ellas. Un sistema puede suministrar varias librerías de hebras, cada una de ellas optimizada para un determinado tipo de aplicaciones.

En segundo lugar, la mayor ventaja de las hebras de usuario es su comportamiento. Son computacionalmente ligeras y no consumen recursos del núcleo excepto cuando se acotan a único proceso ligero. Son capaces de implementar la funcionalidad a nivel de usuario sin utilizar llamadas al sistema. Esto evita la sobrecarga de los cambios de modo. Una noción útil es la de *tamaño crítico de una hebra*, que indica la cantidad de trabajo que una hebra debe realizar para ser considerada útil como entidad separada. Este tamaño depende de la sobrecarga asociada con la creación y uso de una hebra. Para las hebras de usuario, el tamaño crítico es del orden de unas pocas cientos de instrucciones y puede ser reducido a menos de un centenar con el apoyo de un compilador. Las hebras de usuario requieren de mucho menos tiempo para su creación, destrucción y sincronización en comparación con los procesos ligeros y los procesos.

Por otra parte, las hebras de usuario poseen varias limitaciones, principalmente debidas a la separación total de información entre el núcleo y las librerías de hebras. Puesto que el núcleo no sabe de la existencia de las hebras de usuario, no puede usar sus mecanismos de protección para proteger una hebras de otras. Cada proceso tiene su propio espacio de dirección, que el núcleo protege de accesos no autorizados de otros procesos. Las hebras de usuario no disfrutan de esta protección, operan en el espacio de direcciones que es propiedad del proceso. La librería de hebras debe suministrar mecanismos de sincronización, los cuales requieren de la cooperación entre las hebras.

El modelo de planificación dividida produce problemas adicionales. La librería de hebras planifica las hebras de usuario y el núcleo planifica a los procesos subyacentes o procesos. No existe ningún intercambio de información sobre planificación entre ambos. Asimismo, como el núcleo no conoce las prioridades relativas de las hebras de usuario, quizás expropie a un proceso ligero que ejecuta una hebra de usuario de alta prioridad para planificar a un proceso ligero que ejecuta una hebra de usuario de baja prioridad.

Finalmente, sin el apoyo explícito del núcleo, las hebras de usuario pueden mejorar la concurrencia, pero no incrementar el paralelismo. Incluso en un sistema multiprocesador, las hebras de usuario compartiendo un único proceso ligero no pueden ejecutarse en paralelo.

CAPÍTULO

6

Planificación de procesos en UNIX

6.1 INTRODUCCIÓN

La CPU es un recurso de la computadora por cuyo uso compiten los procesos. El sistema operativo debe decidir cómo reparte este recurso entre todos los procesos. El *planificador* es el componente del sistema operativo que determina qué 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 poco 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 los 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 los 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 qué proceso debe ser planificado para ser ejecutado en un cierto instante y cuándo 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.
- Garantizar unos límites para el tiempo de respuesta de las aplicaciones en tiempo real.
- 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 puedan 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 capítulo en primer lugar se describe el tratamiento de las 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. El capítulo finaliza con un par de complementos dedicados a comentar las principales características de los planificadores implementados en las distribuciones SVR4 y Solaris 2.x, respectivamente.

6.2 TRATAMIENTO DE LAS INTERRUPCIONES 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 milisegundos[1].

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:

[1] Éste no es un valor universal y depende de cada variante de UNIX. También depende de la resolución del reloj hardware del sistema.

- 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, 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.

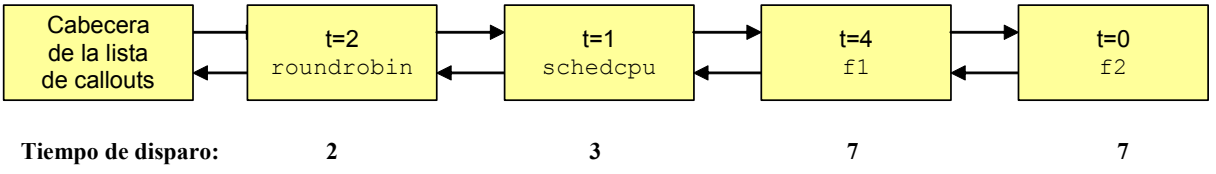
Existen varias formas de implementar la lista de callouts. Un método usado en BSD4.3 es ordenar la lista en función del tiempo que le resta al callout para ser invocado. A este tiempo comúnmente se le denomina *tiempo de disparo*. Cada entrada de la lista de callouts almacena la diferencia entre el tiempo 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.

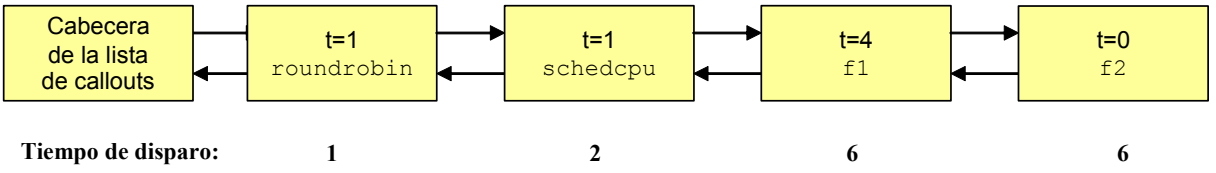
♦ **Ejemplo 6.1:**

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 asociadas 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.



(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 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.

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 tercera 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 cuarta 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 se 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 GMT[2] 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.

[2] GMT es el acrónimo ingles de *Greenwich Mean Time*, es decir, el tiempo medido tomando como referencia el meridiano que pasa por la localidad inglesa de Greenwich.

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`.

♦ **Ejemplo 6.2:**

Considérese el siguiente programa escrito en C:

```
      #include<signal.h>
      void fun1(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");
      };
```

Supóngase que el ejecutable que se crea al compilar este programa se llama `alarma` y que 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 `fun1` 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 que no ignore o que no tenga bloqueada.

Al cabo de 10 segundos 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 y sus descendientes 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:**

Considérese el siguiente programa escrito en C:

```
      #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);
      }
```

Supóngase que el ejecutable que se crea al compilar este programa 6.2 se llama `tiempos` y que 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 BSD4.3[3]. La política de planificación que utiliza este planificador es del tipo *round robin con colas multinivel con realimentación*. Cada proceso tiene asignada una *prioridad de planificación* 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.

[3] El planificador en SVR3 es prácticamente idéntico, difiere únicamente en algunos aspectos menores, tales como el nombre de algunas funciones y variables.

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.
- `p_usrpri`. Contiene la prioridad de planificación actual en modo usuario.
- `p_cpu`. Contiene el tiempo (en tics) de uso 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 qué 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 Tabla 4.1), 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 en modo usuario depende de dos factores: el *factor de amabilidad* (`p_nice`) y el tiempo de uso 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 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:

```
decay=(2*load_average)/(2*load_average+1);        (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;                                    (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);                  (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 una 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 cuánto 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.

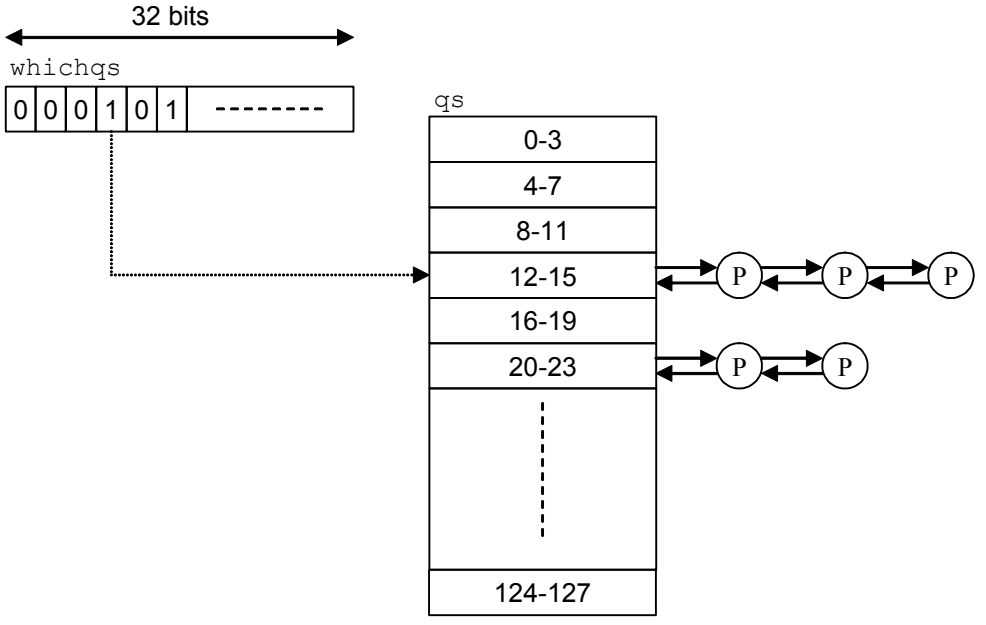


Figura 6.2: Estructuras que usa el planificador en el UNIX BSD4.3

♦

Puesto que tanto BSD4.3, SVR2 y SVR3 tenían a la arquitectura VAX-11 como referencia, 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 denomina `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
```

En 100 ms p_cpu(A)=10 tics y finaliza el cuanto. 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
```

En 200 ms p_cpu(B)=10 tics y finaliza el cuanto. 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)=20 y su prioridad de usuario es p_usrpri(A)= 94 calculada con p_cpu(A)=18.
- En el rango [400, 500] ms se ejecuta el proceso B. Su tiempo de CPU al finalizar el cuanto es p_cpu(B)=20 y su prioridad de usuario es p_usrpri(B)= 94 calculada con p_cpu(B)=18.
- En el rango [500, 600] ms se ejecuta el proceso C. Su tiempo de CPU al finalizar el cuanto es p_cpu(C)=20 y su prioridad de usuario es p_usrpri(C)= 94 calculada con p_cpu(C)=18.
- En el rango [600, 700] ms se ejecuta el proceso A. Su tiempo de CPU al finalizar el cuanto es p_cpu(A)=30 y su prioridad de usuario es p_usrpri(A)= 97 calculada con p_cpu(A)=28.
- En el rango [700, 800] ms se ejecuta el proceso B. Su tiempo de CPU al finalizar el cuanto es p_cpu(B)=30 y su prioridad de usuario es p_usrpri(B)= 97 calculada con p_cpu(B)=28.
- En el rango [800, 900] ms se ejecuta el proceso C. Su tiempo de CPU al finalizar el cuanto es p_cpu(C)=30 y su prioridad de usuario es p_usrpri(C)= 97 calculada con p_cpu(C)=28.
- En el rango [900, 1000] ms se ejecuta el proceso A. Su tiempo de CPU al finalizar el cuanto es p_cpu(A)=40 y su prioridad de usuario es p_usrpri(A)= 98 calculada con p_cpu(A)=38.

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)*40=20
```

```
p_cpu(B)=decay*p_cpu(B)=(1/2)*30=15

p_cpu(C)=decay*p_cpu(C)=(1/2)*30=15
```

Asimismo la rutina schedcpu recalcula la prioridad en modo usuario de todos los procesos usando la fórmula (3)

```
p_usrpri(A)= 50+(20/4)+(2*20)=95

p_usrpri(B)= 50+(15/4)+(2*20)=93

p_usrpri(C)= 50+(15/4)+(2*20)=93
```

Además quita a los tres procesos de la cola de ejecución en la que habían sido colocados al ser creados, 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.

COMPLEMENTO 6.A
Planificador del SVR4

El planificador del SVR4 se rediseñó por completo con objeto de mejorar la planificación tradicional de UNIX. Los principales objetivos de este planificador son:

- Soportar distintos tipos de aplicaciones incluyendo las aplicaciones en tiempo real. Es decir, ser lo suficiente general y versátil para tratar requerimientos de planificación distintos.
- Separar la política de planificación de los mecanismos que la implementan.
- Suministrar aplicaciones con un mayor control sobre sus prioridades y planificación.
- Definir un entorno de planificación con una interfaz bien definida con el núcleo.
- Permitir nuevas políticas de planificación que puedan ser añadidas de una forma modular, incluyendo la carga dinámica de las implementaciones del planificador.
- Limitar el retardo de encaminamiento o distribución para aquellas aplicaciones que deben ser atendidas rápidamente.

La abstracción fundamental que introduce el planificador del SVR4 es la de *clase de planificación*, la cual define la política de planificación para todos los procesos que pertenecen a ella. El sistema puede suministrar diversas clases de planificación. Por defecto, SVR4 suministra dos clases: tiempo compartido y tiempo real.

El planificador dispone de un conjunto de rutinas independientes de la clase que implementan servicios comunes tales como: el cambio de contexto, manipulación de las colas de ejecución y la expropiación. También define una interfaz para funciones dependientes de la clase como las encargadas del cálculo de la prioridad y las encargadas de la herencia. Cada clase implementa estas funciones de forma diferente. Por ejemplo, la clase de tiempo real utiliza prioridades fijas, mientras que la clase de tiempo compartido varía la prioridad del proceso dinámicamente en respuesta a ciertos eventos.

Esta aproximación orientada a objetos es similar a la utilizada en la interfaz nodo-v/sfv (ver sección 8.6). En este caso el planificador representa una clase base abstracta y cada clase de planificación actúa como una subclase.

6.A.1 La capa independiente de la clase

La *capa independiente de la clase* es la responsable del cambio de contexto, gestión de la colas de ejecución y expropiación. El proceso con más alta prioridad siempre se ejecuta (excepto para el procesamiento del núcleo no expropiable). El número de prioridades es de 160. Hay una cola de ejecución para cada prioridad. Por defecto las 160 prioridades se dividen en tres rangos: 0-59 clase de tiempo compartido, 60-99 prioridades del sistema y 100-159 clase de tiempo real. A diferencia de la aproximación tradicional, los valores de prioridad más grandes numéricamente se corresponden con las prioridades más altas. La asignación y cálculo de las prioridades de los procesos son, sin embargo, realizadas por la capa dependiente de la clase.

La Figura 6A.1 describe las estructuras de datos para la gestión de las colas de ejecución. El mapa de bits `dqactmap` muestra que cola de ejecución tiene al menos un proceso listo para ejecución. Los procesos son colocados en la cola mediante `setfrontdq()`y `setbackdq()`. Son eliminados de una cola con `dispdeq()`. Estas funciones pueden ser invocadas desde la línea principal del código del núcleo, así como desde las rutinas dependientes de la clase. Típicamente un proceso nuevo listo para ejecución es colocado al final de una cola de ejecución, mientras que un proceso que fue expropiado antes de que su cuanto expirase es colocado al principio de la cola.

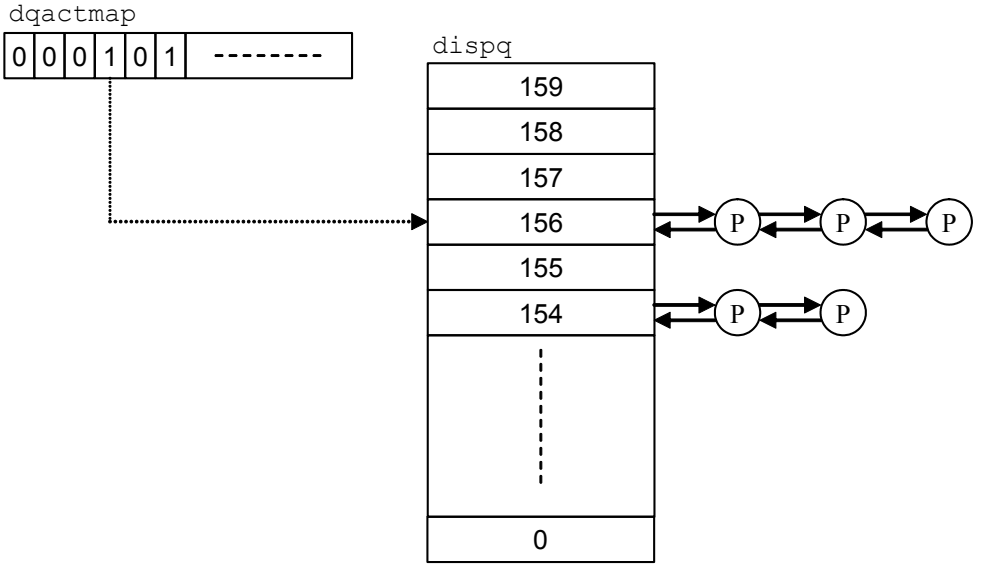


Figura 6A.1: Colas de ejecución del planificador del SVR4

La principal limitación de UNIX para soportar aplicaciones de tiempo real es la naturaleza no expropiativa del núcleo. Los procesos en tiempo real necesitan tener un bajo *retardo de distribución o encaminamiento*, que es el retardo entre el tiempo que pasan a estar listos para ejecución y el tiempo en que realmente comienzan a ejecutarse.

Si un proceso de tiempo real pasa a estar listo para ejecución mientras otro proceso está ejecutando una llamada al sistema, puede existir un retardo considerable antes de que se produzca el cambio de contexto.

Para solucionar este problema, el núcleo de del SVR4 define varios *puntos de expropiación*. Estos son lugares en el código del núcleo donde todas las estructuras de datos del núcleo están en un estado estable y el núcleo está próximo a embarcarse en una computación larga. Cuando se alcanza un punto de expropiación, el núcleo activa un indicador llamado `kprunrun`. Si este indicador esta activado significa que un proceso de tiempo real está listo para ejecutarse y el núcleo expropia la CPU al proceso actual. Esto limita la cantidad de tiempo que un proceso debe esperar antes de ser planificado. La macro `PREEMPT()` comprueba `kprunrun` y llama a la rutina `preempt()` para definitivamente expropiar al proceso. Algunos ejemplos de puntos de expropiación son:

- En la rutina de búsqueda de una ruta de acceso, antes de comenzar a comprobar cada componente individual dentro de la ruta.
- En la llamada al sistema `open`, antes de crear el fichero si éste no existe.
- En el subsistema de memoria, antes de liberar las páginas de un proceso.

El indicador `runrun` es utilizado como en los sistemas tradicionales y solamente expropia a procesos que están cercanos a retornar al modo usuario. La función `preempt()` invoca a la operación `CL_PREEMPT` para realizar el procesamiento dependiente de la clase y después llama a `swtch()` para iniciar el cambio de contexto.

`swtch()` primero llama a `pswtch()` para realizar la parte del cambio de contexto que es independiente de la máquina y después llama a un código de ensamblador de bajo nivel para, entre otras acciones, manipular el contexto a nivel de registro y vaciar los bufers de traducción de direcciones. `pswtch()` desactiva los indicadores `runrun` y `kprunrun`, selecciona el proceso listo para ejecución de mayor prioridad y lo borra de la cola de encaminamiento. Además actualiza `dqactmap` y establece el estado del proceso a `SONPROC` (ejecutándose en el procesador). Finalmente, actualiza los registros de direcciones de memoria para que apunten al área U y a los mapas de direcciones virtuales del nuevo proceso.

6.A.2 Interfaz para las clases de planificación

Toda la funcionalidad dependiente de la clase está suministrada por una interfaz genérica cuyas funciones virtuales son implementadas de forma diferentes para cada clase de planificación. La interfaz define tanto la semántica de las funciones como las conexiones utilizadas para invocar la implementación especifica de cada clase.

La Figura 6A.2 muestra como se implementa la interfaz dependiente de la clase. La estructura `classfuncs` es un vector de punteros a funciones que implementan la interfaz dependiente de la clase para cualquier clase. Una tabla global de clases contiene una entrada por cada clase. Esta entrada está compuesta por el nombre de la clase, un puntero a una función de inicialización y un puntero al vector `classfuncs` de cada clase.

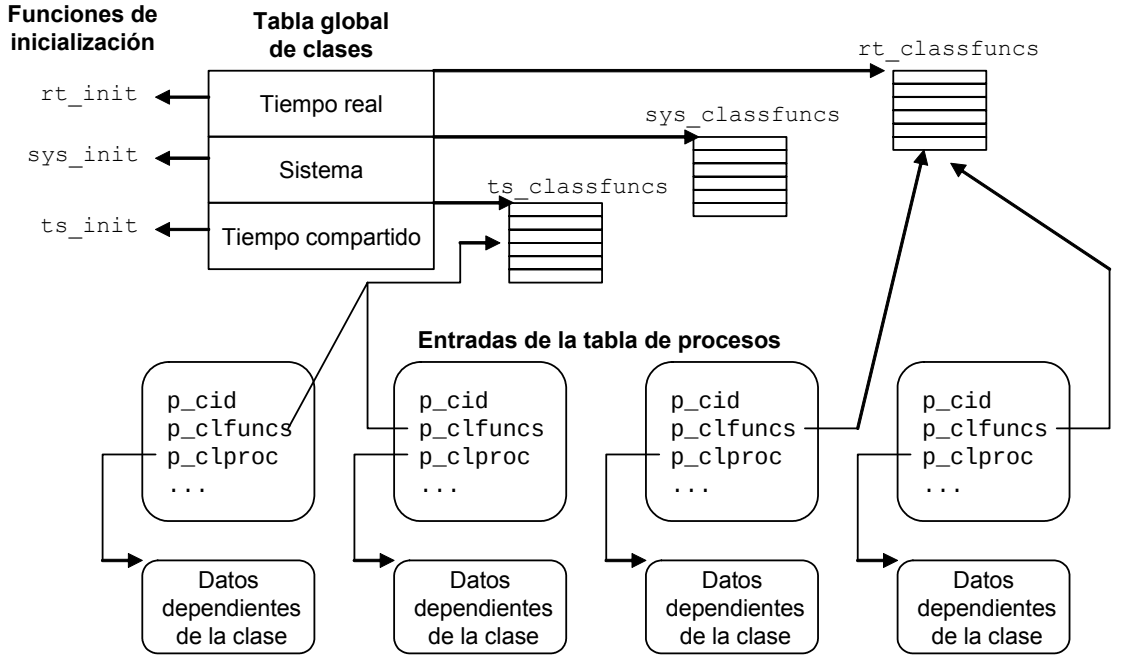


Figura 6A.2: Interfaz dependiente de la clase del planificador del SVR4

Cuando un proceso es creado hereda la clase de prioridad de su padre. Posteriormente, puede ser movido a una clase diferente a través de la llamada al sistema `priocntl`, que suministra diferentes mecanismos para manipular las prioridades y el comportamiento de planificación de un proceso. Una entrada de la tabla de procesos contiene, entre otros, tres campos que son utilizados por las clases de planificación:

- `p_cid`. Es un identificador de la clase que es simplemente un índice dentro de la tabla global de clases.
- `p_clfuncs`. Puntero al vector `classfuncs` para la clase a la cual pertenece el proceso. Este puntero es copiado de la entrada de la tabla de clases.
- `p_clproc`. Puntero a la estructura de datos privada dependiente de la clase.

Un conjunto de macros resuelven las llamadas a las funciones de la interfaz genérica e invocan a la función correcta dependiente de la clase. Las funciones dependientes de la clase pueden ser accedidas de esta manera desde el código independiente de la clase y desde la llamada al sistema `priocntl`.

La clase de planificación decide las políticas para el cálculo de la prioridad y la planificación de los procesos que pertenecen a dicha clase. También determina el rango de prioridades para sus procesos y bajo que condiciones la prioridad del proceso puede cambiar. Asimismo decide el tamaño del cuanto de cada proceso. El cuanto puede ser el mismo para todos los proceso o variar de acuerdo a la prioridad. En general, puede estar comprendido entre un tic e infinito. Un cuanto infinito es apropiado para algunas tareas de tiempo real que deben completar su ejecución rápidamente.

La clase de planificación decide las acciones que cada función debe realizar y cada clase puede implementar estas funciones de forma diferente. Esto permite una aproximación muy versátil para planificar. Por ejemplo, el manipulador de la interrupción de reloj del planificador tradicional carga cada tic al proceso actual y recalcula su prioridad cada cuatro tics. En la nueva arquitectura del SVR4, el manipulador simplemente llama a la rutina `CL_TIOK` de la clase a la cual pertenece el proceso. Esta rutina decide como procesar ese tic del reloj. La clase de tiempo real, por ejemplo, utiliza prioridades fijas y no las recalcula. El código dependiente de la clase determina cuando finaliza el cuanto y activa el indicador `runrun` para iniciar un cambio de contexto.

6.A.3 La clase de tiempo compartido

La *clase de tiempo compartido* es la clase por defecto para un proceso. En ella las prioridades de los procesos se cambian dinámicamente. Se utiliza un algoritmo de planificación de tipo round robin para los procesos con la misma prioridad. Además utiliza una tabla de parámetro de distribución para controlar las prioridades de los procesos y sus cuantos. El cuanto dado a un proceso depende de su prioridad de planificación. La tabla de parámetros define el cuanto para cada prioridad. Por defecto, cuanto menor es la prioridad de un proceso mayor es su cuanto. Esto puede parecer una contradicción pero su explicación es que puesto que los procesos de baja prioridad no se ejecutan muy a menudo es justo darles un cuanto mayor cuando son ejecutados.

La clase de tiempo compartido utiliza planificación conducida por eventos. En vez de recalcular las prioridades de los procesos cada segundo, SVR4 cambia la prioridad de un proceso en respuesta a eventos específicos relacionados al proceso. El planificador penaliza al proceso (reduce su prioridad) cada vez que consume su cuanto. Por otra parte, SVR4 aumenta la prioridad de un proceso si se bloquea (a la espera de que ocurra un evento o esté disponible algún recurso) o si transcurre mucho tiempo hasta que el proceso puede usar su cuanto. Puesto que cada evento normalmente afecta a un único proceso, el recalculo de las prioridades se realiza rápidamente.

La clase de tiempo compartido utiliza la estructura `tsproc` para almacenar los datos dependientes de la clase. Algunos de sus campos son:

- `ts_timeleft`. Contiene el tiempo que resta del cuanto.
- `ts_cpupri`. Contiene la contribución del sistema a la prioridad del proceso
- `ts_upri`. Contiene la contribución del usuario (factor de amabilidad) a la prioridad del proceso
- `ts_umdpri`. Contiene la prioridad en modo usuario..
- `ts_dispwait`. Contiene el número de segundos de la hora del reloj desde que empezó el cuanto.

Cuando un proceso se reanuda después de haber dormido, su prioridad es una prioridad para dormir. Cuando posteriormente retorna a modo usuario, la prioridad es restaurada al valor almacenado en `ts_umdpri`. La prioridad de usuario es la suma de `ts_cpupri` y `ts_upri`, pero está restringida a un valor comprendido entre 0 y 59. `ts_upri` varia entre -20 y 19 y su valor por defecto es 0. Este valor puede ser cambiado con la llamada al sistema `priocntl`, pero sólo el superusuario puede incrementarlo. `ts_cpupri` es ajustado de acuerdo a la *tabla de parámetros de distribución*.

La *tabla de parámetros de distribución* define como diferentes eventos cambian la prioridad de un proceso. Posee una entrada por cada prioridad perteneciente a la clase. Aunque cada clase en SVR4 tiene una tabla de parámetros de distribución, cada tabla tiene una forma diferente. Para la clase de tiempo compartido, cada entrada en la tabla contiene los siguientes campos:

- `ts_globpri`. Prioridad global para esta entrada (coincide para la clase de tiempo compartido con su índice en la tabla).
- `ts_quantum`. Valor del cuanto para esta prioridad.
- `ts_tqexp`. Nuevo `ts_cpupri` a configurar cuando el cuanto expira.
- `ts_slpret`. Nuevo `ts_cpupri` a configurar cuando se retorna a modo usuario después de dormir.
- `ts_maxwait`. Número de segundo a esperar a que el cuanto expire antes de usar `ts_lwait`.
- `ts_lwait`. Usar en vez de `ts_tqexp` si el proceso toma más de `ts_maxwait` en usar su cuanto.

La tabla de parámetros de distribución puede ser indexada por el valor `ts_cpupri` actual para acceder a los campos `ts_tqexp`, `ts_slpret` y `ts_lwait`, puesto que estos campos suministran un nuevo valor de `ts_cpupri` basado en su viejo valor. También es indexada por `ts_umdpri` para acceder a `ts_glopri`, `ts_quantum` y `ts_maxwait`, puesto que estos campos se relacionan con la prioridad de planificación general.

♦ **Ejemplo 6A.1:**

En la Tabla 6A.1 se muestra una tabla de parámetros de distribución de tiempo compartido típica. Para ilustrar como se utiliza supóngase un proceso con `ts_upri`=14 y `ts_cpupri`=1. Su prioridad global (`ts_globpri`) y su `ts_umdpri` son ambas iguales a 15. Cuando su cuanto expira, su `ts_cpupri` será configurada a 0 (puesto que `ts_umdpri` es configurado a 14). Si, no obstante, el proceso necesita más de 5 segundos para consumir su cuanto, su `ts_cpupri` es configurada a 11 (así, `ts_umdpri` es configurado a 25).

Supóngase, que antes de que su cuanto finalice, el proceso hace una llamada al sistema y debe bloquearse en espera de un recurso. Cuando se reanude y en algún momento vuelva a modo usuario, su `ts_cpupri` es configurado a 11 (a partir de la columna `ts_slpret`) y `ts_umdpri` a 25, sin importar cuanto tiempo fue necesario para usar su cuanto.

Índice	ts_globpri	ts_quantum	ts_tqexp	ts_slpret	ts_maxwait	ts_lwait
0	0	100	0	10	5	10
1	1	100	0	11	5	11
...	...	...	...	...	...	...
15	15	80	7	25	5	25
...	...	...	...	...	...	...
40	40	20	30	50	5	50
...	...	...	...	...	...	...
59	59	10	49	59	5	59

Tabla 6A.1 Tabla de parámetros de distribución de la clase de tiempo compartido

♦

6.A.4 La clase de tiempo real

La *clase de tiempo real* utiliza prioridades en el rango 100-159. Estas prioridades son más altas que las de los procesos de tiempo compartido e incluso que las del núcleo, lo que significa que un proceso de tiempo real será planificado antes que cualquier proceso del núcleo. Supóngase que un proceso se está ejecutando en modo núcleo cuando un proceso de tiempo real pasa al estado listo para ejecución. El núcleo no expropia al proceso actual inmediatamente porque podría dejar al sistema en un estado inconsistente. El proceso de tiempo real debe esperar hasta que el proceso actual esté próximo a retornar a modo usuario o se alcance un punto de expropiación del núcleo. Solamente los procesos del superusuario pueden acceder a la clase de tiempo real, mediante la invocación de la llamada al sistema `priocntl`, especificando la prioridad y el valor del cuanto.

Los procesos de tiempo real están caracterizados por una prioridad y cuanto fijos. La única forma en que pueden ser modificados es si el proceso invoca explícitamente a la llamada al sistema `priocntl` para cambiarlos. La tabla de parámetros de distribución de tiempo real es simple, únicamente almacena el valor por defecto del cuanto para cada prioridad, el cual será utilizado excepto si un proceso no especifica un cuanto mientras accede a la clase de tiempo real. Como en el caso de tiempo compartido, aquí también se asignan mayores cuantos para las prioridades más bajas. Los datos dependientes de la clase de un proceso de tiempo real están almacenados en la estructura `rtproc` que incluye el cuanto actual, el tiempo que resta del cuanto y la prioridad actual..

Los procesos de tiempo real necesitan tener acotadas el retardo de distribución y el tiempo de respuesta. El *retardo de distribución o encaminamiento* es el tiempo que pasa desde que un proceso entra en el estado listo para ejecución hasta que comienza a ser ejecutado. Solamente es posible garantizar un valor límite para el retardo de distribución de un cierto proceso de tiempo real si dicho proceso se encuentra en el estado listo para ejecución y además posee la mayor prioridad.

El *tiempo de respuesta* es el tiempo que tarda un proceso en responder a un evento. Es la suma del tiempo requerido por el manipulador de la interrupción para procesar la interrupción, el retardo de distribución y el tiempo empleado en que el código del proceso de tiempo real responda al evento.

6.A.5 Análisis

SVR4 ha sustituido el planificador tradicional por uno completamente diferente tanto en diseño como en comportamiento. Posee una aproximación flexible que permite la adición de clases de planificación al sistema. Las tabla de parámetros de distribución dan más control al administrador del sistema, que puede modificar su comportamiento cambiando la configuración de estas tablas y recompilando el núcleo.

El planificador tradicional de UNIX recalcula la prioridad de cada proceso una vez por segundo. Esto puede emplear una cantidad desproporcionada de tiempo si existen muchos proceso. Por lo tanto el algoritmo no se escala bien para sistemas que tengan millares de procesos. La clase de tiempo compartido del SVR4 cambia la prioridad de un proceso basándose en eventos relacionados con el proceso. Puesto que cada evento normalmente afecta solamente afecta a un proceso, el algoritmo es rápido y altamente escalable.

La planificación basada en eventos favorece deliberadamente a los trabajos limitados por E/S y a los trabajos interactivos frente a los trabajos limitados por la CPU. Esta aproximación tiene algunos inconvenientes importantes. Por ejemplo, los usuarios interactivos cuyos trabajos requieran una gran computación pueden encontrase con que el sistema no responde, puesto que estos procesos pueden no generar suficientes eventos que eleven la prioridad para compensar los efectos del uso de la CPU. Por lo que puede ser necesario resintonizar las prioridades frecuentemente para conseguir que el sistema sea eficiente y receptivo.

Para añadir una clase de planificación el programador debe seguir los siguientes pasos:

1) Suministrar una implementación de cada función de planificación dependiente de la clase.
2) Inicializar un vector `classfuncs` para que apunte a dichas funciones.
3) Suministrar una función de inicialización para realizar tareas de configuración como por ejemplo el alojamiento de las estructuras internas de datos.
4) Añadir una entrada para esta clase en la tabla de clases en un *fichero de configuración maestro*, típicamente localizado en un subdirectorio `master.d` del directorio de construcción del núcleo. Esta entrada contiene punteros a las funciones de inicializiación y el vector `classfunctions`.
5) Reconstruir el núcleo.

Una importante limitación del planificador de SVR4 es que no dispone de una forma adecuada de pasar un proceso de la clase de tiempo compartido a otra clase. La llamada al sistema `priocntl` está restringida al superusuario.

El principal problema con el planificador del SVR4 es que es extremadamente difícil ajustar el sistema adecuadamente para una mezcla de aplicaciones de distintos tipos. Es complicado encontrar una combinación de prioridades y asignación de clases de planificación que permita que todas las aplicaciones progresen adecuadamente.

Con un estudio experimental adecuado, sería posible encontrar la configuración adecuada de prioridades para un conjunto de aplicaciones dado. Pero obviamente esta configuración solamente funcionaría para dicha mezcla especifica de programas. Puesto que la carga de un sistema varia constantemente se tendría que estar continuamente sintonizando manualmente el sistema, lo cual no resultaría útil.

COMPLEMENTO 6.B
Planificador del Solaris 2.x

Solaris 2.x mejoró la arquitectura de planificación básica del SVR4 en varios aspectos. Solaris es un sistema multihebra, así como un sistema operativo multiprocesador simétrico. Por lo tanto su planificador puede soportar estas características. Adicionalmente, Solaris utiliza varios mecanismos de optimización del retardo de distribución de los procesos de alta prioridad que deben ejecutarse en un tiempo límite. El resultado es un planificador que es más adecuado para procesamiento en tiempo real.

El núcleo de Solaris 2.x es completamente expropiable, lo cual permite garantizar buenos tiempos de respuesta. Esto supone un cambio radical con respecto a las anteriores distribuciones de UNIX. Como consecuencia de esta característica la mayoría de las estructuras globales del núcleo deben ser protegidas con mecanismos de sincronización adecuados tales como cerrojos o semáforos.

Otra consecuencia de la expropiabilidad del núcleo es la implementación de las interrupciones mediante hebras especiales del núcleo, que pueden usar primitivas de sincronización estándar del núcleo y bloquear recursos si es necesario. Como consecuencia, Solaris apenas necesita elevar el *npi* para proteger regiones críticas y tiene solamente unos pocos segmentos del código no expropiables. De esta forma los procesos de mayor prioridad pueden ser planificados tan pronto como acceden al estado listo para ejecución.

Solaris mantiene un única cola de ejecución para todos los procesadores. Sin embargo, algunas hebras (como por ejemplo las hebras asociadas a las interrupciones) pueden ser restringidas a ser ejecutados en un determinado procesador. Los procesadores pueden comunicarse unos con otros usando las denominadas como *interrupciones del procesador cruzadas*. Cada procesador tiene el siguiente conjunto de variables de planificación en una estructura de datos por procesador:

- `cpu_thread`. Hebra actualmente ejecutándose en este procesador.
- `cpu_dispthread`. Última hebra seleccionada para ejecutarse en este procesador.
- `cpu_iddle`. Hebra de ocio para este procesador.
- `cpu_runrun`. Indicador de expropiación utilizado por las hebras de tiempo compartido.

- `cpu_krunrun`. Indicador de expropiación utilizado por las hebras de tiempo real.

- `cpu_chosen_level`. Prioridad de la hebra que va a expropiar a la hebra actual en este procesador.

Existen ciertas situaciones donde una hebra de baja prioridad puede bloquear a una hebra de mayor prioridad durante un periodo de tiempo largo. Estas situaciones son causadas o por la *planificación oculta* o por la *inversión de prioridad*.

El núcleo a menudo realiza algunos trabajos asíncronamente en el nombre del proceso. El núcleo planifica este trabajo sin considerar la prioridad de la hebra para la cual está haciendo el trabajo. Ésto es lo que se conoce como *planificación oculta*. Un ejemplo donde se produce planificación oculta es en los callouts. UNIX sirve todos los callouts con el npi más bajo, el cual es todavía más alto que cualquier prioridad de tiempo real. Si el callout fuese dado a una hebra de baja prioridad, su servicio podría retrasar la planificación de una hebra de mayor prioridad. Para solventar este problema, Solaris trata los callouts mediante una *hebra de callout* que se ejecuta a la máxima prioridad del sistema, la cual es menor que cualquier prioridad de tiempo real.

Por otra parte, el problema de la *inversión de prioridad*, se refiere a la situación donde un proceso de baja prioridad retiene un recurso necesitado por un proceso de mayor prioridad, bloqueando por tanto a dicho proceso. Este problema puede ser resuelto usando una técnica conocida como *traspaso de prioridad*. Dicha técnica consiste en que cuando una hebra de alta prioridad se bloquea en espera de un recurso, temporalmente transfiere su prioridad a la hebra de más baja prioridad que posee el recurso, con objeto de que pueda finalizar su ejecución y liberar el recurso.

Solaris necesita mantener un estado extra sobre los objetos bloqueados para implementar el traspaso de prioridad. Necesita identificar que hebra es la propietaria actual de cada objeto bloqueado, así como el objeto por el cual cada hebra bloqueada está esperando. Puesto que el traspaso de prioridad es temporal, el núcleo debe ser capaz de atravesar todos los objetos y hebras bloqueadas en la cadena de sincronización empezando desde cualquier objeto.

En resumen, Solaris 2.x suministra un entorno sofisticado para procesamiento mutihebra y en tiempo real para sistemas monoprocesador o multiprocesador. Resuelve varios inconvenientes de la implementación del planificador del SVR4. Así consigue mantener los retardos de distribución en unos niveles bajos. Esto es debido principalmente a que el núcleo de Solaris 2.x es completamente expropiable y a que implementa la técnica conocida como traspaso de prioridad.

CAPÍTULO 7

Comunicación y sincronización de procesos en UNIX

7.1 INTRODUCCIÓN

UNIX es un sistema operativo multiproceso, es decir, varios procesos pueden estar ejecutándose en el núcleo de UNIX al mismo tiempo. En un sistema con un único procesador solamente un proceso puede estar ejecutándose en la CPU. El sistema rápidamente conmuta de un proceso a otro, generando la ilusión de que todos ellos se ejecutan concurrentemente.

Los procesos deben comunicarse y sincronizarse entre sí para conseguir 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 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.

En consecuencia, el núcleo debe disponer de mecanismos adecuados que implementen la comunicación y sincronización de los procesos.

En este capítulo en primer lugar se describen dos mecanismos de comunicación universales disponibles en todas las versiones de UNIX: las señales y las tuberías. En segundo lugar se describen los mecanismos colectivamente denominados como *mecanismos IPC (InterProcess Comunication) del System V*, es decir, los semáforos, las colas de mensajes y la memoria compartida. En tercer lugar se analizan los mecanismos de sincronización en las distribuciones de UNIX clásicas. El capítulo finaliza con dos complementos, el primero está dedicado al seguimiento de procesos, otro mecanismo de comunicación universal. El segundo complemento describe los mecanismos de sincronización en las distribuciones modernas de UNIX.

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 (ver Complemento 7A).

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 32 señales diferentes (45 en el caso de Solaris). 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 32 tipos de señales distintas.
- 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.

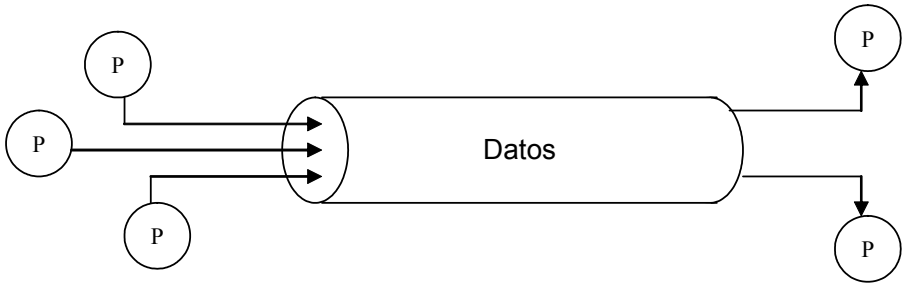


Figura 7.1: Datos fluyendo a través de una tubería

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.

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 y `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. Ésta 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 este 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.

- 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 cuál 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 cuántos mensajes han sido enviados, o dónde 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 asociarle 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>
      #include <string.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);
        }
}
```

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 usando la llamando al sistema `unlink`. Por tanto son útiles para mantener datos que deban sobrevivir a los usuarios activos.

Asimismo, poseen las siguientes desventajas:

- 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 descriptores de la tubería. Un proceso es incapaz de darse cuenta de que el fichero que está leyendo es en realidad una tubería.

Los procesos emisores añaden datos al final de la tubería, 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 provocase que una tubería rebosara, este proceso se bloquearía hasta que se habilitase espacio en la tubería mediante las operaciones de lectura 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.

7.3 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 del 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:

$$N_{IPC} = seq * N_T + I_T \qquad (1)$$

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:

$$I_T = N_{IPC} \bmod(N_T) = N_{IPC} \% N_T \qquad (2)$$

♦ **Ejemplo 7.3:** La tabla asociada a un determinado tipo de mecanismo IPC posee N_T=100 entradas. Calcular I_T 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 mod (100) = 5 % 100 = 5

b) I_T= 30 mod (100) = 30 % 100 = 30

c) I_T= 101 mod (100) = 101 % 100 = 1

d) I_T= 303 mod (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 N_{IPC}= 0*100+1 =1

seq=1 N_{IPC}= 1*100+1 =101

seq=2 N_{IPC}= 2*100+1 =201

seq=3 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 asignara 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 ya debe estar creado 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*/
}
```

En este caso para que `ftok` se ejecute correctamente `archivo1` debe existir en el directorio de trabajo actual.

♦

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 sí, 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 e 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()`[1]. 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.

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

[1] 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 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
}
```

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 (I_T=0 e I_T=1). La primera entrada (I_T=0) contiene información de un conjunto con 4 semáforos cuyo descriptor numérico es `semid`=0, mientras que la segunda entrada (I_T=1) contiene información de un conjunto con 2 semáforos cuyo descriptor numérico es `semid`=1.

Obsérvese cómo 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`.

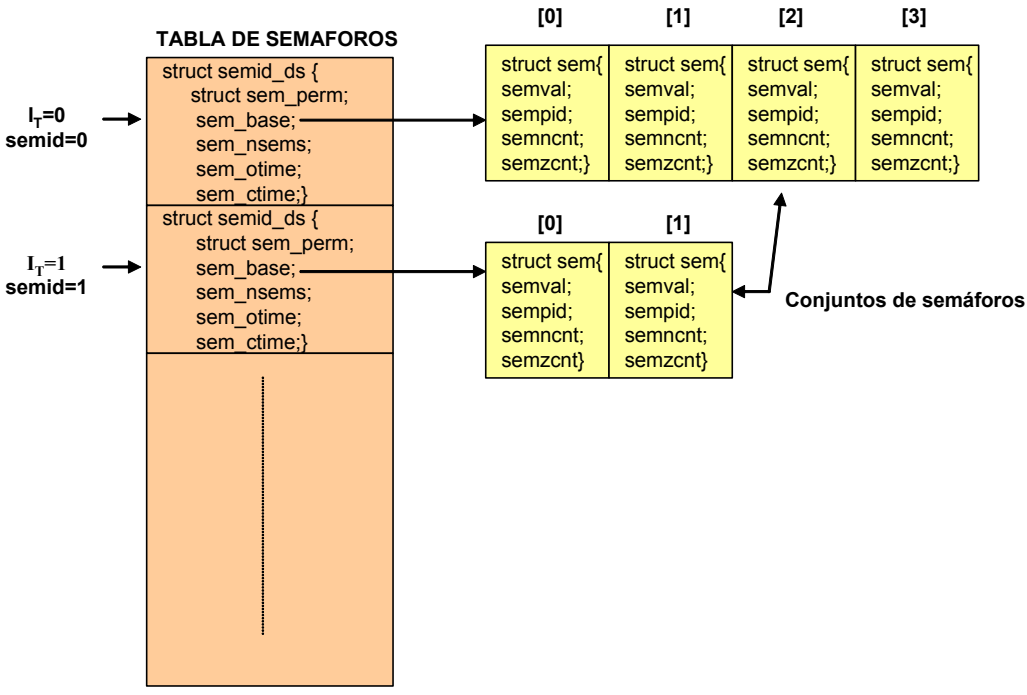


Figura 7.2: Estructura de datos del núcleo necesarias para el manejo de semáforos

♦

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 cómo 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 pueden 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 cómo 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 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
  ushort* 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 `resultado`.
SETVAL	Permite inicializar un semáforo a un valor determinado que se especifica en `arg`.
GETPID	Se usa para leer el *pid* del último proceso que actuó sobre el semáforo. Este número se almacena en `resultado`.
GETNCNT	Permite leer el número de procesos que hay esperando a que se incremente el valor del semáforo. Este número se almacena en `resultado`.
GETZCNT	Permite leer el número de procesos que hay esperando a que el semáforo tome el valor cero. Este número se almacena en `resultado`.
GETALL	Permite leer el valor de todos los semáforos asociados a un identificador `semid`. Estos valores se almacenan en `arg`.
SETALL	Sirve para inicializar el valor de todos los semáforos asociados a un identificador `semid`. Los valores de inicialización deben estar en `arg`.
IPC_STAT, IPC_SET	Permiten leer y modificar la información administrativa asociada al identificador `semid`.
IPC_RMID	Indica al núcleo que debe borrar el conjunto de semáforos agrupados bajo el identificador `semid`. 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 cómo 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 produzcan 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 un área de datos.

Cada entrada de *tabla de colas de mensajes* 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 `msqid` 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 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:**

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 cómo la tabla de cola de mensajes tiene dos entradas activas (I_T=0 e I_T=1). La primera entrada (I_T=0) contiene información de una cola de mensajes cuyo descriptor numérico es `msqid`=0, mientras que la segunda entrada (I_T=1) contiene información de una cola de mensajes cuyo descriptor numérico es `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.

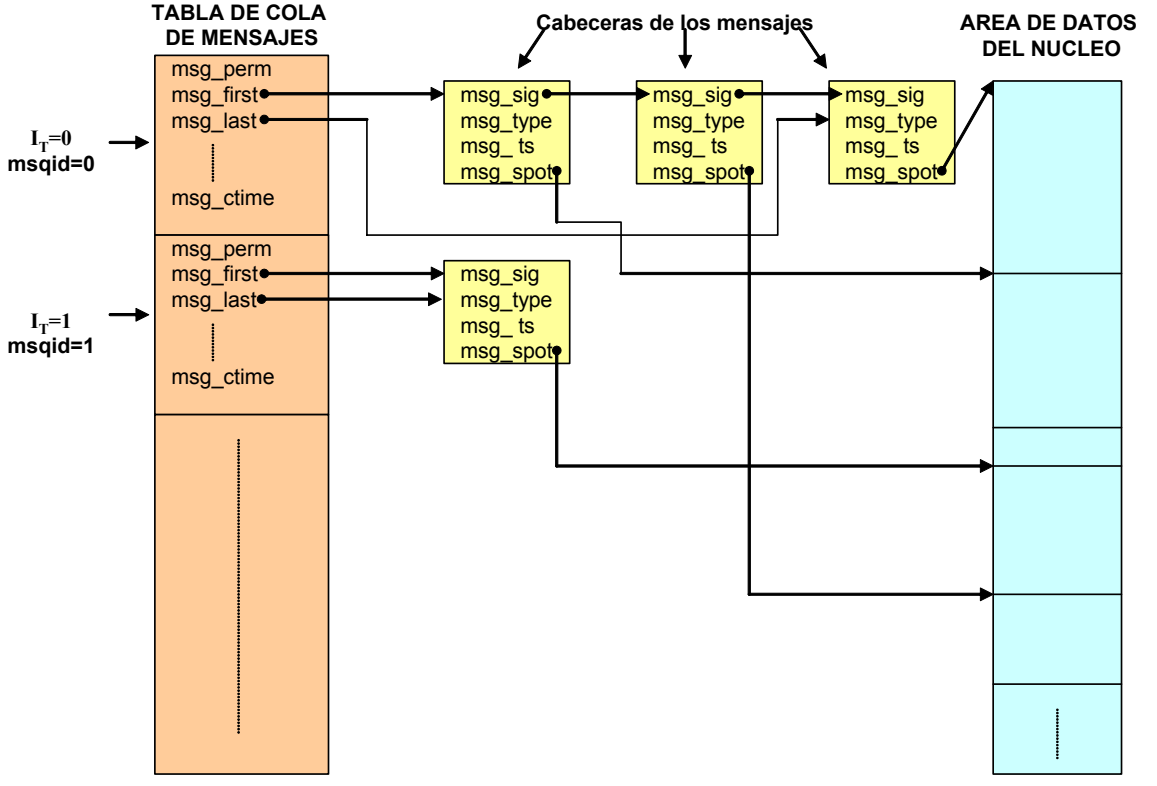


Figura 7.3: Estructuras de datos utilizadas en la implementación del mecanismo IPC de cola de mensajes.

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 cómo 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 descrita 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 cómo 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 si no 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 si no 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 todos 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 `msgtipo` 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, ¿Qué tipo de mensaje extrae el núcleo?*

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 si no se 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 si no se puede leer 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 extraerán únicamente los `msgsz` primeros bytes del mensaje.

♦ **Ejemplo 7.13:**

Las siguientes líneas de código C muestran cómo 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);
...
/*Envio 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 `buffer`
- 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 `buffer`.

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_perm.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 fllujo 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 última 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. Por otra parte, 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 cómo 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.

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.

♦ **Ejemplo 7.16**

En la Figura 7.4a se muestran las estructuras de datos del núcleo utilizadas en la implementación del mecanismo IPC de memoria compartida. Se observa que la tabla de memoria compartida tiene dos entradas activas (I_T=0 e I_T=1). La primera entrada (I_T=0) contiene información de una región de memoria compartida cuyo descriptor numérico es `shmid`=0, mientras que la segunda entrada (I_T=1) contiene información de una región de memoria compartida cuyo descriptor numérico es `shmid`=1. Entre otras informaciones (estructura `shmid_ds`) estas entradas contienen un puntero a la tabla de regiones del núcleo.

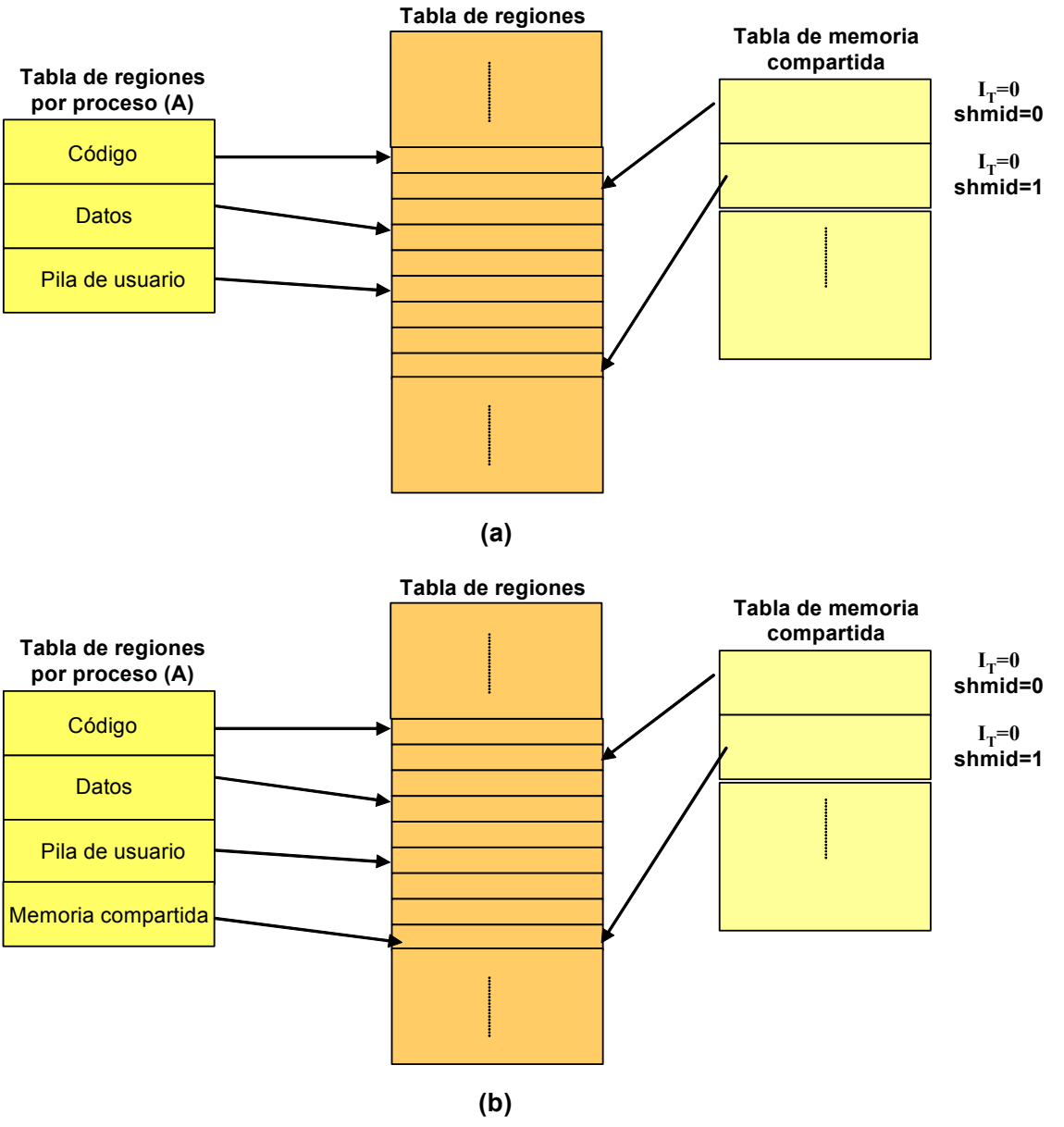


Figura 7.4: Estructuras de datos del núcleo para un proceso (A) que desea acceder a la región de memoria compartida `shmid`*=1: (a) Antes de llamar a* `shmat`*. (b) Después de llamar a* `shmat`

Asimismo en la Figura 7.4a se observa que ninguna de las regiones de memoria compartida está ligada al espacio de direcciones virtuales de ningún proceso. Supóngase que un determinado proceso (A) desea ligar su espacio de direcciones virtuales a la región de memoria compartida `shmid`=1 entonces deberá invocar a la llamada al sistema `shmat`. Las estructuras de datos del núcleo se modificarían en la forma mostrada en la Figura 7.4b. Se crearía una región de memoria compartida en la tabla de regiones por proceso del proceso A que apuntaría a la región de la tabla de regiones asociada a la región de memoria compartida `shmid`=1.

♦

Las reglas que utiliza `shmat` para determinar la dirección son:

- Si `shmdir` = 0, el sistema selecciona la dirección. Es la opción más adecuada si se desea conseguir portabilidad.
- 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`.

En el momento que una región de memoria compartida `shmid` se une a un proceso, ésta 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 `shm_perm` de la memoria compartida y lo devuelve en la zona apuntada por `buffer`.
IPC_SET	Inicializa alguno de los campos de la estructura de control de la memoria compartida `shm_perm`. El nuevo valor para estos campos los toma de la estructura apuntada por `buffer`.
IPC_RMID	Borra del sistema la región de memoria compartida identificada por `shmid`. 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 `shmid`. 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_UNLOK	Desbloquea el segmento de memoria compartida *shmid*, 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 7.17:**

Las siguientes líneas de código C muestran cómo 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);
```

♦

7.4 MECANISMOS DE SINCRONIZACIÓN TRADICIONALES

En las distribuciones clásicas de UNIX existían principalmente tres mecanismos de sincronización: el carácter no expropiable del núcleo, el enmascaramiento o bloqueo de interrupciones y el uso de los indicadores bloqueado y deseado.

7.4.1 Núcleo no expropiable

En UNIX varios procesos pueden estar ejecutándose 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 asegurar la no expropiabilidad del núcleo. 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.

7.4.2 Bloqueo de interrupciones

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 sí 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 mediante la manipulación del *npi*.

Por ejemplo, una rutina del núcleo puede desear eliminar de una cola de buffers a un determinado buffer que almacena una copia de un bloque del disco; esta cola también puede ser accedida por el manipulador de las interrupciones del disco. El código para manipular la cola es una *región crítica*. Antes de acceder a la región crítica, la rutina del núcleo elevará el *npi* para bloquear las interrupciones del disco. Después de completar la manipulación de la cola, la rutina restaurará el *npi* a su valor original, con lo que las interrupciones del disco podrían ser atendidas. De esta forma el *npi* permite la sincronización efectiva de los recursos compartidos por el núcleo y los manipuladores de interrupciones.

7.4.3 Uso de los indicadores bloqueado y deseado

A menudo un proceso desea garantizarse el uso exclusivo de un determinado recurso incluso aunque entre en el estado dormido. Por ejemplo, un proceso A puede desear leer un bloque de datos del disco duro desde 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 B. Si dicho proceso B 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 A se encuentra en el estado dormido.

UNIX asocia dos indicadores, *bloqueado* y *deseado*, a cada recurso compartido. Cuando un proceso A 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 B 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 A 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 comprobara de nuevo el indicador de *bloqueado* y encontrará que está desactivado, entonces lo activará y procederá a usar el recurso.

7.4.4 Limitaciones

Una de las suposiciones básicas en el modelo de sincronización clásico es que un proceso retiene el uso exclusivo del núcleo (excepto por las interrupciones) hasta que voluntariamente deja el núcleo o se bloquea en espera de usar un determinado recurso. Esta suposición ya no es válida en un sistema multiprocesador, puesto que cada procesador podría estar ejecutando código del núcleo al mismo tiempo.

En consecuencia en sistemas multiprocesador es necesario proteger aquellos datos que no necesitaban protección en un sistema con un único procesador. Para ello se tuvieron que usar otros mecanismos de sincronización (ver Complemento 7B) tales como los semáforos, los cerrojos con bucle de espera (spin locks) y las variables de condición. Estos mecanismos de sincronización aunque ideados para sistemas multiprocesador también se pueden utilizar en sistemas con un único procesador.

COMPLEMENTO 7.A
Seguimiento de procesos

Otro mecanismo de comunicación disponible en las primeras distribuciones de UNIX era el conocido como *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.
- Configurar 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 causa esta parada. De esta forma 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.

COMPLEMENTO 7.B
Mecanismos de sincronización modernos

7.B.1 Semáforos

Las primeras distribuciones de UNIX para sistemas multiprocesador implementaban la sincronización casi exclusivamente mediante el uso de semáforos. Los *semáforos* son objetos que pueden tomar valores enteros que soportan dos operaciones atómicas: `P()` y `V()`. 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.

♦ **Ejemplo 7B.1:**

Las siguientes líneas de código C describe una posible implementación de las operaciones `P()` y `V()`. Además se incluye una función de inicialización `initsem()` y una función `CP()` que es una implementación no bloqueante de `P()`.

```
void initsem (semaphore *sem, int val) /*Inicializar semáforo*/
{
      *sem=val;
}
void P(semaphore *sem) /*Decrementar semáforo*/
{
      *sem -= 1;
      while (*sem <0)
            /*Bloquear el proceso*/
}
void V(semaphore *sem) /*Incrementar semáforo*/
```

```
{
      *sem += 1;
      if (*sem >= 0)
            /*Despertar a un proceso bloqueado en el semáforo*/
}
bolean_t CP(semaphore *sem) /*Intenta decrementar
                              el semáforo sin bloquearse*/
{
      if (*sem > 0){
            *sem -= 1;
            return TRUE;
      } else
            return FALSE;
}
```

El núcleo garantiza que las operaciones sobre el semáforo serán atómicas, incluso en sistemas multiprocesador. Así si dos procesos intentan operar sobre el mismo semáforo, una operación se completará o bloqueará antes que la otra comience. Por otra parte, la operación `CP()` permite preguntar por el valor de un semáforo sin bloquearse y es utilizada por manipuladores de interrupciones y otras funciones que no pueden permitirse el permanecer bloqueadas. También se utiliza para evitar el interbloqueo en aquellos casos donde el uso de la operación `P()` podría provocarlo.

♦

Un semáforo se puede utilizar para implementar *exclusión mutua* sobre un determinado recurso. Para ello se debe asociar un semáforo a un recurso compartido e inicializarlo a uno. Cada proceso realiza una operación `P()` para adquirir el uso del recurso en exclusiva y una operación `V()` para liberarlo. La primera operación `P()` que se realice configurará a 0 el valor del semáforo, causando que las siguientes operaciones `P()` bloqueen a los procesos invocadores. Cuando se haga una operación `V()`, el valor del semáforo será incrementado en una unidad y uno de los procesos bloqueados será despertado.

♦ **Ejemplo 7B.2:**

Las siguientes líneas de código C muestran el uso de un semáforo para implementar exclusión mutua.

```
/*Inicialización*/
semaphore sem;
initsem (&sem,1);

/*Código que se debe usar cada vez que se desee usar el recurso*/
```

```
P(&sem);
/*Uso del recurso*/
V(&sem);
```

♦

También es posible utilizar un semáforo para implementar la espera por un determinado evento. En este caso el semáforo se debe inicializar a cero. Los procesos que hagan una operación `P()` se bloquearán. Cuando se produzca el evento se hará una operación `V()` y además cada proceso que se despierte realizará otra operación `V()`.

♦ **Ejemplo 7B.3:**

Las siguientes líneas de código C muestran el uso de un semáforo para implementar la espera por un determinado evento.

```
/*Inicialización*/
semaphore evento;
initsem (&evento,0);

/*Código  ejecutado por el proceso que debe esperar a que
 se produzca un evento*/
P(&evento);          /*Se bloquea si el evento no se ha producido*/
/*Ocurre el evento*/
V(&evento);          /*Para que pueda despertar otro proceso que
                       esté esperando*/

/*Código ejecutado cuando se produce el evento*/
V(&evento);          /*Despierta a un proceso*/
```

♦

Los semáforos también resultan útiles 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. De esta forma el valor del semáforo indica el número de instancias del recurso actualmente disponibles. Si el valor es negativo, entonces su valor absoluto es el número de peticiones pendientes (procesos bloqueados) por ese recurso. Esta es una solución natural al clásico problema de los consumidores-productores.

♦ **Ejemplo 7B.4:**

Las siguientes líneas de código C muestran el uso de un semáforo para administrar un recurso con un número fijo de instancias

```
/*Inicialización*/
semaphore contador;
initsem(&contador,Numero_instancias);

/*Código  ejecutado para usar una instancia de un recurso*/
P(&contador);          /*Se bloquea hasta que existe una instancia
                         disponible*/
/*Uso del recurso*/
V(&contador);          /*Libera la instancia*/
```

♦

Aunque los semáforos suministran una abstracción simple suficientemente flexible para tratar diferentes tipos de problemas de sincronización, poseen algunos inconvenientes que hacen que su uso no sea adecuado en ciertas situaciones. En primer lugar, un semáforo es una abstracción de alto nivel basada en primitivas de bajo nivel que suministran la atomicidad y los mecanismos de bloqueo. Para que las operaciones `V()` y `P()` sean atómicas en un sistema multiprocesador, debe haber una operación atómica para garantizar el acceso exclusivo a la propia variable semáforo.

En segundo lugar el bloqueo y el desbloqueo de procesos implica la realización de cambios de contexto y la manipulación de las colas del planificador y de las colas de procesos dormidos, lo cual hace que sean operaciones lentas. Esto puede ser aceptable para algunos recursos que necesitar ser retenidos durante un periodo de tiempo largo, pero resulta inaceptable si los recursos sólo van a estar retenidos brevemente.

Finalmente, la abstracción de semáforo también oculta información sobre si un proceso ha tenido que bloquearse definitivamente en una operación `P()`. Esto no suele a menudo ser muy importante, pero en algunos casos esto puede ser crucial. Por ejemplo la cache de buffers de bloques de disco de UNIX utiliza una función llamada `getblk()` para buscar un bloque de disco determinado en la caché de buffers. Si el bloque buscado se encuentra en la caché, `getblk()` intenta acceder a él usando una operación `P()`. Si `P()` mandase a dormir al proceso invocador (A) porque el buffer ya estuviese retenido por otro proceso, no existen garantías de que, cuando despierte (A), el buffer de la caché contendrá el mismo bloque que tenía cuando el proceso (A) paso al estado dormido. Con lo que el proceso (A) pasaría a retener un buffer incorrecto. Este problema puede ser resuelto en el marco de los semáforos, pero la solución es complicada e ineficiente.

En general, es deseable tener un conjunto de primitivas de bajo nivel de ejecución económica en vez de una abstracción monolítica de alto nivel. Esta es la tendencia en los núcleos actuales que soportan multiprocesamiento. Entre estos mecanismos de bajo nivel destacan los cerrojos con bucle de espera (spin locks) y las variables de condición.

7.B.2 Cerrojos con bucle de espera

Un *cerrojo con bucle de espera* (*spin locks*)[2] es una primitiva muy simple que permite el acceso en exclusiva a un recurso. Si un recurso está protegido por un cerrojo, un proceso intentando acceder a un recurso no disponible estará ejecutando un bucle, lo que se denomina *espera ocupada*, hasta que el recurso esté disponible. Un cerrojo suele ser una variable escalar que vale 0 si el recurso está disponible y que vale 1 si el recurso no está disponible. Poner a 1 el cerrojo significa "cerrar el cerrojo" y ponerlo a 0 significa "abrir el cerrojo". La variable es manipulada usando un bucle sobre una instrucción atómica del tipo comprobar-configurar.

♦ **Ejemplo 7B.5:**

Las siguientes líneas de código C muestran una implementación de un cerrojo.

```
void spin_lock (spinlock_t *s) /*Función para cerrar el cerrojo*/
{
      while (test_and_set(s)!=0) /*Todavía no disponible*/
      /* Se ejecuta el bucle hasta que esté disponible
}

void spin_unlock (spinlock_t *s) /*Función para abrir el cerrojo*/
{
      *s=0;
}
```

Si un proceso A desea obtener el uso de un recurso invocará a la función `spin_lock` para cerrar el cerrojo. La función `test_and_set(s)` comprueba y devuelve el valor pasado del cerrojo `s`. En el momento que el cerrojo sea abierto por otro proceso B, `test_and_set()` devolverá 0 y el proceso saldrá del bucle. Además `test_and_set()` pone el cerrojo a 1 para asegurar al proceso A el uso exclusivo del recurso. Cuando proceso termina de usar un recurso, debe invocar a la función `spin_unlock` para abrir el cerrojo.

♦

La característica más importante de un cerrojo es que un proceso retiene una CPU mientras espera a que el cerrojo sea abierto. Es por lo tanto esencial que un cerrojo

[2] Por comodidad, en lo que resta de sección se escribirá únicamente *cerrojo* en vez de *cerrojo con bucle de espera*

permanezca cerrado durante periodos de tiempo muy pequeños. En particular, no debe estar cerrados entre operaciones de bloqueo. Asimismo también es deseable bloquear las interrupciones antes de cerrar un cerrojo, para así garantizar que el tiempo de cierre será pequeño.

La premisa básica de un cerrojo es que un proceso realiza una espera ocupada en un procesador mientras que otro proceso está usando el recurso en un procesador diferente. Obviamente esto sólo es posible en sistemas multiprocesador. En un sistema con un único procesador, si el proceso intenta cerrar un cerrojo que ya está cerrado, entonces permanecerá en un bucle infinito. Los algoritmos para sistemas multiprocesador, no obstante, deben operar adecuadamente independientemente del número de procesadores, lo que significa que también deben funcionar adecuadamente en un sistema monoprocesador. En el caso de los cerrojos, esto requiere el cumplimiento estricto de la siguiente regla: un proceso nunca le cede el uso de la CPU mientras tiene cerrado un cerrojo. De esta forma, se asegura en el caso monoprocesador que una hebra nunca tendrá una espera ocupada en un cerrojo.

La mayor ventaja de los cerrojos es que son muy económicos en cuanto a tiempo de ejecución. Cuando no hay disputa por un cerrojo, tanto la operación de cierre como la de apertura típicamente requieren únicamente una instrucción cada una. Resultan ideales para estructuras de datos de uso exclusivo que necesitan ser accedidas rápidamente, como por ejemplo la eliminación de un elemento en una lista doblemente enlazada o mientras se realiza una operación del tipo carga-modificación-almacenamiento de una variable. Por tanto, los cerrojos son utilizados para proteger aquellas estructuras de datos que no necesitaban de protección en un sistema monoprocesador. Los semáforos utilizan un cerrojo para garantizar la atomicidad de sus operaciones.

7.B.3 Variables de condición

Una *variable de condición* es un mecanismo más complejo asociado con un *predicado* (una expresión lógica para evaluar si es verdadera o falsa) basado en algún dato compartido. Permite a un proceso bloquearse en función de su valor y suministra los servicios para despertar a uno o todos los procesos bloqueados cuando el resultado del predicado cambia. En general, resulta más útil para implementar la espera por un evento que para asegurar el acceso exclusivo a un recurso.

Supóngase, por ejemplo, uno o más procesos de un servidor que están esperando por la llegada de peticiones de clientes. Las peticiones entrantes son pasadas a los procesos que esperan o colocadas en una cola si no hay ningún proceso listo para

atenderlas. Cuando un proceso del servidor está listo para atender la siguiente petición, primero comprueba la cola. Si hay una petición pendiente, el proceso la elimina de la cola y la atiende. Si la cola está vacía, el proceso se bloquea hasta que llega una petición. Este escenario de funcionamiento se puede implementar asociando una variable de condición con la cola. El dato compartido es la propia cola de peticiones y el predicado es que la cola no este vacía.

Puede suceder que una petición llegue después de comprobar la cola pero antes de bloquear el proceso. El proceso se bloqueará aunque exista una petición pendiente. En consecuencia se requiere una operación atómica para comprobar el predicado y bloquear al proceso si fuese necesario.

Las variables de condición suministran esta atomicidad usando un cerrojo. El cerrojo protege el dato compartido y evita el problema de no atención de peticiones anteriormente comentado. Se implementa para cada variable de condición una función llamada `wait()` que recibe el cerrojo como argumento y atómicamente bloquea al proceso y abre el cerrojo. Cuando se produce el evento `wait()` despierta al proceso y vuelve a cerrar el cerrojo antes de retornar.

En el caso del ejemplo, el proceso del servidor cierra el cerrojo sobre la cola de peticiones, entonces comprueba si la cola está vacía. En caso afirmativo, llama a la función `wait()` de la variable de condición con el cerrojo cerrado para bloquearse y abrir el cerrojo. Cuando una petición llegue a la cola el proceso será despertado, la función `wait()` vuelve a cerrar el cerrojo antes de retornar.

CAPÍTULO 8

Sistemas de archivos en UNIX

8.1 INTRODUCCIÓN

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 del System V* (*System V file system* (*s5fs)*). Con la distribución BSD4.2 se introdujo un nuevo sistema de ficheros denominado *sistema de ficheros rápido* (*Fast File System* (FFS)) 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 modernos usan técnicas sofisticadas que suministran un mejor comportamiento, 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 *distribuido*s permiten a un usuario acceder a ficheros que residen en máquinas remotas. Ejemplos de 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 una interfaz en el subsistema de archivos de UNIX que permitiera soportar múltiples tipos de sistemas de ficheros: UNIX y no-UNIX, locales y distribuidos. Este objetivo se consiguió con la *interfaz nodo-v/sfv* desarrollado por Sun Microsystems que introdujo los conceptos de *nodo virtual* (*nodo-v*) y *sistema de ficheros virtual* (*sfv*).

El presente capítulo consta de cuatro partes claramente diferenciadas, en la primera se estudian los ficheros especiales, el montaje de sistemas de ficheros, los enlaces simbólicos y la caché de buffers de bloques. En la segunda parte se describe la interfaz nodo-v/sfv del SVR4. La tercera parte se dedica al estudio, debido a su importancia histórica y a su sencillo diseño, del sistema de ficheros *s5fs* implementado en SVR4. El capitulo concluye con tres complementos. El primero está dedicado a la comprobación del estado de un sistema de ficheros. El segundo complemento incluye una serie de consideraciones adicionales sobre la interfaz nodo-v/sfv. Finalmente, el tercer complemento describe el sistema de ficheros FFS.

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/men` (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 opera sobre los ficheros ordinarios también manipula 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 qué 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, dónde 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 constituye 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[1]) 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 una interfaz abstracto. Cada dispositivo lo implementa a través de funciones específicas, por ejemplo, `lpread()` para una impresora en línea o `ttread()` para un terminal.
7) De `cdevsw`, obtiene el puntero a la rutina que implementa la operación de lectura (`d_read`) para este dispositivo.
8) Invoca a la rutina `d_read` para realizar la operación de lectura sobre el dispositivo.
9) Desbloquea el nodo-im y devuelve el resultado al usuario.

[1] Esta nomenclatura es exclusiva de este libro.

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 es independiente del tipo de fichero. Los pasos 5 al 7 representan la 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 qué 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 sistema 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 pueden 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:**

La Figura 8.1 muestra un árbol de ficheros compuesto de dos sistemas de ficheros. En este ejemplo, `fs0` está instalado como 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 ocultados 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.

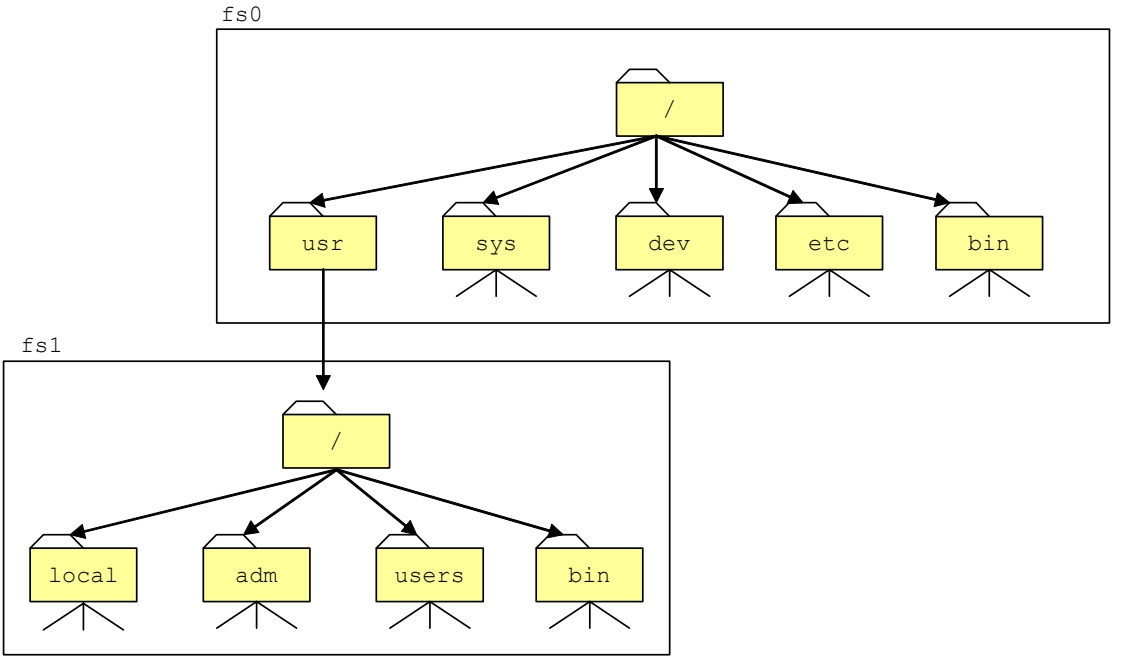


Figura 8.1: Montaje de un sistema de ficheros en otro

♦

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 sí es necesario especificar en otros sistemas operativos). Asimismo, cada sistema de ficheros montado puede 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 bien al montar el sistema usando las opciones adecuados del comando de montaje `mount` o bien a posteriori usando el comando `chmod` para cambiar los permisos de acceso del directorio de montaje.

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         /dosc           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`).

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 `/dosc`, 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 una 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 una 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 la 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 el 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 de la 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 un 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.

También es posible invocar a mount con un único argumento:

```
mount arg
```

donde `arg` puede ser un dispositivo o un punto de montaje. En este caso se busca en la tabla de montaje por si ya existiera una entrada que tenga un campo con el valor de `arg`.

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“. Conviene saber que un sistema de archivos montado de esta forma usualmente no tiene permiso de escritura para otros usuarios.

Con la orden

```
# mount /dev/fd0
```

o con la orden

```
# mount /mnt/floppy
```

se buscaría en la tabla de montaje por si ya está definido allí en que punto de montaje se monta el dispositivo `/dev/fd0` o que dispositivo hay que montar en el punto de montaje `/mnt/floppy`. A diferencia del caso anterior, un sistema de archivos montado de esta forma sí dejaría escribir en él a otros usuarios siempre que las opciones de montaje recogidas en la tabla de montaje así lo permitan.

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 fisico `/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. En primer lugar 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, continúa existiendo debido al otro enlace.

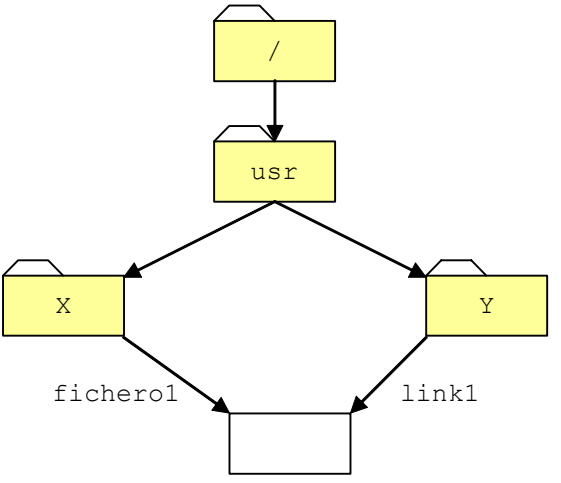


Figura 8.2: Enlaces duros (`fichero1` y `link1`) de un fichero.

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 continúa 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 enlace simbólicos son muy útiles porque no tienen las limitaciones asociadas a los enlaces duros. 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
```

La 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 poco 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, vía 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 *caché de buffers* es típicamente el 10% de la memoria principal. La *caché 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 pueden configurar el número de colas de dispersión de la caché de buffers.

♦ **Ejemplo 8.7:**

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 contiene, obsérvese que todos estos números cumplen la regla

Nº 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.

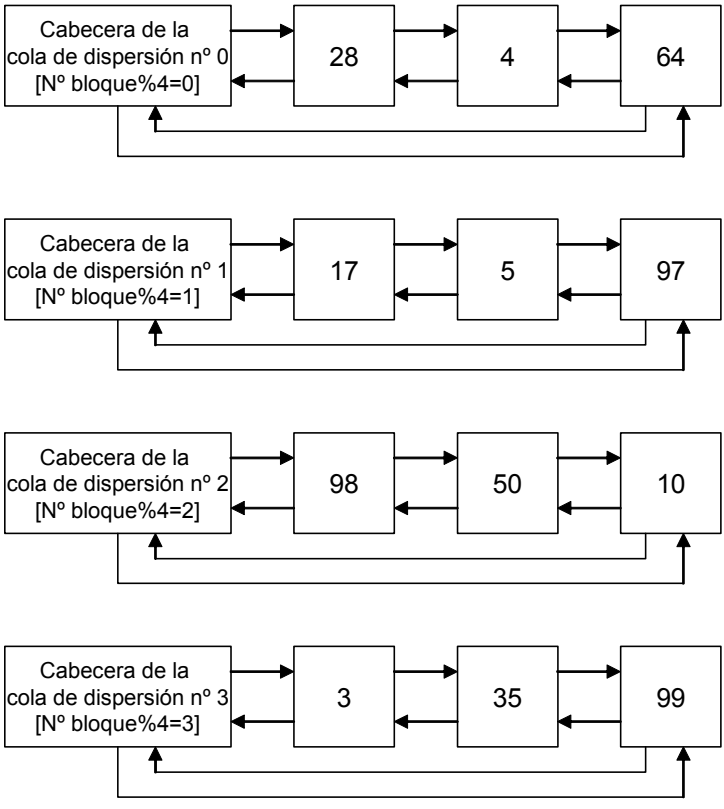


Figura 8.3: Caché de buffers

♦

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 sobreescrito). 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 LRU[2]. 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:**

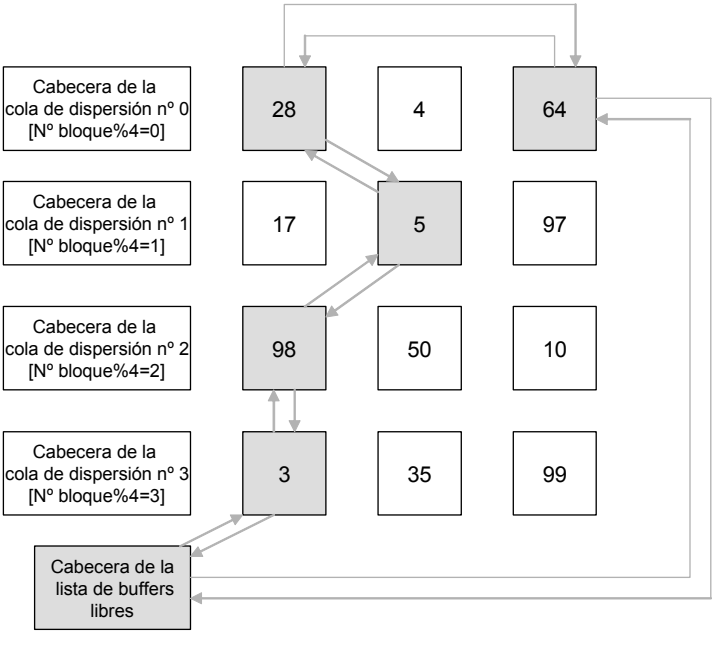


Figura 8.4: Implementación de la lista de buffers libres dentro de la caché de buffers

[2] LRU es el acrónimo del término inglés "*Least Recently Used*", que significa "usado menos recientemente".

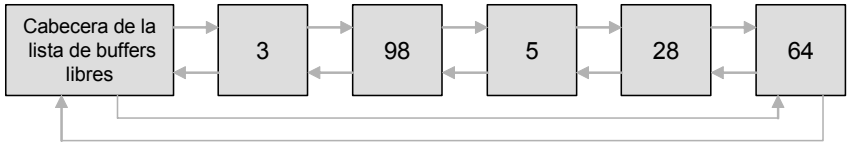


Figura 8.5: Detalle de la lista de buffers libres

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.

♦

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.

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 otros indicadores, como por ejemplo: B_READ, B_WRITE, B_ASYNC, B_DONE y B_ERROR.

Campos	Descripción
`int b_flags`	Indicadores del estado del buffer
`struct buf *b_forw, *b_back`	Punteros para mantener el buffer en la cola hash
`struct buf *av_forw, *av_back`	Punteros para mantener el buffer en la lista de buffers libres
`cadrr_t b_addr`	Puntero al bloque de datos del buffer
`dev_t b_edev`	Número de dispositivo
`daddr_t b_blkno`	Número de bloque en el dispositivo
`int b_error`	Estado de error E/S
`unsigned b_resid`	Número de bytes que restan por transferir

Tabla 8.1: Campos de la estructura `buf`

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 una interfaz modular entre el driver del disco y el resto del núcleo. Ninguna otra parte del núcleo puede acceder al driver del disco y la interfaz entera está encapsulada 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 órdenes 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, ralentizando 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 cómo 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 LA INTERFAZ NODO-V/SFV

Sun Microsystems introdujo *la 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. Desde su aparición ha ido ganando una amplia aceptación, SVR4 fue la primera distribución del UNIX System V que incluyó esta interfaz.

La *interfaz nodo-v/sfv* permite al sistema UNIX:

- Soportar diferentes tipos de sistemas de ficheros locales simultáneamente, tanto UNIX (*s5fs* o *ufs*[3]) y no-UNIX (DOS, A/UX, etc).
- Soportar sistemas 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, la 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 la interfaz nodo-v/sfv asociada a esta llamada al sistema que realiza un primer conjunto de operaciones

[3] *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.

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.

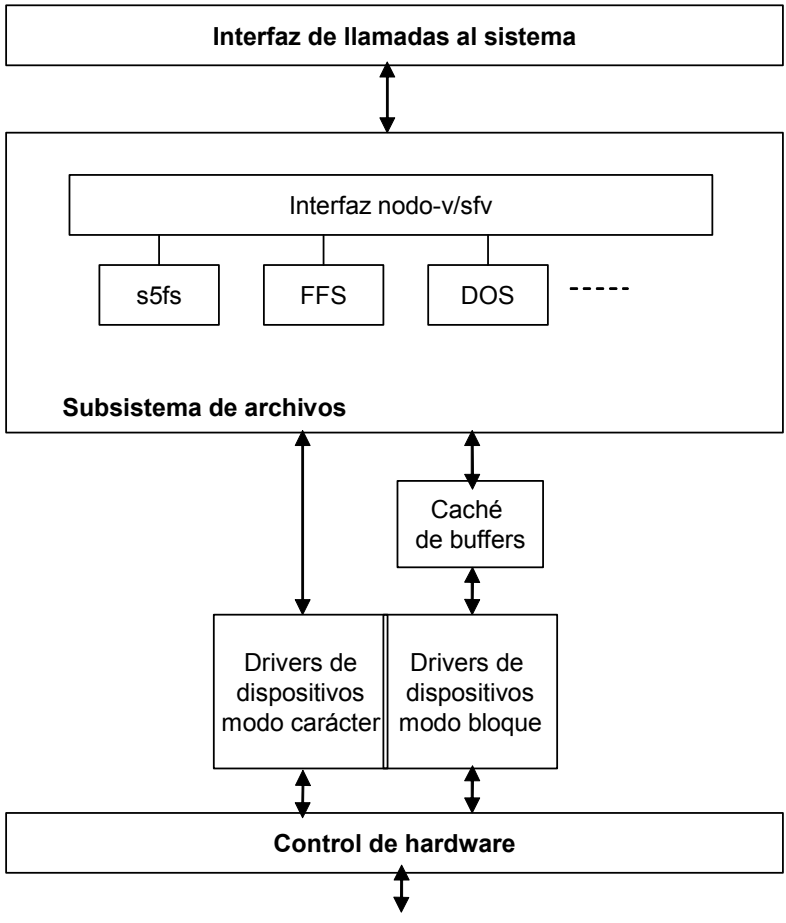


Figura 8.6: Ubicación de la interfaz nodo-v/sfv dentro del núcleo

8.6.1 Una breve introducción a la programación orientada a objetos

La 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 *público* o *privado*. 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 sí 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.

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 una 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.

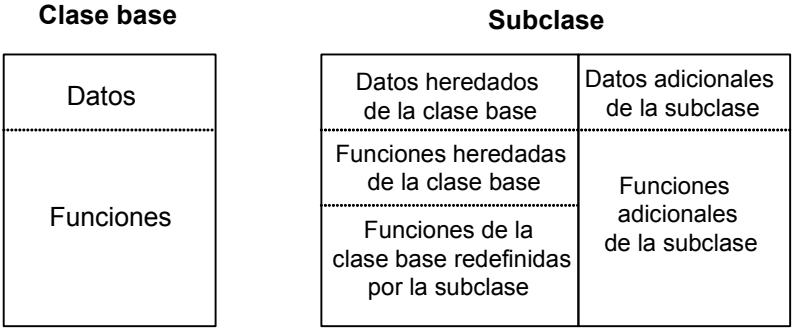


Figura 8.7: Relación entre una clase base y su subclase

Frecuentemente, se usa una clase base simplemente para representar una abstracción y definir una 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 diferente 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 qué subclase pertenece. Cuando una función virtual es invocada por tal objeto, la implementación automáticamente determina a qué función específica debe llamar, dependiendo del subtipo actual del objeto.

8.6.2 Perspectiva general de la interfaz nodo-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 en 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 depende 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 la 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 vuelta 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). 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.

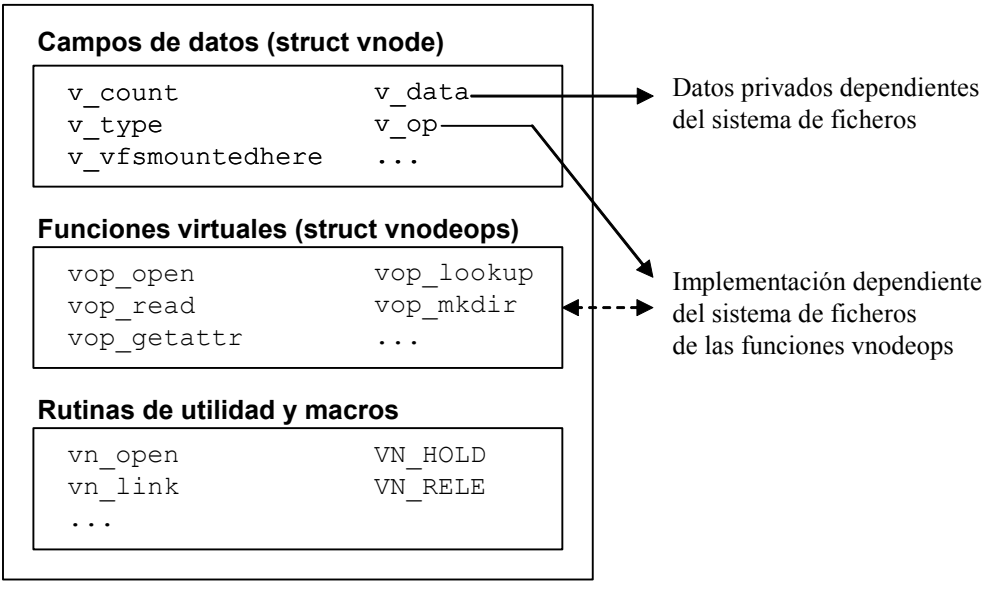


Figura 8.8: La abstracción nodo-v.

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 el 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 datos:

```
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` que 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.

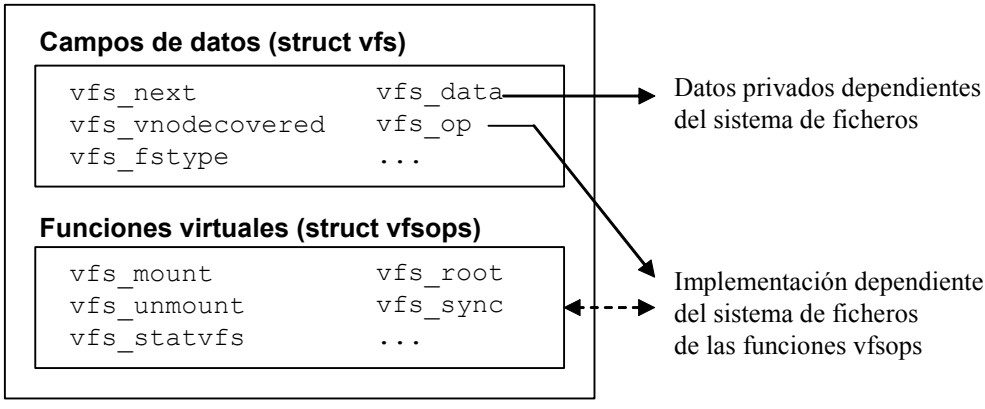


Figura 8.9: La abstracción sfv

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 vfs_fstype;                     /*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.

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.

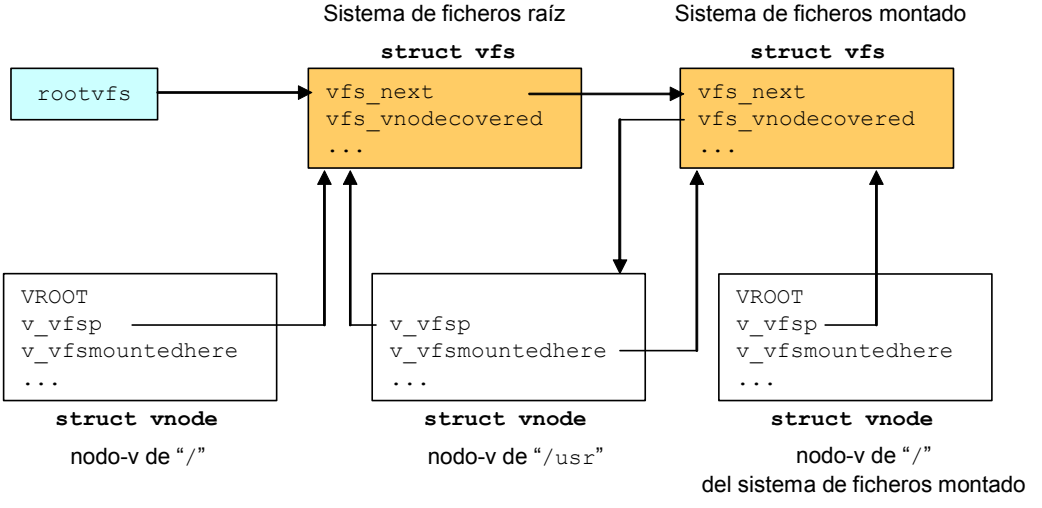


Figura 8.10: Relaciones entre los objetos nodo-v y sfv

♦

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 la 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 (ver sección 8.B.6), que utilizan las estructuras de datos dependientes del sistema de ficheros para localizar el nodo-v.

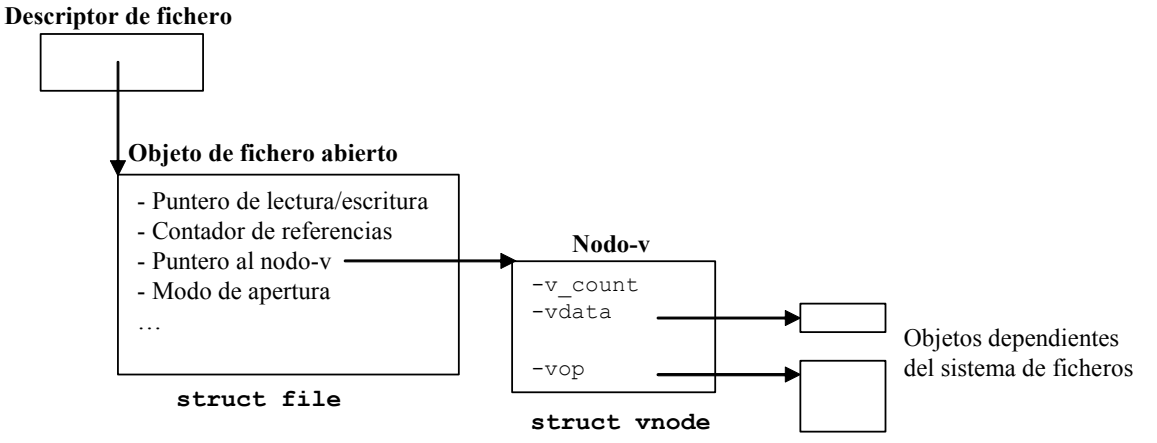


Figura 8.11: Estructuras independientes del sistema de ficheros

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.

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 FCLOSEXEC, que pide al núcleo que cierre el descriptor cuando el proceso invoca a la llamada al sistema `exec` y U_FDLOCK 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, él 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 32 entradas normalmente 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 `kmen_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 cuánto 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 multiples 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 sí mismo continúa 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.

8.7 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 8.12 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 numero 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.

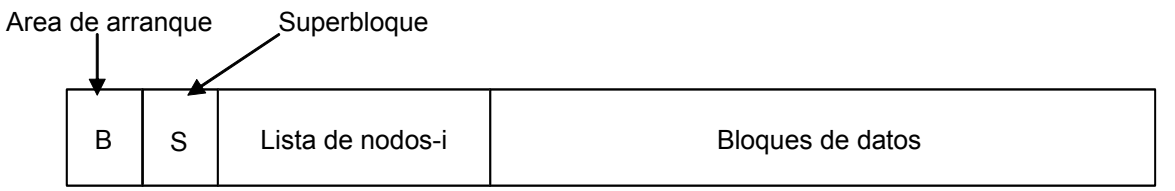


Figura 8.12: Estructura en el disco del s5fs

Al comienzo de la partición se encuentra el *área de arranque*, que puede contener el código requerido para arrancar (carga e inicialización) el sistema operativo. De todas las particiones existentes solamente una de ellas necesita contener esta información, posiblemente el resto de particiones tendrá 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 un 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 los 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 significa 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. La primera entrada corresponde al propio directorio ('`.`') su número de nodo-i es 73. La segunda entrada corresponde al directorio padre "`..`" cuyo número de nodo-i es 38. La tercera entrada corresponde al fichero `adl` cuyo número de nodo-i es 9. La cuarta entrada corresponde a un fichero que ha sido borrado ya que su número de nodo-i es 0. La última entrada corresponde al fichero `prueba` cuyo número de nodo-i es 65. Nótese que no es posible únicamente con esta información saber si `adl` y `prueba` son ficheros ordinarios o directorios. Para averiguarlo habría que examinar sus nodos-i.

Número de nodo-i	Nombre del fichero
73	.
38	..
9	adl
0	(fichero borrado)
65	prueba

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 de aquí en adelante 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	*uid*
`di_gid`	2	*gid*
`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 nodo-i pueden cambiar sin que necesariamente se hayan modificado los contenidos del fichero.

♦ **Ejemplo 8.11:**

En la Figura 8.13 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.

[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 8.13: Ejemplo del contenido de un nodo-i

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.

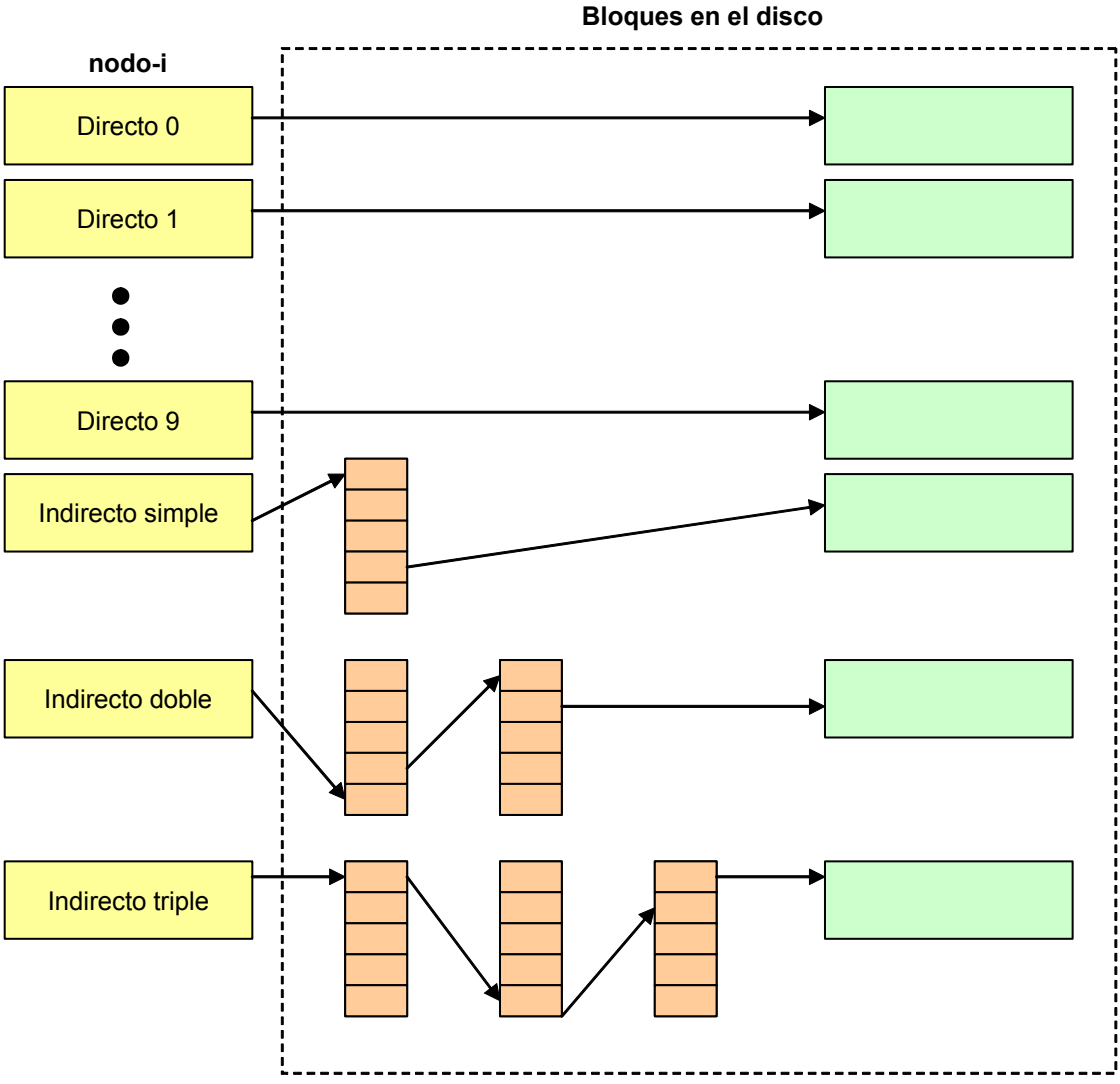


Figura 8.14: Lista de direcciones de bloques físicos almacenada en el campo `di_addr` *del nodo-i.*

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 8.14), 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, quíntuples, etc., pero se tiene más que suficiente con una indirección triple.

♦ **Ejemplo 8.12:**

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 → $N_D = 2^8 = 256$ 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 → 2^8 bloques indirectos simples → $(N_D)^2 = (2^8)^2 = 2^{16} = 65536$ bloques de datos	$(N_D)^2 \cdot S_B = 2^{16} \cdot 2^{10}$ bytes $= 2^6 \cdot 2^{20} =$ 64 Mbytes
1 entrada indirecta triple	1 bloque indirecto triple → 2^8 bloques indirectos dobles → $(2^8)^2$ bloques indirectos simples → $(N_D)^3 = (2^8)^3 = 2^{24} = 16777216$ bloques de datos	$(N_D)^3 \cdot S_B = 2^{24} \cdot 2^{10}$ bytes $= 2^4 \cdot 2^{30} =$ 16 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= 1Kbyte

Supóngase que se dispone de un sistema de archivos con bloques de capacidad S_B=1Kbyte y que un bloque se direcciona con n=32 bits. El espacio de direcciones de bloques es $N=2^{32}=2^2 \cdot 2^{30}=4$ Gdirecciones. Por otra parte, el número de direcciones N_D que puede contener un bloque es:

$$N_D = \frac{S_B}{n}$$

Sustituyendo valores se obtiene:

$$N_D = \frac{1(Kbyte/bloque)}{32(bits/direccion)} = \frac{2^{10} \cdot 2^3 (bits/bloque)}{2^5 (bits/direccion)} = 256\ \frac{direcciones}{bloque}$$

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= 1Kbyte.

♦

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 8.15. Se supone también que un bloque tiene un tamaño de* S_B*= 1Kbytes. 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* E_I *de la lista de direcciones del nodo-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*.*

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 $\boldsymbol{B_L=8}$.

b) El bloque lógico B_L=8 se corresponde con la entrada $\boldsymbol{E_I=8}$ (recuérdese que las entradas se comienzan a numerar con el 0) de la tabla de direcciones del nodo-i. Dentro de la entrada $\boldsymbol{E_I=8}$, 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 8.15, es el bloque del disco $\boldsymbol{B_D=412}$.

c) Para calcular el desplazamiento D_D (los bytes del bloque en el disco se numeran desde 0 a 1023) dentro del bloque B_D= 412 del disco que se corresponde con D_L, se realiza el siguiente cálculo:

$$D_D = D_L \bmod(S_B) = 9125 \bmod(1024) = 933\, bytes$$

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_L= 9125 alojado en el bloque B_L= 8 del fichero se corresponde con el byte alojado en la posición **D_D= 933** del bloque B_D= 412 del disco.

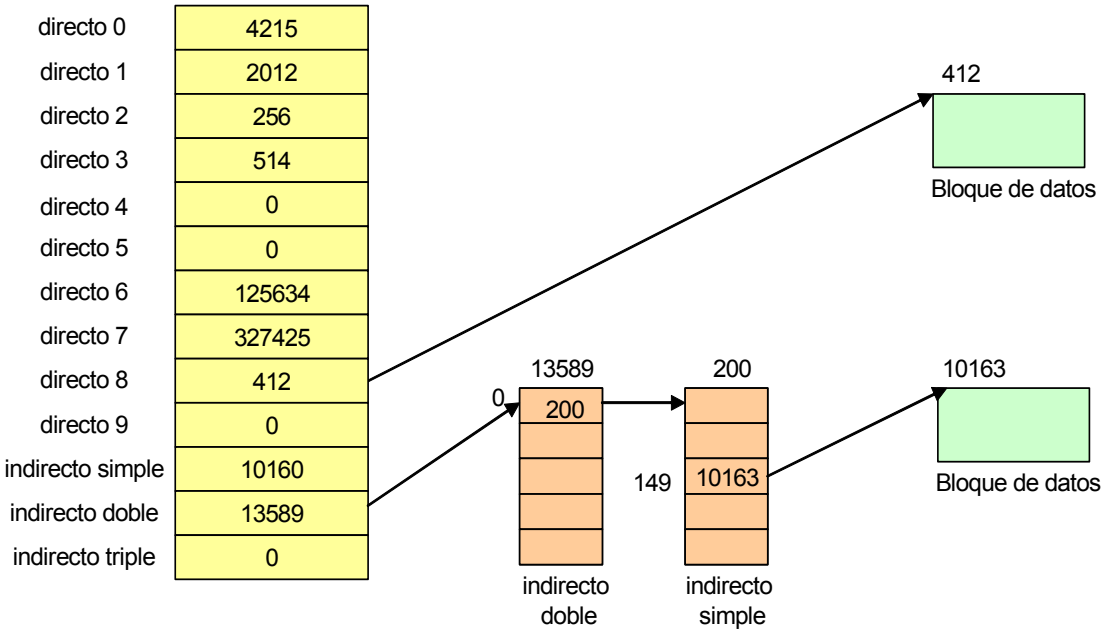


Figura 8.15: Lista de direcciones de bloques físicos almacenada en el campo `di_addr` *del nodo-i*

♦

♦ **Ejemplo 8.14:** *Resolver el ejemplo anterior suponiendo ahora que* D_L=425.000 bytes.

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_I=11** de la tabla de direcciones del nodo-i.

De acuerdo con la Figura 8.15, 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 8.15 la entrada 149 del bloque del disco B'_D contiene el número $\boldsymbol{B_D}$**=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 numeran 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 8.15 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 en el superbloque una lista de nodo-i libres completa y una lista de bloques libres completa. 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 8.16 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.

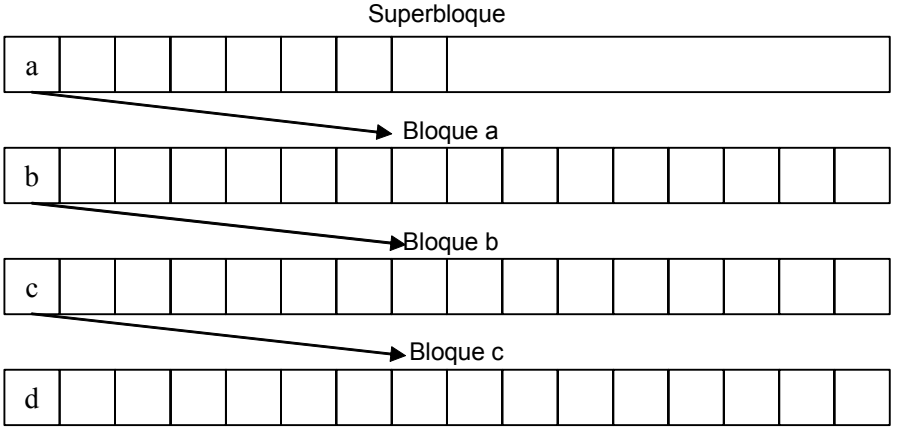


Figura 8.16: Lista de bloques libres en s5fs

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 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 un 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 permiten localizar rápidamente a los nodos-im cuando los necesite. Asimismo mantiene una lista de nodos-im libres.

♦ **Ejemplo 8.16:**

En la Figura 8.17 se representa una posible organización de los nodos-im que consta de cuatro colas de dispersión. La cola de dispersión 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

Nº 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.

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.

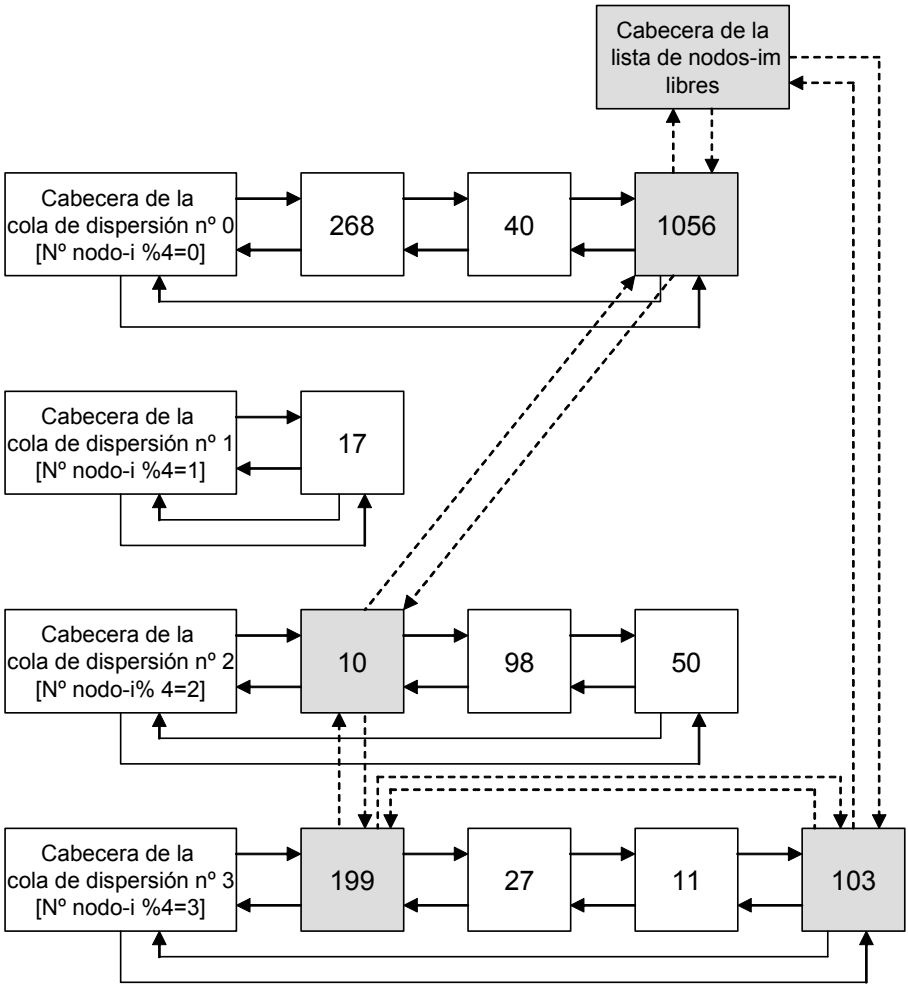


Figura 8.17: Organización de los nodos-im del s5fs (se ha resaltado la lista de nodos-im libres)

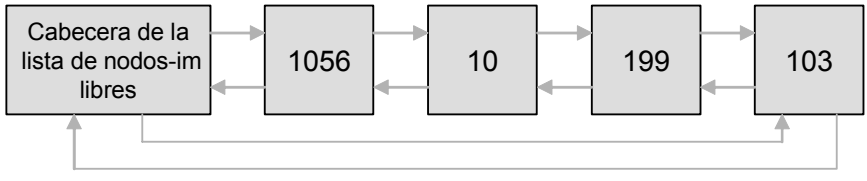


Figura 8.20: Lista de nodos-im libres

♦

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. 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* que 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. 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 como por ejemplo 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 aleatorio a disco.

En segundo lugar, el alojamiento de los bloques en el disco es también poco óptimo. 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 ralentiza 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 en el UNIX BSD4.2. del sistema de ficheros rápido (FFS).

COMPLEMENTO 8.A
Comprobación del estado de un sistema de ficheros

Ciertos programas tales como `newfs` o `mkfs` permiten crear un sistema de ficheros UNIX en un disco físico. Una vez creado se debe revisar para verificar su consistencia y asegurar que todos sus bloques son accesibles. Esto se consigue con el programa `fsck`. Su sintaxis es la siguiente:

```
$ fsck [-opciones] [sistema...]
```

`Fsck` revisa y repara de forma interactiva las posibles inconsistencias que encuentra en los sistemas de ficheros UNIX. En el caso de que no existan inconsistencias, `fsck` informa sobre el número de ficheros, número de bloques usados y número de bloques libres de que dispone el sistema. Si el sistema presenta inconsistencias, `fsck` proporciona mecanismos para corregirlo.

`Fsck` revisa los sistemas de ficheros que se le indican en la línea de órdenes. Si no se especifica ningún sistema, `fsck` revisa los sistemas que se especifican en la tabla de montaje.

Las inconsistencias que revisa `fsck` son las siguientes:

- Bloques reclamados por más de un nodo-i o la lista de bloques libres.
- Bloques reclamados por un nodo-i o la lista de bloques libres, pero que están fuera del rango del sistema.
- Contadores de enlaces incorrectos.
- Número de bloques demasiado grande y tamaño de directorios inadecuados.
- Formato inadecuado para los nodos-i.
- Bloques no registrados por nadie (nodos-i, lista de bloques libres, etc).
- Revisión de los directorios en busca de ficheros que apuntan a nodos-i no asignados o números de nodo-i fuera de rango.
- Existencia en el superbloque de más bloques para nodos-i de los que hay en el sistema de ficheros.
- Formato incorrecto de la lista de bloques libres.
- Total de bloques libres o contador de nodos-i incorrecto.

Si `fsck` encuentra un fichero o directorio cuyo directorio padre no puede determinarse, colocará el fichero huérfano en el directorio `lost+found` perteneciente al sistema que se está revisando. Puesto que el nombre del fichero se registra en su directorio `home` y éste es desconocido, a la hora de guardarlo en `lost+found` se nombrará con su número de nodo-i.

Aparte de revisar un sistema de ficheros recién creado `fsck` se utiliza principalmente para revisar sistemas estropeados por alguna causa accidental como una parada imprevista del sistema. El núcleo mantiene copias en memoria tanto del superbloque como de algunos nodos-i (nodos-mi). Además el acceso a disco se realiza a través de la caché de buffers de bloques de disco. Esto crea inconsistencias de contenido entre el disco y la memoria. Estas inconsistencias se corrigen periódicamente con la intervención de los procesos demonio `syncer` o `update` que se encargan de invocar a la llamada al sistema `sync` para actualizar el disco con la memoria. Si por cualquier circunstancia el sistema deja de funcionar antes de que se produzca una actualización, la próxima vez que se intente utilizar ese sistemas de ficheros, será necesario repararlo dentro de la posible con la ayuda de `fsck`.

COMPLEMENTO 8.B
Consideraciones adicionales sobre la interfaz nodo-v/sfv del SVR4

8.B.1 Partes dependientes del sistema de ficheros de un objeto nodo-v

El nodo-v es un objeto abstracto que no puede existir por sí 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, sí 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, la 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 esta 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,
      ...
};
```

El campo `v_op` del nodo-v apunta a la estructura `vnodeops` para el tipo de sistema de ficheros asociado. Como se muestra en la Figura 8B.1, todos los ficheros del mismo tipo de sistema de ficheros comparten una misma instancia de esta estructura y acceden al mismo conjunto de funciones.

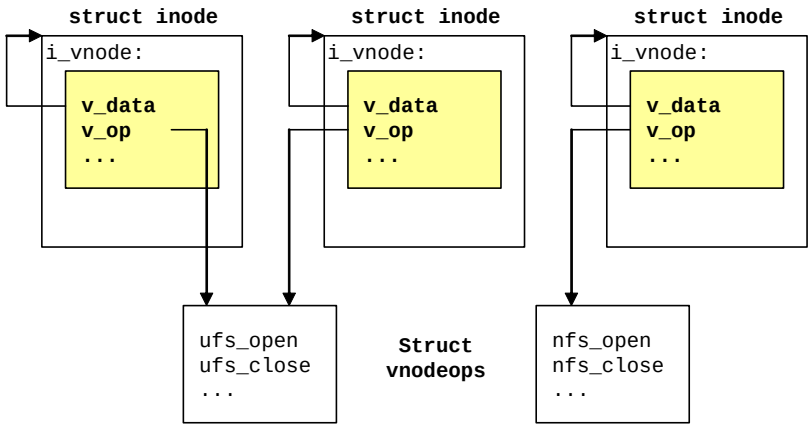


Figura 8B.1: Objetos de un nodo-v dependientes del sistema de ficheros

8.B.2 Partes dependientes del sistema de ficheros de un objeto sfv

Como el nodo-v, el objeto *sfv* 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 *sfv* y su estructura de datos privada normalmente se asignan por separado. El campo `vfs_op` apunta a una estructura `vfsops`, que se define de la siguiente forma:

```
struct vfsops     {
      int (*vfs_mount)();
      int (*vop_unmount)();
      int (*vop_root)();
      int (*vop_statvfs)();
      int (*vop_sync)();
      ...
};
```

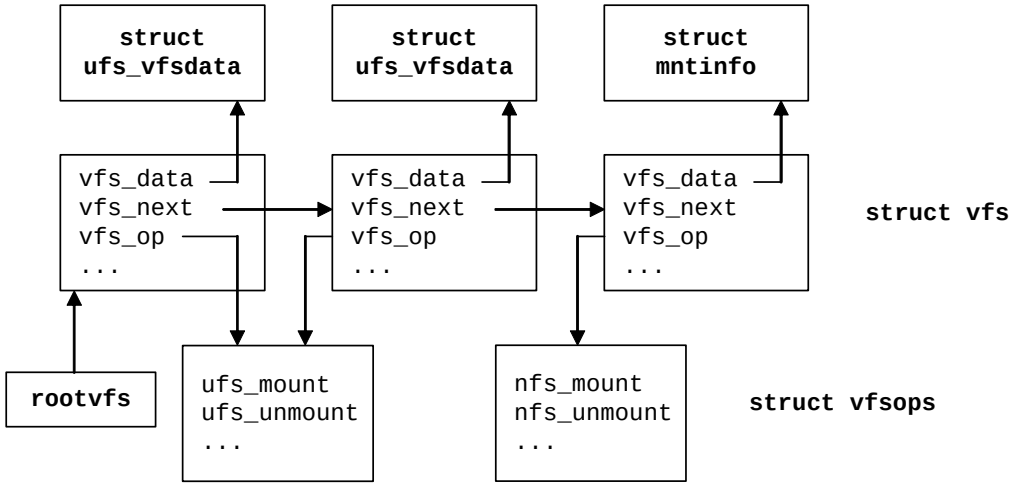


Figura 8B.2: Estructuras de datos de la capa sfv

Cada tipo de sistema de ficheros suministra su propia implementación de estas operaciones. Por lo tanto existe una instancia de la estructura `vfsops` por cada tipo de sistema de ficheros: `ufs_vfsops` para *ufs*, `nfs_vfsops` para NFS, etc. La Figura 8B.2 muestra las estructuras de datos de la capa sfv para un sistema que contiene dos sistemas de ficheros del tipo *ufs* y un sistema de ficheros del tipo NFS.

8.B.3 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.B.4 Implementación de mount

La llamada al sistema `mount` obtiene el nodo-v del directorio punto de montaje llamando a la rutina `lookuppn()`. 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.B.5 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.B.6 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 libera la referencia del nodo-v final y devuelve al proceso invocador un puntero a este nodo-v.

8.B.7 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.B.8 La operación VOP_LOOKUP

VOP_LOOKUP es la 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,&tvp,...);
```

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, `tvp` debe apuntar al nodo-v de `componente` y su contador de referencias debe ser incrementado.

Como otras operaciones en esta interfaz, esto resulta en una llamada a un 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.B.9 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 `lookuppn()` para analizar la ruta de acceso y devolver el nodo-v del fichero para ser abierto. `lookuppn()` 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 de 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.

COMPLEMENTO 8.C
El sistema de ficheros FFS (o UFS)

Los problemas que presenta el sistema de ficheros *s5fs* condujeron al desarrollo de un nuevo sistema de ficheros en el UNIX BSD. Se trata del sistema de ficheros FFS (Fast File System) también conocido como sistema de ficheros UFS (Universal File System) que fue incorporado por primera vez en el BSD4.2.

8.C.1 Organización

Las *unidades de disco* son el principal soporte de la memoria secundaría del computador y en ellos se ubica el sistema de archivos. El número de platos o discos de grabación en una unidad de disco varía en función de su capacidad. Todos los discos de una unidad de disco, giran a la misma velocidad constante (típicamente 3600 rpm). Los datos se leen o se escriben mediante *cabezas de lectura/escritura* montadas de forma que contacten con la parte del disco que contiene los datos. Cada disco tiene dos superficies (o caras) por lo que existen dos cabezas de lectura y escritura para cada disco. Los datos se almacenan en las superficies magnéticas del disco en forma de círculos concéntricos llamados *pistas*. Se llama *cilindro* al conjunto de pistas de todas las superficies de todos los discos de la unidad que se encuentran situadas a la misma distancia del eje de rotación del disco. Las pistas se dividen en *sectores* y cada sector contiene varios centenares de bytes.

Una partición de un disco se compone de un conjunto de cilindros consecutivos del disco. Asimismo, una partición formateada contiene un sistema de ficheros. Un sistema de ficheros FFS divide adicionalmente la partición en uno o más *grupos de cilindros*, cada uno de los cuales contiene un pequeño conjunto de cilindros consecutivos. Este permite a UNIX almacenar datos relacionados en el mismo grupo de cilindros, disminuyendo así los movimientos de la cabeza lectora del disco.

El superbloque de un sistema de ficheros FFS contiene información sobre el sistema de ficheros completo, como por ejemplo, el número, tamaño y posición de cada grupo de cilindros, el tamaño de un bloque, el número total de bloques y nodos-i, etc. Adicionalmente, cada grupo de cilindros tiene una estructura de datos que contiene información sobre el grupo, incluyendo las listas de nodos-i libres y de bloques libres. Los datos del superbloque son muy importantes y deben ser protegidos de posibles errores del disco. Por ello, aparte de ubicarse el superbloque en su localización habitual, es decir, al comienzo de una partición (después del área de arranque), cada grupo de cilindros contiene una copia del superbloque. FFS mantiene estas copias en diferentes posiciones

en cada grupo de cilindros de tal forma que ninguna pista, cilindro o plato contenga todas las copias del superbloque. El espacio entre el comienzo de un grupo de cilindros y la copia del superbloque es utilizada para bloques de datos, excepto en el caso del primer grupo de cilindros.

Es conocido que si el tamaño de los bloques de datos de un disco es grande se mejora el rendimiento del sistema ya que se estarían transmitiendo más datos en una operación de E/S. Sin embargo, se desperdiciaría más espacio (en promedio, cada fichero desperdicia medio bloque). FFS intenta mejorar el rendimiento del sistema y disminuir el espacio desperdiciado dividiendo los bloques en *fragmentos*. En FFS, aunque todos los bloques en un sistema de ficheros deben tener el mismo tamaño, diferentes sistemas de ficheros en la misma máquina pueden tener diferentes tamaños de bloques. El tamaño de un bloque es una potencia de dos mayor o igual a 4096. La mayoría de las implementaciones añaden un límite superior de 8192 bytes. Este valor es mucho más grande que el usado en *s5fs* (512 o 1024), además incrementa la productividad permitiendo ficheros tan grande como 2^{32} bytes (4 Gbytes) que son direccionados con dos niveles de indirección únicamente. FFS no utiliza la indirección triple, aunque algunas variantes soportan tamaños de ficheros mayores de 4 Gbytes.

Los sistemas UNIX típicos tienen numerosos ficheros pequeños que requieren ser almacenados eficientemente. Un tamaño de bloque de 4 Kbytes desperdicia mucho espacio en el caso de estos ficheros. FFS resuelve este problema permitiendo que cada bloque sea dividido en dos o más fragmentos. El tamaño de un fragmento es fijo para un sistema de ficheros y se especifica cuando se crea dicho sistema. El número de fragmentos por bloque puede ser configurado a 1, 2, 4 o 8 permitiendo un límite inferior de 512 bytes, el mismo que el tamaño de un sector del disco. Cada fragmento puede ser individualmente direccionado y asignado. Para ello es necesario sustituir la lista de bloques libres por un mapa de bits con un bit por fragmento.

Un fichero FFS está compuesto por completo por bloques de disco, excepto en el caso del último bloque, que puede contener uno o más fragmentos consecutivos. El bloque de un fichero debe estar completamente contenido dentro de un único bloque de disco. Incluso si dos bloques de disco adyacentes tiene suficientes fragmentos libres consecutivos para almacenar un bloque de un fichero, ellos no se pueden combinar. Adicionalmente, si el último bloque de un fichero contiene más de un fragmento, éstos deben ser contiguos y parte del mismo bloque.

Este esquema reduce el desperdicio de espacio pero produce un duplicado ocasional de datos del fichero. Considérese un fichero llamado `prueba` cuyo último bloque ocupa un único fragmento. Los fragmentos restantes de ese bloque pueden ser asignados a otros ficheros. Si `prueba` aumenta su tamaño en otro fragmento, será necesario encontrar otro bloque con dos fragmentos libres consecutivos. El primer fragmento debe ser copiado desde su posición original y el segundo rellenado con los nuevos datos. Si `prueba` crece usualmente en pequeños incrementos, sus fragmentos pueden tener que ser copiados varias veces, empeorando el rendimiento del sistema. FFS controla este problema forzando a que los fragmentos únicamente puedan ser contenidos por bloques con direccionamiento directo.

8.C.2 Políticas de asignación

A diferencia de s5fs, FFS intenta tener bien organizada la información sobre el disco, así como optimizar los accesos secuenciales a dicha información. FFS suministra un mayor control en la asignación de bloques de disco y nodos-i, así como en los directorios. Estas políticas de asignación utilizan el concepto de grupo de cilindros y requieren que el sistema de ficheros conozca varios parámetros asociados con el disco. A continuación se recogen las principales reglas de estas políticas de asignación:

- Intentar situar los nodos-i de todos los ficheros de un mismo directorio en el mismo grupo de cilindros. Muchos comandos (`ls -l` sería el mejor ejemplo) acceden a todos los nodos-i de un directorio en una rápida sucesión. Los usuarios tienden a exhibir una cierta localidad en sus accesos, trabajando sobre muchos ficheros en el mismo directorio (directorio de trabajo actual) antes de moverse a otro.

- Crear cada nuevo directorio en un grupo de cilindros diferente al de su directorio padre, para así conseguir una distribución homogénea de los datos sobre el disco. La rutina de asignación elige el nuevo grupo de cilindros de entre los grupos con un número de nodos-i libres superior a la media; de estos, selecciona aquel que posea el menor número de directorios.

- Intentar situar los bloques de datos de un fichero en el mismo grupo de cilindros que el nodo-i, porque típicamente el nodo-i y los datos serán accedidos conjuntamente.

- Evitar rellenar un grupo de cilindros entero con un fichero grande, cambiar el grupo de cilindros cuando el tamaño del fichero alcance los 48 Kbytes y de nuevo en cada megabyte. Este límite de 48 Kbytes fue elegido porque para un bloque de tamaño 4096 bytes, las entradas de bloques directos del nodo-i

describen los primeros 48 Kbytes. En FFS el número de bloques directos en el array de direcciones fue incrementado de 10 a 12. La selección de un nuevo grupo de cilindros está basado en el número de bloques libres.

- Asignar bloques secuenciales de un fichero en posiciones óptimas rotacionalmente, si es posible. Cuando un fichero está siendo leído secuencialmente, hay un retardo de tiempo entre que la lectura de un bloque se completa y el núcleo procesa la terminación de la E/S e inicia la siguiente lectura. Puesto que el disco está girando durante este tiempo, uno o más sectores pueden haber pasado bajo la cabeza del disco. La optimización rotacional intenta determinar el número de sectores que se deben dejar pasar por debajo de la cabeza del disco hasta que el sector deseado esté bajo la cabeza del disco cuando se inicia la operación de lectura. A este número se le conoce como *factor de entrelazado del disco*.

La política de asignación utilizada por un sistema de ficheros FFS es muy efectiva cuando el disco tiene bastante espacio libre, pero se deteriora rápidamente si el disco está cerca del 90% de su capacidad. Cuando existen pocos bloques libres, es difícil encontrar bloques libres en localizaciones óptimas. Por lo tanto, FFS debe mantener una reserva de espacio, usualmente el 10% de la capacidad del disco. Sólo el superusuario puede asignar espacio de esta reserva.

8.C.3 Mejoras en la funcionalidad de un sistema de ficheros FFS

Una de las principales mejoras en la funcionalidad de un sistema de ficheros FFS con respecto a un sistema de ficheros *s5fs* es la posibilidad de usar *nombres de ficheros largos*. FFS cambió la estructura de los directorios para permitir que los nombres de ficheros fuesen mayores de 14 caracteres. Las entradas de un directorio FFS varían en longitud. La parte fija de la entrada consiste del número de nodo-i, el tamaño asignado y el tamaño del nombre del fichero en la entrada. Éste está seguido por un nombre de fichero terminado en un carácter nulo con espacio extra de 4 bytes. El tamaño máximo de un nombre de fichero es de 255 caracteres. Cuando se borra un nombre de fichero, FFS fusiona el espacio liberado con una entrada previa. Por lo tanto, el campo de tamaño asignado almacena el espacio total consumido por la parte variable de la entrada. El propio directorio está asignado en trozos de 512 bytes y ninguna entrada puede ocupar varios trozos. Finalmente para facilitar la escritura de código portable, la librería estándar añade un conjunto de rutinas de acceso a directorios que permiten el acceso independiente del sistema de ficheros a la información del directorio.

Otras de las principales mejoras en la funcionalidad de un sistema de ficheros FFS es la implementación de enlaces simbólicos, los cuales solucionaban muchas de las limitaciones de los enlaces duros.

Por otra parte BSD4.2 añadió una llamada al sistema `rename` para permitir el renombramiento atómico de ficheros y directorios, lo cual requería una llamada al sistema `link` seguida de una llamada `unlink`. Esto añadía un mecanismo de cuota para limitar los recursos disponibles del sistema de ficheros para cualquier usuario. Las cuotas se aplican tanto a los nodos-i como a los bloques de disco y tienen un límite suave que dispara un aviso, junto con un límite duro que el núcleo hace respetar.

Algunas de las mejoras comentadas han ido siendo incorporadas dentro de *s5fs*. En SVR4, *s5fs* permitía enlaces simbólicos y soporte atómico para renombrar un fichero. No obstante no soporta nombre de ficheros largos o cuotas de disco.

8.C.4 Análisis

En general, FFS posee mayores ventajas que *s5fs*, lo que ha contribuido a su aceptación. De hecho, UNIX System V añadió en SVR4 a FFS como sistema de ficheros soportado. Por ejemplo, medidas realizadas en un VAX/750 mostraban que la productividad de lectura se incrementaba de 29 Kbytes/s en un *s5fs* (con bloques de 1 Kbyte) a 221 Kbytes/s en un FFS (con bloques de 4 Kbytes y fragmentos de 1 Kbyte). Además la utilización de la CPU se incrementaba de 11% a 43%. Con la misma configuración la productividad de escritura se incrementaba de 48 a 142 Kbytes/s y el uso de CPU de 29% a 43%.

Con respecto al espacio desperdiciado en promedio en el disco, un sistema de ficheros *s5fs* desperdicia medio bloque por fichero. Mientras que un sistema FFS desperdicia medio fragmento por fichero. La ventaja de tener bloques grandes es que se requiere menos espacio para alojar todos los bloques de un fichero grande. Por lo tanto el sistema de ficheros requiere de menos bloques indirectos. En contraposición, se requiere más espacio para monitorizar los bloques libres y los fragmentos. Por lo tanto estos dos factores tienden a cancelarse por lo que el resultado neto de utilización del disco llega a ser prácticamente el mismo cuando el tamaño de un fragmento se iguala al de un bloque de s5fs. La reserva de espacio libre necesaria para FFS, no obstante, debe ser contabilizada como espacio desperdiciado, puesto que no está disponible para los ficheros de los usuarios. Cuando se contabiliza este factor, el porcentaje de espacio desperdiciado en un *s5fs* con bloques de 1 Kbyte se hace aproximadamente igual al de un FFS con bloques de 4 Kbytes y fragmentos de 512 bytes y reserva de espacio libre del 5 %.

CAPÍTULO

9

Gestión de memoria en UNIX

9.1 INTRODUCCIÓN

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 muy preciado.

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* (MMU[1]), 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, TLBs[2], o ambos.

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

[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.

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.

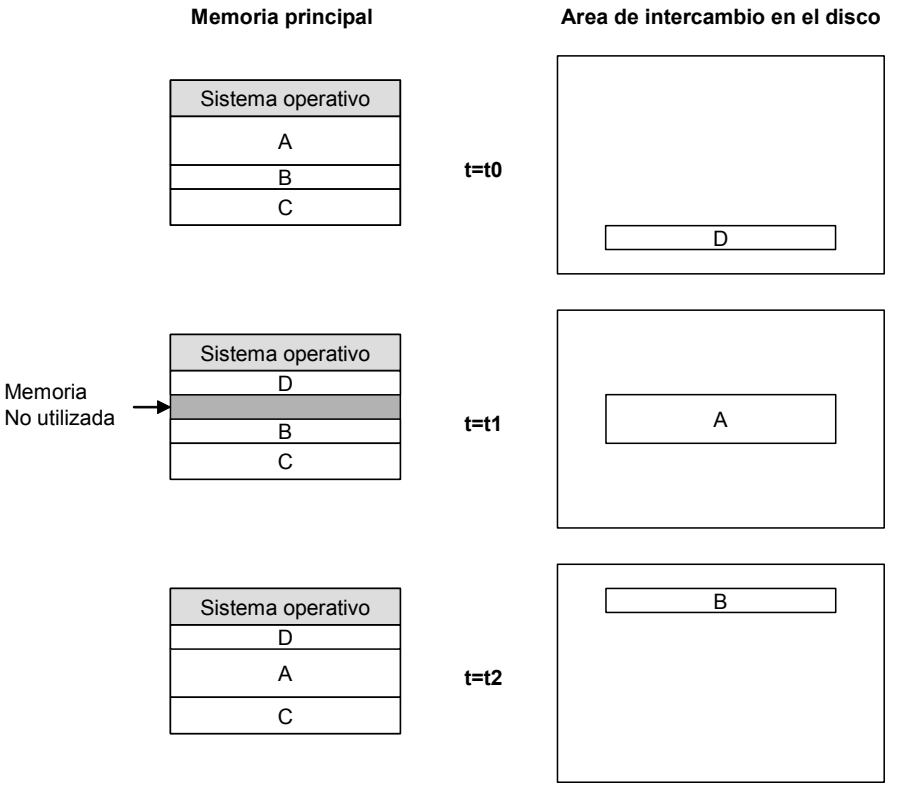


Figura 9.1: Administración de memoria basada en intercambio

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).

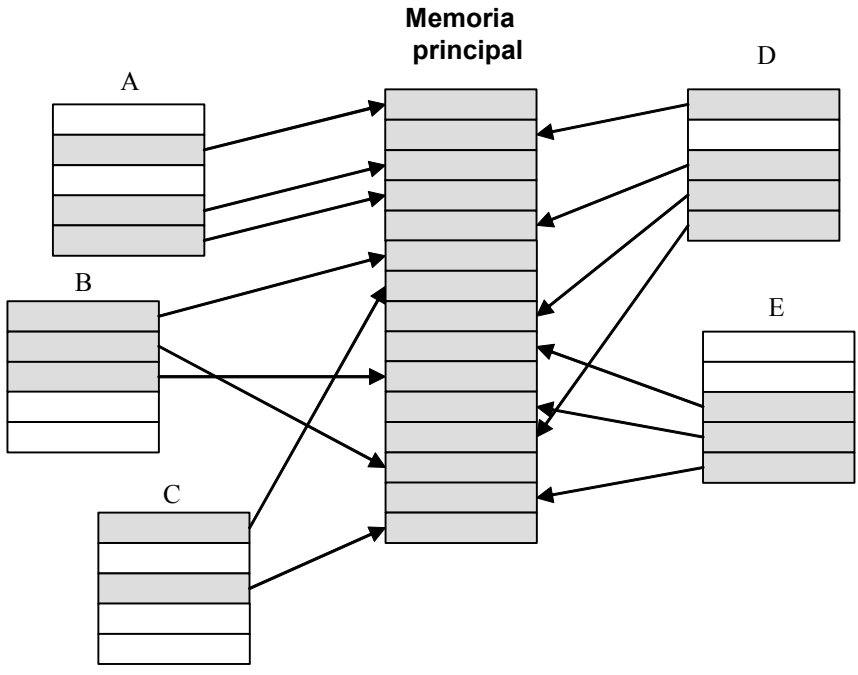


Figura 9.2: La memoria física almacena unas pocas páginas de cada proceso

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 cómo 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 en 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, 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 capítulo está dedicado principalmente a describir la política de gestión de memoria mediante *demanda 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 cómo 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 Capítulo 4, 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* divida 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 configura inicialmente durante la llamada al sistema `fork`. Si este campo está activado y

un proceso intenta escribir en la página, se produce un fallo de protección. Al tratar este fallo el núcleo creará una nueva copia de la página.

- *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.
- *Válida,* 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 *válida*, *copiar al escribir*, *edad y bits de protección*, 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:

$$DIR_V = DIR_{V0} + DES_V \tag{1}$$

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:

$$DIR_F = DIR_{F0} + DES_F \tag{2}$$

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:

$$i = floor\left(\frac{DES_V}{S_P}\right) \tag{3}$$

La función matemática *floor (X)* redondea X hacia el entero más cercano a menos infinito, por ejemplo, *floor*(2.1)=2, *floor*(2.5)=2, *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:

$$DES_F = DES_V \bmod S_P = DES_V \,\%\, S_P \qquad (4)$$

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=1Kbytes. Se desea Calcular la dirección física DIR_F asociada a DIR_V.

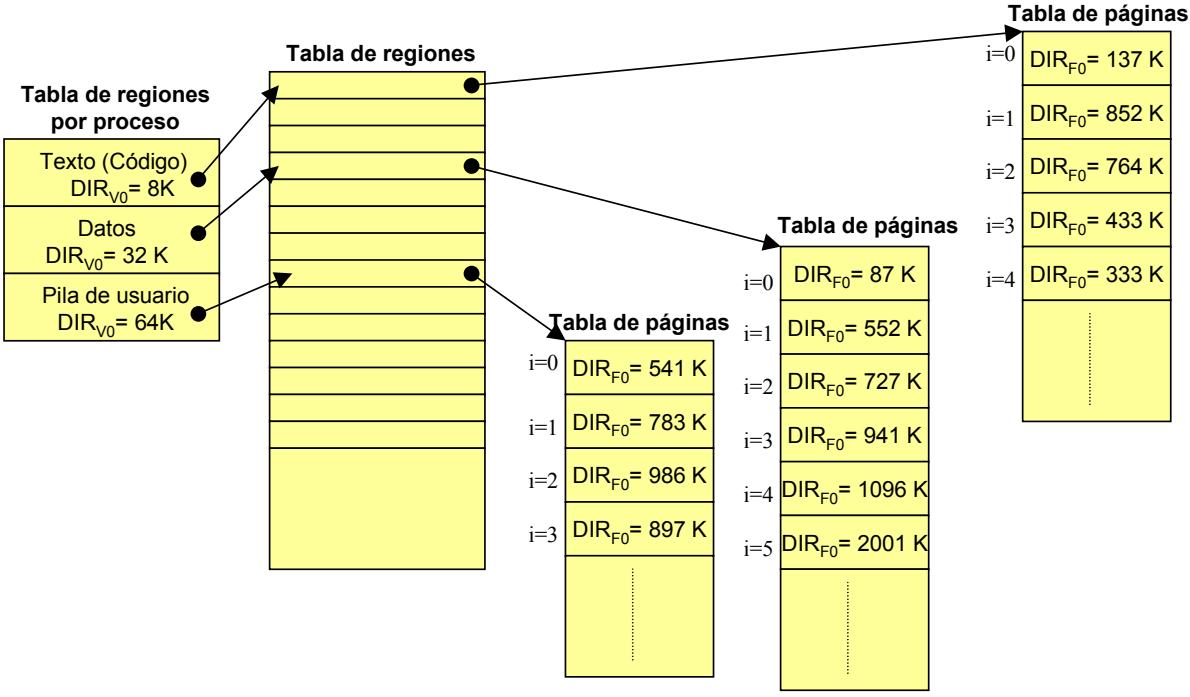


Figura 9.3: Asignación de memoria física de un proceso A.

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} =64K=64·2^{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:

$$DES_V = DIR_V - DIR_{V0} = 68432 - 65536 = 2896$$

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):

$$i = floor\left(\frac{DES_V}{S_P}\right) = floor\left(\frac{2896}{1024}\right) = floor(2.82) = 2$$

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 dónde se almacena la copia de la página.

- *Tipo*. Este campo permite al núcleo conocer dónde 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 del fichero que contiene la página. El núcleo puede rápidamente traducir este número en direcciones del 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:

$$N_{TM} = \frac{C_{Mp}}{S_P} \tag{5}$$

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,...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 descompone en dos campos (ver Figura 9.4): el número *j* de marco de página y el desplazamiento relativo dentro de la página DES_F .

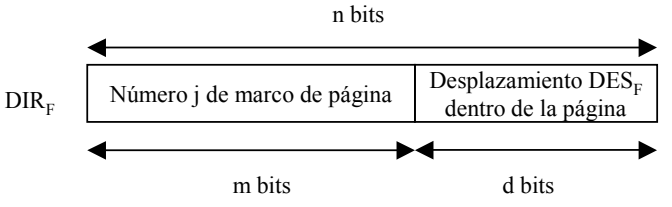


Figura 9.4: Dirección de memoria principal

♦ **Ejemplo 9.2:**

Supóngase un computador con una memoria principal de capacidad C_{Mp}=2 Mbytes y un tamaño de página S_P= 1Kbytes. 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

Luego *j*= 01111011010 = 986 y DES_F = 1101010000= 848

Este resultado está de acuerdo con el obtenido en el Ejemplo 9.1.

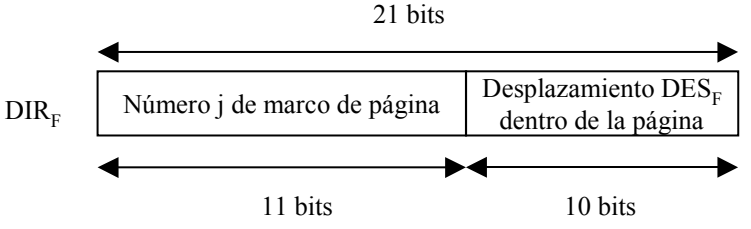


Figura 9.5: Configuración de DIR_F

♦

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 contendida 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 han 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.

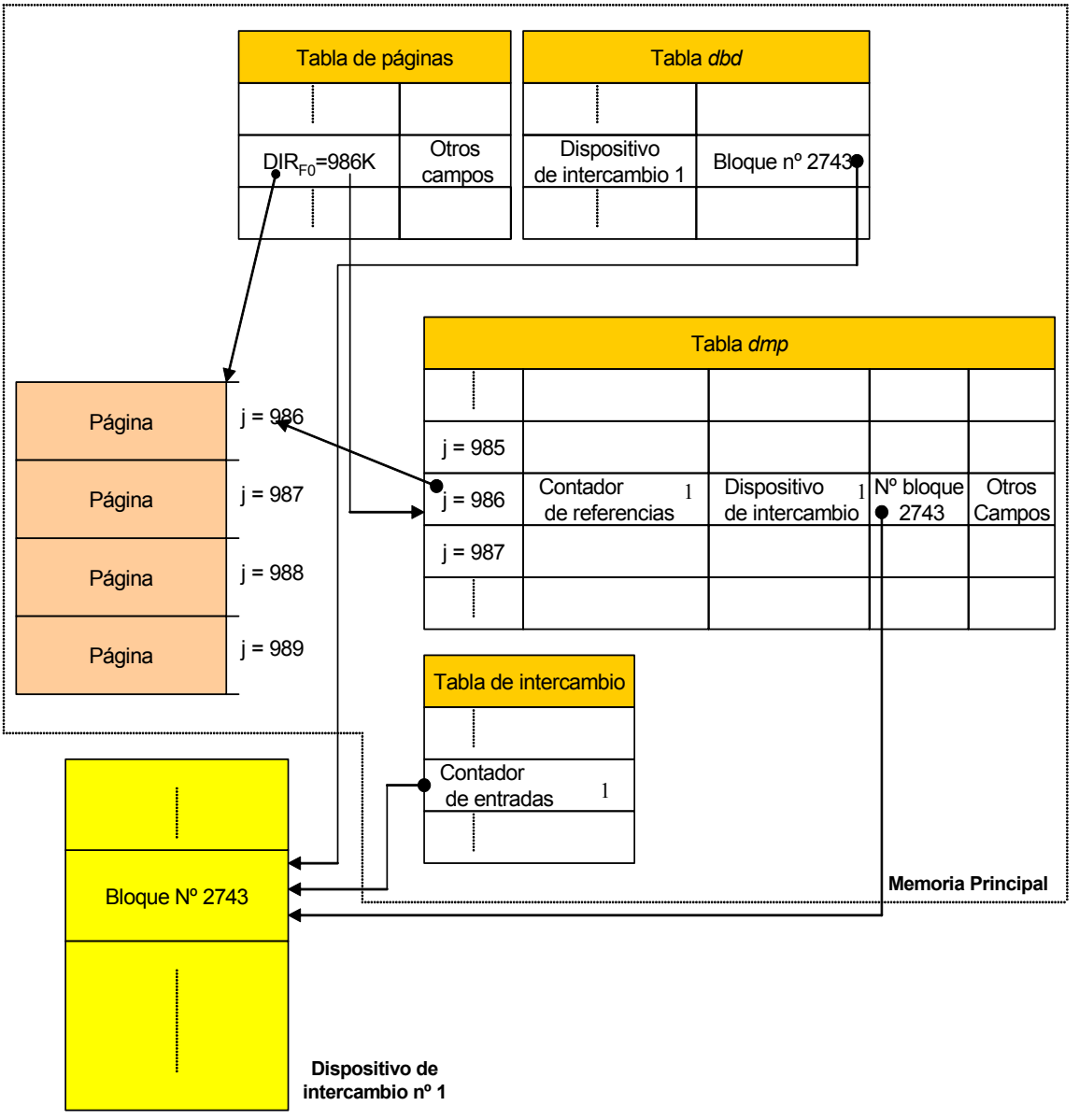


Figura 9.6: Estructuras de datos asociadas a la gestión de la memoria mediante demanda de páginas.

♦

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 privada del padre y del hijo durante `fork`. Si un proceso escribe una página, provocará un fallo de protección. 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.

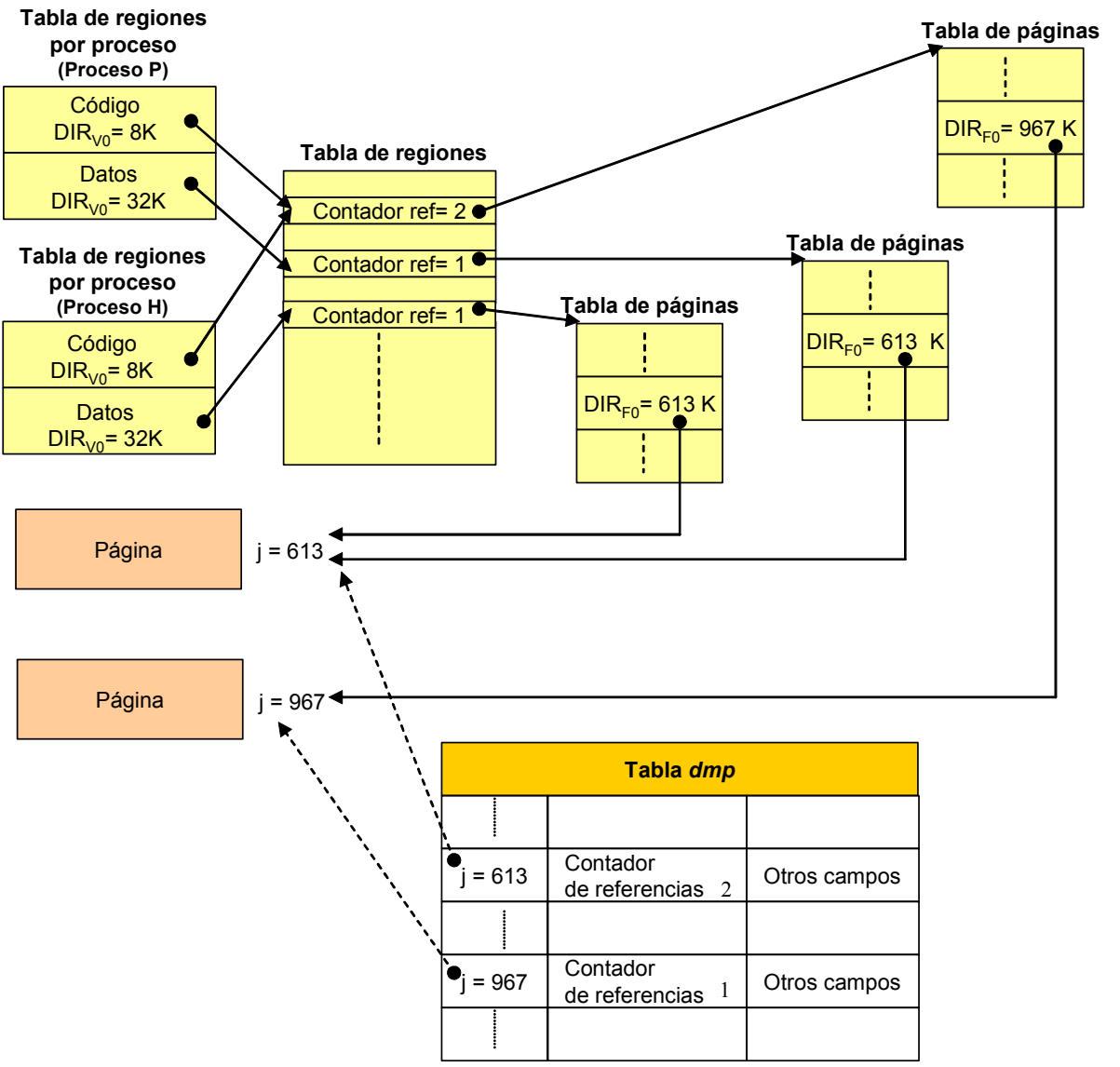


Figura 9.7: Una página en un proceso que ha realizado una llamada al sistema `fork`.

♦

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 y va marcando las entradas de las tablas *dbd* 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 a la región a la que está asociada la tabla de páginas 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.

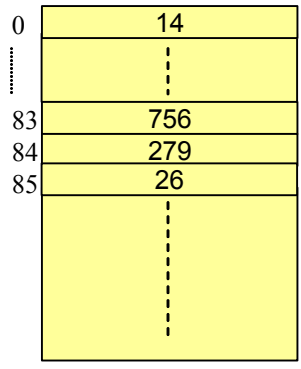


Figura 9.8: Ejemplo de lista de números de bloques del fichero ejecutable almacenada en el nodo-i durante la ejecución de la llamada al sistema `exec`

♦

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 administrador 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 tener que leerla del dispositivo de intercambio. No obstante, la página será, de todos modos, intercambiada si ya se ha colocado en la lista de páginas 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 por qué ser necesariamente contiguas.

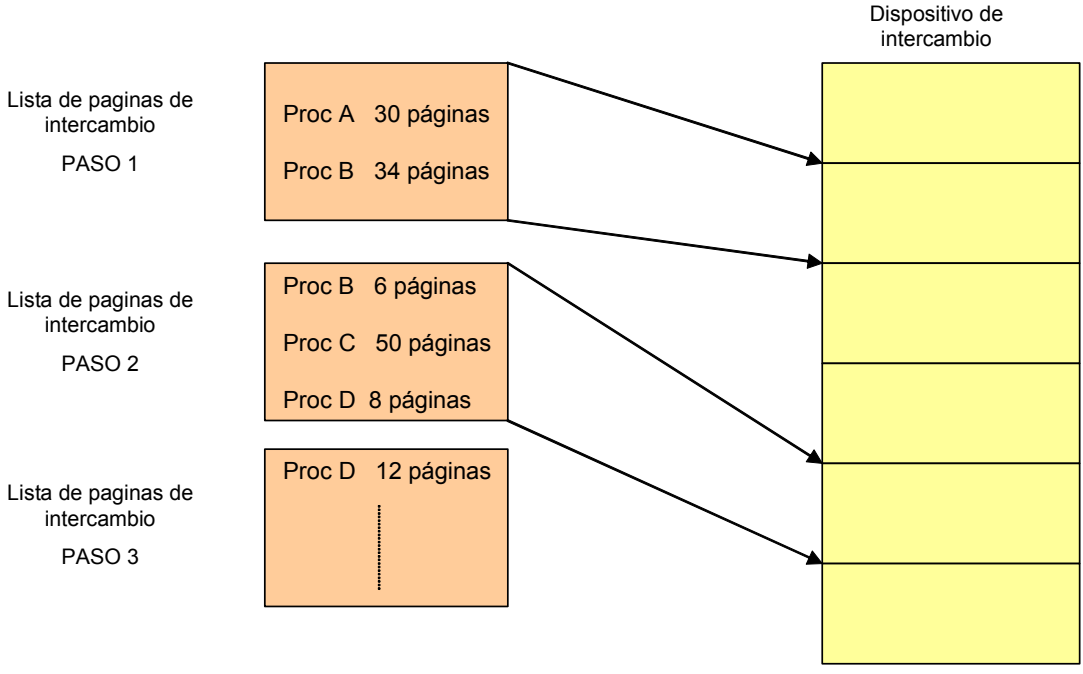


Figura 9.9: Asignación del espacio de intercambio en un esquema de gestión de memoria por demanda de página.

♦

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 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.

♦

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 paginas libres, si fuese necesario.

DIR_{V0}	Tabla de páginas		Tabla *dbd*	
	Marco	**Válida**	**Tipo**	**Nº de bloque**
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 *dmp*		
Marco	**Contador de referencias**	**Nº de bloque**
⋮		
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.

	Tabla de páginas		Tabla *dbd*	
DIR_{V0}	**Marco**	**Válida**	**Tipo**	**Nº de bloque**
	⋮			
66K	1776	1	Disco	847

Tabla *dmp*		
Marco	**Contador de referencias**	**Nº de bloque**
⋮		
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.

♦ **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 sí 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 realizadas 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 SIGBUS[3] al proceso que provocó el fallo, desbloquea la región y finaliza. Cuando la señal sea tratada provocará la finalización del proceso.

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 decrementa 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, decrementa 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).

[3] No todas las distribuciones envían esta misma señal.

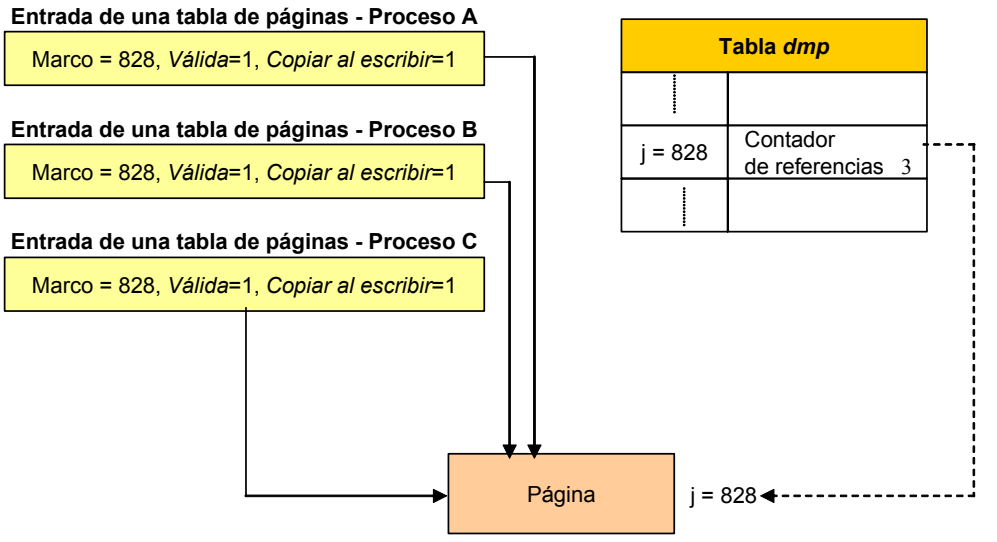


Figura 9.12: *Detalle de parte del contenido de algunas de las estructuras asociadas a la gestión de memoria mediante demanda de páginas en un determinado instante de tiempo*

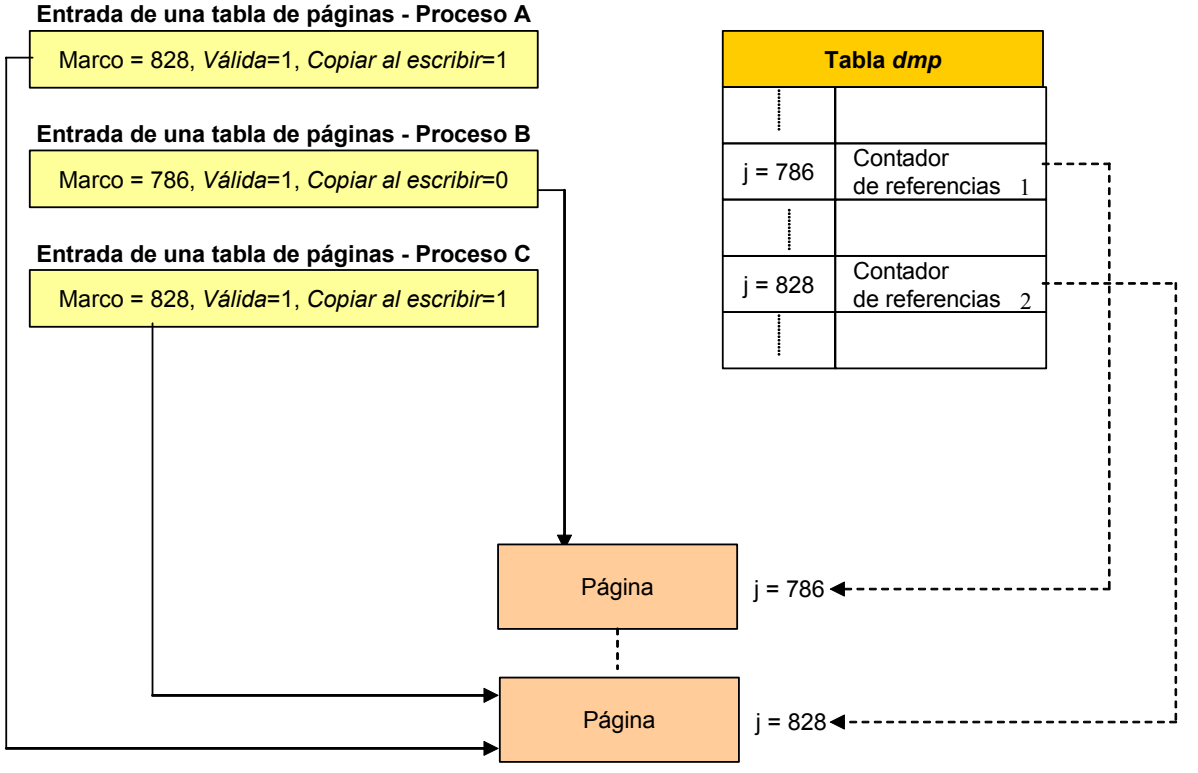


Figura 9.13: *Detalle de parte del contenido de algunas de las estructuras asociadas a la gestión de memoria mediante demanda de páginas después de gestionar el fallo de protección*

♦

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 1Kbyte, 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 (sólo 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 maquina 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 prohiben 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.

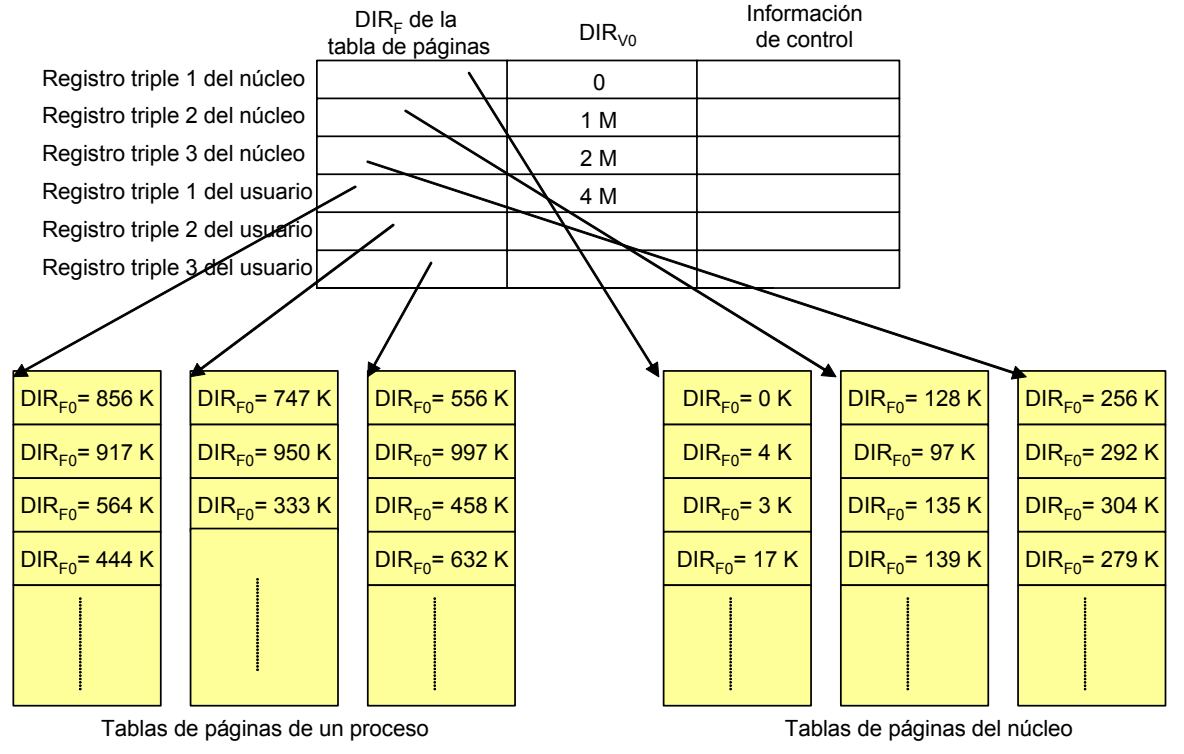


Figura 9.14: Ubicación en memoria de las tablas de páginas del núcleo y de un cierto proceso.

♦

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 prohiba 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 del 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:**

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.

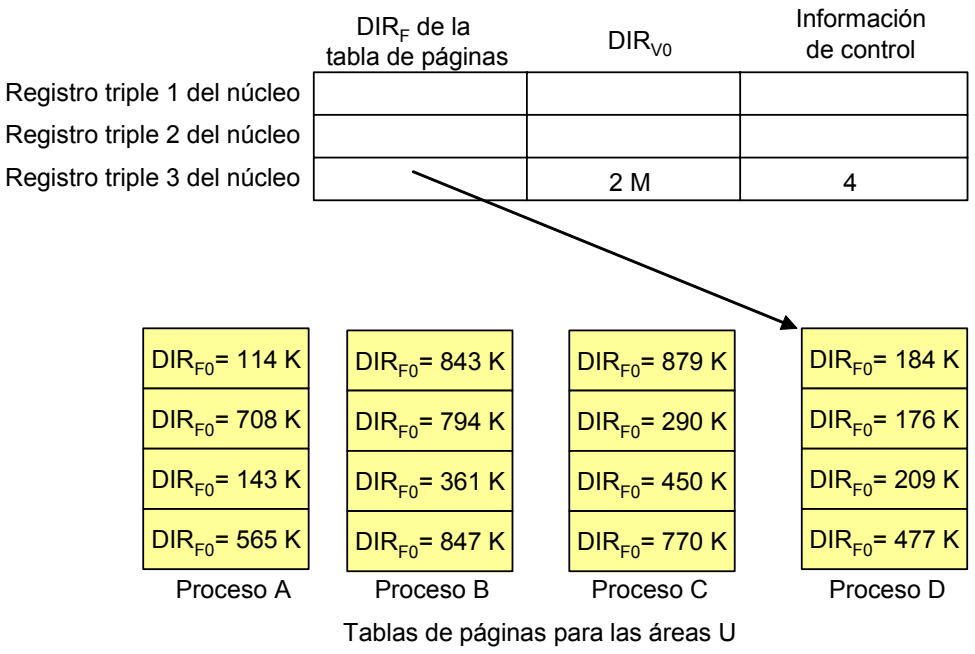


Figura 9.15: Localización en memoria de las tablas de páginas del área U de diferentes procesos.

♦

CAPÍTULO

10 El subsistema de entrada/salida de UNIX

10.1 INTRODUCCIÓN

El subsistema de entrada/salida de UNIX se encarga de la transferencia de datos entre la memoria principal y los dispositivos periféricos (discos duros, impresoras, terminales,...). El núcleo interactúa con estos dispositivos mediante los drivers de dispositivos. Un driver controla uno o más dispositivos y es la única interfaz existente entre el dispositivo y el resto del núcleo. Esta separación permite ocultar al núcleo las complejidades del hardware de cada dispositivo. Así el núcleo puede acceder al dispositivo usando una interfaz consistente y de funcionamiento simple.

En este capítulo en primer lugar se realizan unas consideraciones generales sobre la entrada/salida en UNIX. En segundo lugar se describen los drivers de dispositivos. En tercer lugar se analiza la implementación del subsistema de entrada/salida de UNIX. El capítulo finaliza con una introducción a los STREAMS que suministran, entre otras muchas funcionalidades, una aproximación modular a la escritura de drivers de dispositivos.

10.2 CONSIDERACIONES GENERALES

Un *driver de dispositivo* es una parte del núcleo que consiste en una colección de estructuras de datos y funciones que controlan a uno o más dispositivos, e interactúa con el resto del núcleo mediante una interfaz bien definida. El driver es el único módulo del núcleo que puede interactuar con el dispositivo y no interactúa con otros drivers. Suele estar escrito por el fabricante del dispositivo. El núcleo puede acceder al driver mediante una interfaz de pequeño tamaño pero bien definida. Muchas son las ventajas de usar esta aproximación:

- Es posible aislar el código especifico de cada dispositivo en módulos separados.
- Es fácil añadir nuevos dispositivos.
- Los fabricantes pueden añadir dispositivos a una computadora sin el código fuente del núcleo.

- El núcleo dispone de una vista consistente de todos los dispositivos y accede a todos ellos mediante la misma interfaz.

La Figura 10.1 ilustra el papel del driver de un dispositivo. Las aplicaciones de usuario se comunican con los dispositivos periféricos a través del núcleo, usando la interfaz de las llamadas al sistema. El subsistema de entrada/salida del núcleo trata estas peticiones. Y utiliza la interfaz del driver de dispositivo para comunicarse con los dispositivos.

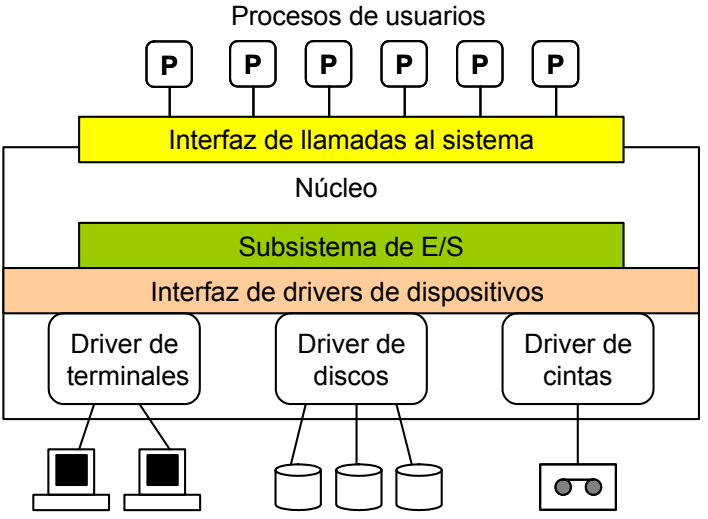


Figura 10.1: Papel de un driver de dispositivo

Cada capa tiene un entorno y unas responsabilidades bien definidas. Las aplicaciones de usuario no necesitan conocer si se están comunicando con un dispositivo o con un fichero ordinario. Un programa que escribe datos en un fichero debería ser capaz de escribir el mismo dato en un terminal o en una impresora sin tener que ser modificado o recompilado. Por lo tanto, el sistema operativo suministra una vista de alto nivel consistente de todo el hardware de la máquina a los procesos de usuarios.

El núcleo delega todas las operaciones con los dispositivos al subsistema de E/S, que es responsable de realizar todos el procesamiento independiente del dispositivo. El subsistema de E/S desconoce las características de cada dispositivo individual. Considera a los dispositivos como abstracciones de alto nivel manipuladas por la interfaz del driver de dispositivo y se encarga de tareas tales como el control de acceso, el almacenamiento temporal de datos y la identificación de los dispositivos.

El driver es responsable de toda la interacción, propiamente dicha, con el dispositivo. Cada driver maneja a uno o varios dispositivos de características similares. Por ejemplo, un driver de un disco duro puede manejar varios discos duros de características

similares. Únicamente conoce las características del hardware del dispositivo tales como, el número de sectores, pistas y cabezas de lectura.

El driver acepta ordenes del subsistema de E/S a través de la interfaz del driver del dispositivo. También recibe mensajes de control procedentes del dispositivo, los cuales incluyen la terminación de una operación, el estado del dispositivo y las notificaciones de errores. El dispositivo consigue la atención del driver mediante la generación de una interrupción. Cada driver tiene asociado un manipulador de interrupciones, que el núcleo invoca cuando recibe la interrupción apropiada.

10.2.1 Configuración del hardware

Los drivers de dispositivos son, por naturaleza, extremadamente dependientes del hardware. La estructura del driver tiene en cuenta como la CPU interactúa con el dispositivo. En la Figura 10.2 se esquematiza la configuración del hardware de un sistema típico.

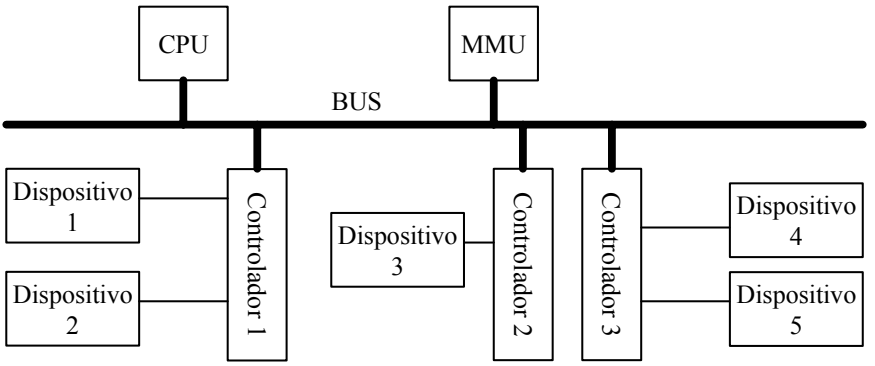


Figura 10.2: Configuración del hardware de un sistema típico

El bus del sistema es un canal de comunicación al que están conectados la CPU, la MMU y los controladores de los dispositivos.

Se puede considerar que un dispositivo está compuesto de dos componentes: una parte electrónica, que es denominada *controlador* o *adaptador*; y una parte mecánica, que es el propio dispositivo. El *controlador* es normalmente un circuito impreso en una tarjeta que se conecta a la computadora y al bus. Una computadora de sobremesa típica tendrá un controlador de disco, una tarjeta gráfica, una tarjeta de E/S y posiblemente una tarjeta de red.

Cada controlador puede tener conectados uno o más dispositivos, que suelen ser del mismo tipo, aunque no tiene porque ser necesariamente así. Por ejemplo, un controlador SCSI (Small Computer Systems Interface) puede controlar discos duros, disqueteras, unidades de CD-ROM y unidades de cinta.

El controlador tiene un conjunto de *registros de control y de estado* (RCE) para cada dispositivo. Cada dispositivo puede tener uno o varios RCE y sus funciones son completamente dependientes del dispositivo. El driver escribe en los RCE para mandar órdenes a los dispositivos y los lee para conocer el estado del dispositivo o la aparición de algún error. Estos registros son bastante diferentes de otros registros de propósito general. Escribir directamente en un registro de control genera una acción sobre el dispositivo, como por ejemplo iniciar una operación de E/S con un disco. Leer un registro de estado puede tener diferentes efectos, como por ejemplo, limpiar el registro. Por lo tanto, el driver no obtiene los mismos resultados si lee un registro de dispositivo dos veces seguidas. Por el contrario, si intenta leer un registro que acaba de ser escrito, el valor leído puede ser bastante diferente del valor escrito.

El espacio de E/S de una computadora incluye el conjunto de todos los registros de dispositivo, así como buffers para almacenamiento temporal de datos. Cada registro tiene una dirección bien definida en el espacio de E/S. Estas direcciones son usualmente asignadas cuando se arranca la máquina, usando un conjunto de parámetros especificados en un fichero de configuración utilizado para montar el sistema. El sistema podría asignar un rango de direcciones a cada controlador que a su vez podría asignar espacio para cada dispositivo que controla.

Existe dos formas de configurar el espacio de E/S en un sistema. En algunas arquitecturas como la Intel 80x86, el espacio de E/S está separado del espacio de memoria principal y es accedido por instrucciones de E/S especiales. A esta configuración se le denomina *E/S aislada de memoria*. En otras arquitecturas como Motorola 680x0, se utiliza una configuración denominada *E/S localizada en memoria*, que consiste en reservar un conjunto del espacio de direcciones de memoria para el espacio de E/S, de esta forma se usan instrucciones de acceso a memoria ordinarias para escribir y leer en los registros.

Asimismo, existen dos formas de transferir datos entre el núcleo y el dispositivo. El método utilizado depende del propio dispositivo. Es posible clasificar a los dispositivos en dos categorías en función del método de transferencia de datos utilizado:

- *Dispositivos de E/S controlada por programa*. Requieren que la CPU se encargue de la transferencia de datos byte a byte. Cuando el dispositivo está listo para enviar o recibir el siguiente byte, activa una interrupción. Ejemplos típicos de este tipo de dispositivos son los modems y las impresoras en línea.

- *Dispositivos con acceso directo a memoria (DMA)*. El núcleo puede dar la localización (fuente y destino) del dato en memoria, la cantidad de datos a transferir y otras informaciones relevantes. El dispositivo completará la transferencia accediendo directamente a memoria, sin la intervención de la CPU. Cuando la transferencia está completa, el dispositivo interrumpe a la CPU para indicarle que ya se encuentra listo para realizar la siguiente operación. Ejemplos típicos de este tipo de dispositivos son los discos.

10.2.2 Interrupciones asociadas a los dispositivos

Los dispositivos utilizan interrupciones para conseguir la atención de la CPU. La manipulación de las interrupciones es altamente dependiente de la máquina, aunque es posible dar una serie de principios generales. Muchos sistemas UNIX definen un conjunto de *niveles de prioridad de interrupción* (*npi*). El número de *npis* soportado es diferente para cada sistema. El *npi* más bajo es cero; de hecho, todo el código de usuario y la mayoría del código normal del núcleo se ejecuta a *npi* 0. El *npi* más alto depende de cada implementación. Algunos valores bastante comunes son 6, 7, 15 y 31. Si llega una interrupción con un *npi* menor que el *npi* actual, la interrupción es bloqueada hasta que el *npi* del sistema se reduzca a un nivel inferior al *npi* de la interrupción pendiente. De esta forma el sistema establece una prioridad a la hora de atender las interrupciones.

Cada dispositivo interrumpe siempre con un mismo *npi*; usualmente, todos los dispositivos conectados a un mismo controlador tienen el mismo *npi*. Cuando el núcleo trata una interrupción, primero configura el *npi* del sistema al valor del *npi* de la interrupción, para así bloquear las posibles interrupciones adicionales de ese dispositivo, así como las de otros con un *npi* inferior. Además, algunas rutinas del núcleo elevan el *npi* temporalmente para bloquear ciertas interrupciones. Por ejemplo, la rutina que manipula las colas de dispersión de la caché de buffers de bloques de disco eleva el *npi* para bloquear las interrupciones del disco. Ya que de otra forma, una interrupción del disco podría ocurrir mientras la cola se encuentra en un estado inconsistente, confundiendo por tanto al driver del disco.

El núcleo utiliza un conjunto de rutinas para manipular el *npi*. Por ejemplo, `spltty()` eleva el *npi* hasta el *npi* asignado a la interrupción de un terminal. La rutina `splx()` disminuye el *npi* al valor de *npi* anteriormente almacenado. Estas rutinas son usualmente implementadas como macros por motivos de eficiencia.

Usualmente todas las interrupciones invocan a una rutina común en el núcleo y le pasan alguna información que identifica a dicha interrupción. Esta rutina salva el contexto

a nivel de registros, eleva el *npi* del sistema al mismo *npi* que el de la interrupción e invoca al manipulador de la interrupción. Cuando el manipulador finaliza, se restaura el *npi* al valor que tenía anteriormente y se restaura el contexto del proceso.

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 apropiada.

El tratamiento de las interrupciones es una de las tareas más importantes del sistema, por ello un manipulador se ejecuta preferentemente a cualquier proceso de usuario o del sistema. Puesto que el manipulador interrumpe todas las otras actividades (excepto las interrupciones de mayor *npi* al suyo), debe ser extremadamente rápido. La mayoría de las implementaciones de UNIX no permiten que los manipuladores duerman. Si un manipulador necesita un recurso que podría estar retenido, debe intentar adquirirlo de un modo no bloqueante.

Estas consideraciones influyen el trabajo que el manipulador debe hacer. En primer lugar, su código debe ser de pequeño y rápido de ejecutar, en consecuencia debe hacer lo mínimo posible. Por otra parte, debe asegurarse que el dispositivo no se encuentra ocioso en una situación de carga pesada. Por ejemplo, cuando una operación de E/S se completa, el disco interrumpe al sistema. El manipulador debe notificar al núcleo los resultados de la operación. También debe iniciar la siguiente operación de E/S si existía alguna petición pendiente. En caso contrario, el disco permanecería ocioso hasta que el núcleo recuperara el control y comenzase la siguiente petición.

Aunque estos mecanismos son comunes en un gran número de distribuciones de UNIX, distan mucho de ser universales. Solaris 2.x, por ejemplo, va más allá del uso de *npi* (salvo en un número pequeño de casos) y utiliza hebras para tratar las interrupciones. Asimismo, permite que estas hebras se bloqueen si fuera necesario.

10.3 DRIVERS DE DISPOSITIVOS

10.3.1 Clasificación de los dispositivos y de los drivers

El subsistema de E/S gestiona la parte independiente del dispositivo de todas las operaciones de E/S. Requiere una vista de alto nivel del funcionamiento de los dispositivos. Desde esta perspectiva, un dispositivo es una caja negra que soporta un conjunto estándar de operaciones. Cada dispositivo implementa estas operaciones de

forma diferente, pero el subsistema de E/S no es consciente de ello. En términos de programación orientada a objetos, la interfaz del driver forma una clase base abstracta, cada driver es una subclase o implementación especifica de la clase base. En la práctica, una única interfaz no es apropiada para todos los dispositivos, puesto que varían bastante en cuanto al método de acceso y funcionalidad. Así, UNIX divide a los dispositivos en *dispositivos modo bloque* y *dispositivos modo carácter*. Existiendo una interfaz para cada una.

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 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.

No todos los dispositivos caen claramente en una de estas dos categorías. En UNIX, cada dispositivo que no tiene las propiedades de un dispositivo modo bloque es clasificado como de modo carácter. Algunos dispositivos no tienen ninguna E/S en absoluto. El reloj del hardware, por ejemplo, es un dispositivo cuyo trabajo es simplemente interrumpir a la CPU a intervalos de tiempo fijos, típicamente 100 veces por segundo.

Un driver no tiene porque controlar un dispositivo físico. Es posible simplemente usar la interfaz del driver para suministrar una funcionalidad especial. El driver `men`, por ejemplo, permite a los usuarios leer o escribir en posiciones de memoria principal. El dispositivo *null* es un sumidero de bits, es decir, solamente deja escribir y simplemente se deshace de todos los datos que se escriben en él. A tales dispositivos se les denomina *pseudodispositivos*.

La mayoría de las distribuciones UNIX modernas soportan una tercera clase de drivers, denominados drivers STREAMS que típicamente controlan las interfaces de red y los terminales. En las distribuciones UNIX clásicas estos elementos eran controlados con drivers de carácter. Por motivos de compatibilidad la interfaz de los drivers STREAMS se deriva de la de los drivers de carácter.

10.3.2 Invocación del código del driver

El núcleo puede invocar a un driver de dispositivo por varios motivos:

- *Configuración.* El núcleo llama al driver cuando se arranca el sistema para comprobar e inicializar el dispositivo.
- *Entrada/Salida*. El subsistema de E/S llama al driver para escribir o leer datos.
- *Control*. El usuario puede hacer peticiones de control tales como la apertura o cierre de un dispositivo o el rebobinado de una cinta magnética.
- *Interrupciones.* El dispositivo genera interrupciones una vez que se ha completado una operación de E/S o se produce algún cambio en el estado del dispositivo.

Las funciones de configuración son llamadas una única vez, cuando el sistema arranca. La funciones de entrada/salida y control son operaciones síncronas. Son invocadas en respuesta a peticiones de usuario especificas y se ejecutan en el contexto del proceso invocador. La rutina `d_strategy` del driver de modo bloque es una excepción a esta norma. Las interrupciones son eventos asíncronos, el núcleo no puede predecir cuando ocurrirán y se ejecutan en el contexto de cualquier proceso.

Esto sugiere dividir el driver en dos partes:

- *Parte superior del driver*. Contiene las rutinas síncronas. Se ejecutan en el contexto del proceso. Pueden acceder al espacio de direcciones y al área U del proceso invocador y pueden poner al proceso a dormir si fuese necesario
- *Parte inferior del driver*. Contiene las rutinas asíncronas. Se ejecutan en el contexto del sistema y usualmente no tienen ninguna relación con el proceso actualmente en ejecución y en consecuencia no pueden acceder al espacio de direcciones de dicho proceso o a su área U. Además, no pueden poner a dormir a ningún proceso puesto que podrían bloquear a un proceso no relacionado.

Las dos partes del driver necesitan sincronizar sus actividades. Si un objeto es accedido por ambas partes, entonces las rutinas de la parte superior deben bloquear las interrupciones (mediante la elevación del *npi*) mientras manipulan el objeto. En caso contrario, el dispositivo podría interrumpir mientras el objeto se encuentra en un estado inconsistente, con lo que el resultado sería impredecible.

10.3.3 Los conmutadores de dispositivos

Un *conmutador de dispositivo* es una estructura de datos que define puntos de entrada para cada dispositivo que debe soportar. Existen dos tipos de conmutadores: `struct bdevsw` para dispositivos modo bloque y `struct cdevsw` para dispositivos modo carácter. Su definición típica es la siguiente:

```
struct bdevsw{
   int (*d_open)();
   int (*d_close)();
   int (*d_strategy)();
   int (*d_size)();
   int (*d_xhalt)();
   ...
} bdevsw[];
struct cdevsw{
   int (*d_open)();
   int (*d_close)();
   int (*d_read)();
   int (*d_write)();
   int (*d_ioctl)();
   int (*d_mmap)();
   int (*d_segmap)();
   int (*d_xpoll)();
   int (*d_xhalt)();
   struct streamtab* d_str;
   ...
} cdevsw[];
```

El núcleo mantiene un array separado para cada tipo de conmutador, cada driver de dispositivo tiene una entrada en el array apropiado. Si un driver suministra una interfaz modo bloque y otra modo carácter, dispondrá de una entrada en cada array.

El conmutador define la interfaz abstracta. Cada driver suministra la implementación especifica de estas funciones. Cuando el núcleo desea realizar una acción sobre un dispositivo, localiza el driver en la tabla de conmutadores e invoca la función apropiada

del driver. Por ejemplo, para leer datos desde un dispositivo de modo carácter, el núcleo invoca la función `d_read()` del dispositivo. En el caso del driver de un terminal, éste referenciaría a una rutina llamada `ttread()`.

Los drivers de dispositivos siguen una convención estándar para nombrar a las funciones del conmutador. Cada driver utiliza una abreviatura de dos letras para describirse a si mismo. Ésta es un prefijo para cada una de sus funciones. Por ejemplo, el driver de disco utiliza el prefijo `dk` y nombra a sus rutinas como `dkopen()`, `dkclose()`, `dkstrategy()` y `dksize()`.

Un driver puede no soportar todos los puntos de entrada. Por ejemplo, una impresora en línea no permite normalmente leer. Para ese punto de entrada, el driver puede usar la rutina global `nodev()`, que simplemente retorna el código de error `ENODEV`. Para algunos puntos de entrada, el driver puede desear no tener ninguna acción. Por ejemplo, muchos dispositivos no suministran ninguna acción especial cuando su cerrados. En dicho caso, el driver puede usar la rutina global `nulldev()`, que simplemente devuelve el valor `0` (que indica éxito).

Como se ha mencionado con anterioridad, los drivers STREAMS son nominalmente tratados y accedidos como drivers de dispositivo modo carácter. Se identifican con el campo `d_str`, que vale `NULL` para los drivers de dispositivos de modo carácter ordinarios. Para un driver STREAMS, este campo apunta a la estructura `streamtab`, que contiene punteros a funciones y datos específicos del STREAMS.

10.3.4 Puntos de entrada de un driver

A continuación se describen las funciones de dispositivo accedidas a través del conmutador de dispositivo:

- `d_open()`. Se invoca cada vez que el dispositivo es abierto y puede traer dispositivos en línea o inicializar estructuras de datos. Los dispositivos que requieren acceso exclusivo (tales como impresoras o unidades de cinta) pueden activar un indicador cuando son abiertos y desactivar dicho indicador cuando son cerrados. Si el indicador ya se encuentra activado, `d_open()` puede bloquearse o fallar. Es común tanto a los dispositivos modo bloque como modo carácter.

- `d_close()`. Se invoca cuando se libera la última referencia al dispositivo, es decir, cuando ningún proceso tiene este dispositivo abierto. Puede apagar el dispositivo o dejarlo fuera de línea. Un driver de una unidad de cinta puede

rebobinar la cinta. Es común tanto a los dispositivos modo bloque como modo carácter.

- `d_strategy()`. Punto de entrada común para peticiones de lectura o escritura a un dispositivo modo bloque. Se llama así ya que el driver puede usar alguna estrategia para reordenar las peticiones pendientes con objeto de optimizar el rendimiento del dispositivo. Opera asíncronamente, si el dispositivo está ocupado esta rutina simplemente coloca en una cola la petición y retorna. Cuando la operación de E/S se completa, el manipulador de la interrupción quitará de la cola la siguiente petición e iniciará la siguiente operación de E/S.

- `d_size()`. Es utilizado por los discos para determinar el tamaño de una partición.

- `d_read()`. Lee datos de un dispositivo modo carácter.

- `d_write()`. Escribe datos en un dispositivo modo carácter.

- `d_ioctl()`. Punto de entrada genérico para operaciones de control sobre un dispositivo modo carácter. Cada driver puede definir un conjunto de comandos e invocarlos mediante la interfaz `ioctl`. Los argumentos de esta función incluyen `cmd`, un entero que especifica que comando ejecutar; y `arg` un puntero a un conjunto de argumentos específicos del comando. Se trata de un punto de entrada bastante versátil que soporta operaciones arbitrarias sobre el dispositivo.

- `d_segmap()`. Traduce la memoria del dispositivo en una dirección del espacio de direcciones del proceso. Es utilizado por dispositivos modo carácter traductores de memoria para configurar la traducción en respuesta a la llamada al sistema `mmap`.

- `d_mmap()`. No es utilizado si se suministra la rutina `d_segmap()`. Si `d_segmap` es `NULL`, la llamada al sistema `mmap` sobre un dispositivo modo carácter invoca a `spec_segmap()`, que cuando retorna llama a `d_mmap()`. Comprueba si el desplazamiento en el dispositivo es válido y devuelve la dirección virtual correspondiente.

- `d_xpoll()`. Encuesta al dispositivo para comprobar si ha ocurrido algún evento de interés. Puede ser usado para comprobar si un dispositivo está listo para leer o escribir sin bloquearlo, si ha ocurrido alguna condición de error, etc.

- `d_xhalt()`. Apaga el dispositivo controlado por el driver. Es invocado durante el apagado del sistema o cuando se descarga un driver desde el núcleo.

Salvo `d_xhalt()` y `d_strategy()`, todas las demás son rutinas de la parte superior del driver.

Los puntos de entrada de un driver para manipulación de interrupciones e inicialización no suelen accederse a través de la tabla del conmutador. De hecho, son especificados en un fichero de configuración maestro, que es usado para construir el núcleo. Este fichero contiene una entrada por cada controlador y driver. La entrada también contiene información como por ejemplo el *npi*, el número del vector de interrupción y la dirección base de los RCE para el driver. Los contenidos específicos y el formato de este fichero son diferentes para cada distribución de UNIX.

SVR4 define dos rutinas de inicialización para cada driver: `init` y `start`. Cada driver registra estas rutinas en los arrays `io_init[]` y `io_start[]`, respectivamente. El código de arranque del sistema invoca todas las funciones `init` antes de inicializar el núcleo y todas las funciones `start` después de que el núcleo es inicializado.

10.4 EL SUBSISTEMA DE ENTRADA/SALIDA

El subsistema de E/S es la porción del núcleo que controla la parte independiente del dispositivo de las operaciones de E/S e interactúa con los drivers de dispositivos para tratar la parte dependiente del dispositivo. Es también responsable de nombrar y proteger a los dispositivos. Además se encarga de suministrar a las aplicaciones de usuario una interfaz consistente para todos los dispositivos.

10.4.1 Número principal y número secundario de un dispositivo

El *espacio de nombres de dispositivos* describe cuantos dispositivos diferentes son identificados y referenciados. El *espacio de nombres del hardware* identifica a los dispositivos por el controlador al que se encuentra conectados y el número lógico de dicho controlador. El núcleo utiliza un esquema numérico para nombrar a los dispositivos. Los usuarios requieren un espacio de nombre simple y familiar y utilizan las rutas de acceso de los sistemas de ficheros con este propósito. El subsistema de E/S define la semántica de los espacios de nombres del núcleo y del usuario, además realiza la traducción entre ellos.

El núcleo identifica cada dispositivo mediante el tipo de dispositivo (bloque o carácter), más un par de números denominados *número principal (mayor number)* y

número secundario (minor number) del dispositivo. El número principal del dispositivo identifica el tipo de dispositivo, o más específicamente, el driver. El número secundario del dispositivo identifica la instancia especifica 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 número principal es el índice de dicho driver en la tabla de conmutación apropiada. En el ejemplo anterior, si el núcleo desea invocar una operación `open` de un driver de disco, localiza la entrada número 5 de `bdevsw[]` y llama a su función `d_open[]`. Normalmente, los números principal y secundario son combinados dentro de una misma variable de tipo `dev_t`. Los bits más significativos contienen el número principal, mientras que los bits menos significativos contienen el número secundario. Las macros `getmajor()` y `getminor()` permiten extraer estos números de la variable donde están almacenados.

El núcleo pasa el número de dispositivo como un argumento de la rutina `d_open()` del driver. El driver del dispositivo mantiene varias tablas internas para traducir el número secundario a un RCE especifico o a un número de puerto del controlador. Extrae el número secundario de `dev_t` y lo utiliza para acceder al dispositivo correcto.

Un mismo driver puede ser configurado con varios números principales. Esto resulta útil si el driver gestiona diferentes tipos de dispositivos que realizan algún procesamiento común. Asimismo, un mismo dispositivo puede ser representado por varios números secundarios. Por ejemplo, una unidad de cinta puede usar un número secundario para seleccionar un modo de autorebobinado y otro para un modo de no-rebobinado. Finalmente, si un dispositivo tiene tanto una interfaz de modo bloque como de modo carácter, utilizará entradas distintas en ambas tablas de conmutación con números principales distintos para acceder a cada una.

En las primeras distribuciones de UNIX, `dev_t` tenía un campo de 16 bits, con 8 bits para el número principal y 8 bits para el número secundario. Esto imponía un límite de 256 números secundarios por cada número principal, lo cual resultaba demasiado restrictivo para algunos sistemas. Para evitar esto, los drivers utilizaban varios números principales.

Otro problema es que las tablas de conmutación pueden hacerse de un tamaño muy grande si contienen entradas por cada dispositivo posible, incluyendo aquellos que no están conectados al sistema o aquellos cuyos drivers no están enlazados al núcleo. Esto sucedía porque los fabricantes no deseaban ajustar la tabla de conmutación para cada configuración diferente que fabricaban y tendían a colocar toda la información posible en las tablas de conmutación.

SVR4 realizó varios cambios para resolver este problema. El tipo `dev_t` fue aumentado a 32 bits, normalmente divido en 14 bits para un número principal y 18 para un número secundario. También introdujo la noción de números de dispositivos externos e internos. Los *números de dispositivos internos* identifican al driver y sirven como índice dentro de la tabla de conmutación. Los *números de dispositivos externos* forman la representación de un dispositivo visible para el usuario. y son almacenados en el campo `i_rdev` del nodo-i de un fichero especial de dispositivo.

En muchos sistemas, como por ejemplo Intel x86, los números externos e internos son idénticos. En otros que soportan autoconfiguración, como AT&T 3B2, estos números son distintos. En estos sistemas, `bdevsw[]` y `cdevsw[]` son construidos dinámicamente cuando el sistema arranca y sólo contienen entradas para los drivers que están configurados en el sistema. El núcleo mantiene un array llamado `MAJOR[]`, el cual es indexado mediante el número principal externo. Cada elemento de este array almacena el número principal interno correspondiente.

10.4.2 Ficheros de dispositivos

El par <*número principal*, *número secundario*> proporciona al núcleo un espacio de nombres de dispositivo simple y efectivo. A nivel de usuario, sin embargo, es bastante inútil, ya que los usuarios no desean recordar un par de números por cada dispositivo. Además, los usuarios desean usar las mismas aplicaciones y comandos para leer y escribir tanto ficheros ordinarios como dispositivos. La solución natural es usar el espacio de nombres del sistema de ficheros para describir tanto a los ficheros ordinarios como a los dispositivos.

UNIX suministra una interfaz consistente a los ficheros y a los dispositivos a través de la introducción de la noción de *fichero de dispositivo*. Este es un fichero especial localizado en cualquier parte del sistema de ficheros y asociado con un dispositivo especifico. Por convención, todos los ficheros de dispositivos son mantenidos en el directorio `/dev` o en un subdirectorio del mismo.

Desde el punto de vista de un usuario, un fichero de dispositivo no es muy diferente de un fichero ordinario. No tiene bloques de datos en el disco pero tiene un nodo-i permanente en el sistema de ficheros en el cual está localizado (normalmente, el sistema de ficheros raiz). El campo `di_mode` del nodo-i muestra de que tipo es cada uno: `IFBLK` (para dispositivos modo bloque) o `IFCHR` (para dispositivos modo carácter). En vez de la lista de números de bloques, el nodo-i contiene un campo llamado `di_rdev` que almacena los números principal y secundario del dispositivo al que representa. Esto permite al núcleo traducir el nombre del fichero a nivel de usuario (ruta de acceso del fichero) al nombre interno del dispositivo (el par <*número principal*, *número secundario*>).

Un fichero de dispositivo no se puede crear de la forma usual. Sólo el superusuario puede crear un fichero de dispositivo usando la llamada al sistema

```
mknod(path, mode, dev)
```

donde `path` es la ruta de acceso del fichero especial, `mode` especifica el tipo de fichero (`IFBLK` o `IFCHR`) y los permisos. `Dev` es el número que combina el número principal y el número secundario. Esta llamada al sistema crea un fichero especial e inicializa los campos `di_mode` y `di_rdev` a partir de los argumentos.

Unificar los espacios de nombres de ficheros y dispositivos tienen grandes ventajas. La E/S con los dispositivos utilizan el mismo conjunto de llamadas al sistema que la E/S con un fichero. Así los programadores pueden escribir aplicaciones sin preocuparse sobre si la entrada y la salida es sobre un dispositivo o sobre un fichero. Los usuarios ven una vista consistente del sistema y pueden usar cadenas de caracteres descriptivas como nombres para referenciar a los ficheros.

Otro beneficio importante es el control de acceso y la protección. Cada fichero de dispositivo tiene asignados los permisos estándar de lectura/escritura/ejecución para el propietario, el grupo y otros usuarios. Estos permisos son inicializados y modificados de la forma usual, tal y como se hace en los ficheros. Típicamente, algunos dispositivos tales como los discos son directamente accesibles únicamente por el superusuario, mientras que otros como las unidades de cinta puede ser accedidas por todos los usuarios.

10.4.3 El sistema de ficheros specfs

Cómo se estudio en el Capítulo 8 UNIX dispone de la interfaz nodo-v/sfv que permite tener diferentes tipos de sistemas de ficheros en el mismo núcleo. Esta aproximación asocia un objeto del núcleo denominado *nodo-v* con cada fichero abierto. La interfaz define un conjunto de operaciones abstractas en cada nodo-v. Cada sistema de ficheros

suministra su propia implementación de estas funciones. Por ejemplo, el nodo-v de fichero de un sistema *ufs* apunta a un vector llamado `ufsops` que contiene punteros a las funciones del sistema *ufs* tales como `ufslookup()`, `ufsclose()` y `ufslink()`.

Es necesario una forma especial de tratar los ficheros de dispositivos. Un fichero de dispositivo reside en el sistema de ficheros raíz el cual, para el propósito de la siguiente discusión, se va a considerar que es un sistema *ufs*. Por tanto su nodo-v es un nodo-v *ufs* y apunta a `ufsops`. Cualquier operación sobre este fichero será tratado mediante las funciones del sistema *ufs*.

Esta forma de proceder, no obstante, no es la más correcta. El fichero de dispositivo no es un fichero ordinario, sino un fichero especial que representa un dispositivo. Todas las operaciones en el fichero deben ser implementadas mediante la correspondiente acción sobre el dispositivo, usualmente a través del conmutador de dispositivos. Por lo tanto, se necesita una forma de traducir todos los accesos del fichero de dispositivo al dispositivo que subyace.

SVR4 utiliza un tipo de sistema de ficheros especial llamado *specfs*. Implementa todas las operaciones a los nodos-v buscando en el conmutador de dispositivo e invocando a la función apropiada. El nodo-v *spefcs* tiene una estructura de datos privada llamada *nodo-s* (de hecho, el nodo-v es parte del nodo-s). El término nodo-s viene de node shadow (nodo ensombrecido). El subsistema de E/S debe asegurarse que, cuando un usuario abre un fichero de dispositivo, adquiere una referencia al nodo-v *specfs* y que todas las operaciones sobre el fichero son encaminadas hacia él.

♦ **Ejemplo 10.1:**

Supóngase que un usuario desea abrir el fichero del fichero `/dev/lp`. El directorio `/dev` se encuentra en el sistema de ficheros raíz, que se supone que es del tipo *ufs*. La llamada al sistema `open` traduce la ruta de acceso usando repetidamente la rutina `ufs_lookup()`, en primer lugar localiza el nodo-v del directorio `dev`. Después el nodo-v de `lp`. Cuando `ufs_lookup()` obtiene el nodo-v para `lp`, descubre que el tipo de fichero es `IFCHR`. Entonces, extrae del nodo-i los números principal y secundario del dispositivo y se los pasa a la rutina llamada `specvp()`.

El sistema de ficheros *specfs* mantiene todos los nodos-s en una tabla de dispersión, indexada por los números de dispositivos. `specvp()` busca en la tabla de dispersión y sino encuentra el nodo-s, crea un nuevo nodo-s y un nuevo nodo-v.

El nodo-s tiene un campo llamado `s_realvp`, en el cual `specvp()` almacena un puntero al nodo-v de `/dev/lp`. Finalmente, devuelve un puntero al nodo-v *specfs*. El nodo-v *specfs* oculta el nodo-v de `/dev/lp` y su campo `v_op` apunta al vector de las operaciones *specfs* (como por ejemplo

`spec_read()` y `spec_write()`), que a su vuelta llama a los puntos de entrada del dispositivo. En la Figura 10.3 se ilustra la configuración resultante.

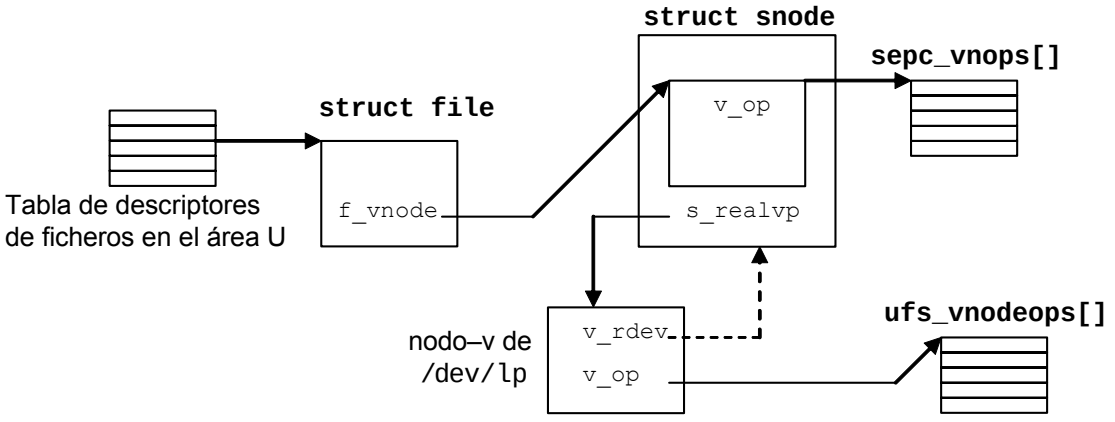


Figura 10.3: Estructuras de datos después de abrir `/dev/lp`

Antes de retornar, `open` invoca la operación `VOP_OPEN` sobre el nodo-v, la cual llama a `spec_open()` en el caso de un fichero de dispositivo. La función `spec_open()` llama a la rutina `d_open()` del driver, la cual realiza los pasos necesarios para abrir el dispositivo.

♦

10.4.4 El nodo-s común

El sistema *specfs* tal y como se ha descrito hasta ahora está bastante incompleto y dista de ser correcto. Se ha supuesto una relación uno a uno entre los ficheros de dispositivos y los dispositivos que subyacen. En la práctica, es posible tener varios ficheros de dispositivos, cada uno representando al mismo dispositivo (sus campos `di_rdev` tendrán el mismo valor). Estos ficheros pueden estar en el mismo o en diferentes sistemas de ficheros.

Esto crea varios problemas. La operación de dispositivo `close`, por ejemplo, debe ser invocada solamente cuando el último descriptor del dispositivo es cerrado. Supóngase que dos procesos abren el dispositivo usando diferentes ficheros de dispositivo. El núcleo debería ser capaz de reconocer esta situación y llamar a la operación `close` del dispositivo solamente después de que ambos ficheros sean cerrados.

Otro problema está relacionado al direccionamiento de páginas. En SVR4, el nombre de una página en memoria está definido mediante el nodo-v que pertenece a la página y el desplazamiento de la página en el fichero.

Para una página asociada con un dispositivo, el nombre es ambiguo si múltiples ficheros se refieren al mismo dispositivo. Dos procesos accediendo al dispositivo a través de diferentes ficheros de dispositivo podrían crear dos copias de la misma página en memoria, produciendo un problema de consistencia de datos.

Cuando se tienen varios nombres de ficheros para un mismo dispositivo, se pueden clasificar las operaciones sobre el dispositivo en dos grupos. La mayoría de las operaciones son independientes del nombre del fichero utilizado para acceder al dispositivo. Así pueden ser canalizadas a través de un objeto común. Al mismo tiempo, existen unas pocas operaciones que dependen del fichero utilizado para acceder al dispositivo. Por ejemplo, cada fichero puede tener un propietario y unos permisos diferentes; por lo tanto, es importante mantener la pista del nodo-v "real" (el del fichero del dispositivo) y encaminar todas las operaciones hacia él.

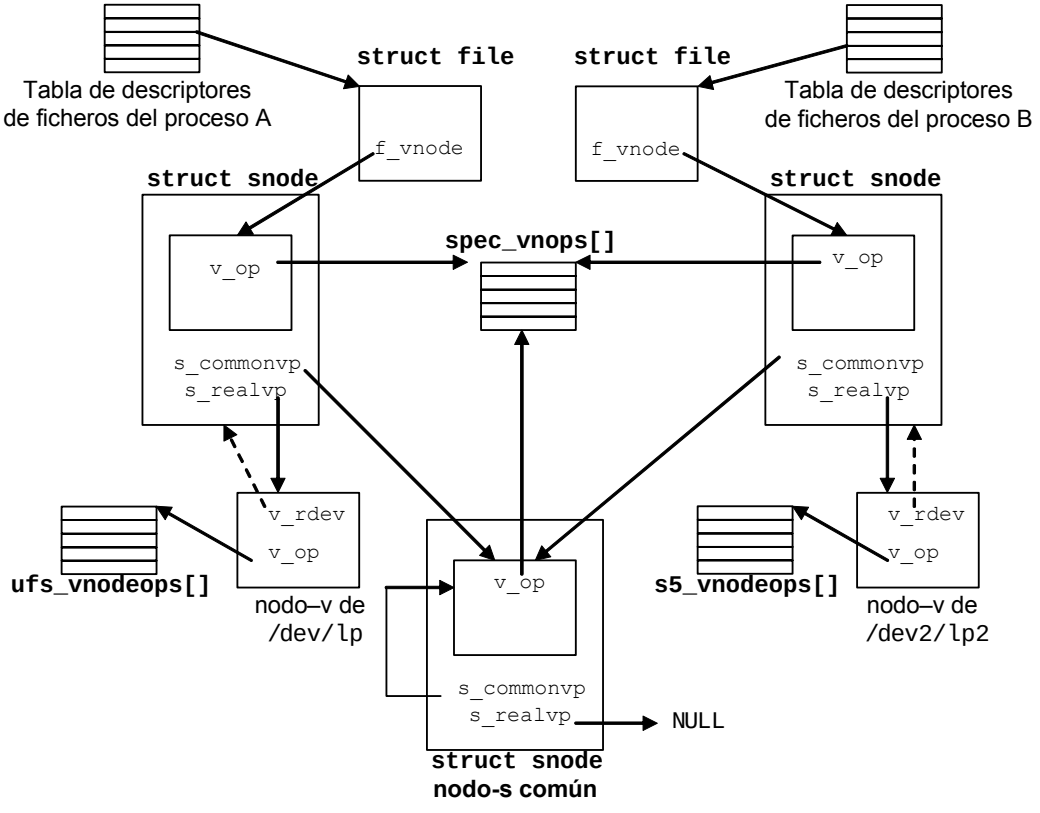


Figura 10.4: El nodo-s común

El sistema de ficheros *specfs* utiliza la noción de *nodo-s común* para permitir ambos tipos de operaciones. La Figura 16.4 describe las estructuras de datos. Cada dispositivo tiene solamente un nodo-s común, creado cuando se accede al dispositivo por primera vez. Existen también un nodo- s por cada fichero de dispositivo. Los nodos-s de todos los ficheros representando al mismo dispositivo comparten el nodo-s común y referencian a él a través del campo `s_commonvp`.

La primera vez que un usuario abre un fichero de dispositivo para un dispositivo en particular, el núcleo crea un nodo-s y un nodo-s común. Posteriormente, si otro usuario abre el mismo fichero, compartirá estos objetos. Si un usuario abre otro fichero que representa al mismo dispositivo, el núcleo creará un nuevo nodo-s, que referenciará al nodo-s común a través del campo `s_commonvp`. El nodo-s común no está directamente asociado con un fichero de dispositivo; por lo tanto, su campo `s_realvp` es `NULL`. Su campo `s_commonvp` apuntará a si mismo.

10.5 STREAMS

10.5.1 Motivación

El entorno tradicional de los driver de dispositivos tiene muchos inconvenientes. En primer lugar, el núcleo interactúa con los drivers a alto nivel (los puntos de entrada del driver), haciendo al driver responsable de la mayoría del procesamiento de una petición de E/S. Los drivers de dispositivos son normalmente escritos independientemente por el fabricante del dispositivo. Muchos fabricantes escriben drivers para el mismo tipo de dispositivo. Solo parte del código del driver es dependiente del dispositivo, el resto implementa el procesamiento de la E/S de alto nivel independiente del dispositivo. Como resultado, estos drivers duplican muchas de sus funcionalidades, creando un núcleo mucho más grande y una mayor posibilidad de conflicto.

Otro inconveniente se encuentra en el área de almacenamiento temporal. La interfaz de los dispositivos modo bloque suministra un apoyo razonable para la asignación y gestión del espacio de almacenamiento temporal (buffers). No obstante, no existe este esquema uniforme para los dispositivos modo carácter. La interfaz de los dispositivos modo carácter fue diseñada originalmente para soportar dispositivos lentos que leyeran o escribieran un carácter a la vez. Por lo tanto el núcleo suministraba un apoyo mínimo para el almacenamiento temporal, delegando esta responsabilidad a cada dispositivo. Esto resultó en el desarrollo de varios esquemas de gestión de memoria y almacenamiento temporal de datos que producían un uso ineficiente de la memoria y una duplicación de código.

Finalmente, la interfaz suministrada limita las funcionalidades de las aplicaciones. La E/S a los dispositivos modo carácter requiere las llamadas al sistema `read` y `write`, las cuales tratan los datos como un flujo de bytes de tipo FIFO. No hay soporte para reconocer los límites de un mensaje, distinguir entre datos regulares e información de control, o asociar prioridades a los diferentes mensajes. Asimismo, tampoco existe control del flujo, cada driver y cada aplicación tiene sus propios mecanismos para resolver este tema.

Los requerimientos de los dispositivos en red pusieron de manifiesto estas limitaciones. Los protocolos de red son diseñados en capas. Los datos transferidos son mensajes o paquetes, cada capa del protocolo realiza algún procesamiento sobre el paquete y después se lo pasa a la siguiente capa. Los protocolos distinguen entre datos ordinarios y datos urgentes. Las capas contienen partes intercambiables y un protocolo dado puede combinarse con diferentes protocolos en otras capas. Esto sugiere un entorno modular que soporte esta estructuración en capas y permita que los drivers sean construidos mediante la combinación de varios módulos independientes.

El subsistema de STREAMS resuelve la mayoría de los problemas que se han expuesto. Suministra una aproximación modular a la escritura de drivers. Tiene una interfaz completamente basada en mensajes que contienen las funcionalidades necesarias para la gestión del almacenamiento temporal, el control de flujo y la planificación basada en prioridades. Soporta protocolos en capas. Promueve la compartición de código puesto que cada stream está compuesto de varios módulos reutilizables que pueden ser compartidos por diferentes drivers. Ofrece funcionalidades adicionales a las aplicaciones a nivel de usuario para transferencia basada en mensajes y separación de los datos y la información de control.

Originariamente desarrollados por Dennis Richie, los STREAMS están ahora soportados por la mayoría de las distribuciones de UNIX y se han convertido en la interfaz preferida para escribir driver de red y protocolos. Adicionalmente, SVR4 también utiliza STREAMS para sustituir a los drivers de terminales tradicionales y para la implementación de tuberías.

Otra de las ventajas de los STREAMS es que ofrecen una forma simple de implementar ficheros FIFO y tuberías. Asimismo también suministran al núcleo la infraestructura necesaria para trabajar en red. Los programadores necesitaban una interfaz de alto nivel para escribir aplicaciones de red.

El entorno de conectores (sockets), introducidos en 1982 en la distribución BSD4.1, proporcionaba un apoyo global para la programación en red. El UNIX System V trató este problema mediante un conjunto de interfaces en capas apoyadas en STREAMS. Entre las que se incluían la interfaz TPI (Transport Provider Interface), que definía las interacciones entre los suministradores del transporte y los usuarios del transporte. Así como la interfaz TLI (Tranport Layer Interface) que proporcionaba funcionalidades de programación a alto nivel. Puesto que los conectores aparecieron mucho antes que los STREAMS, hay un gran número de aplicaciones que los utilizan. Para facilitar la

portabilidad de estas aplicaciones; SVR4 añadió soporte para conectores mediante una colección de librerías y módulos STREAMS.

10.5.2 Consideraciones generales

Un *stream* es un procesamiento full-duplex y un camino de transferencia de datos entre un driver en el espacio del núcleo y un proceso en el espacio de usuario. STREAMS es una colección de llamadas al sistema, recursos del núcleo y rutinas de utilidad del núcleo que crean, usan y desmontan un stream. Es también un entorno para escribir drivers de dispositivos. Y suministrar los mecanismos y utilidades que permiten a tales drivers ser desarrollados de una manera modular.

La Figura 10.5 describe un stream típico. Un stream reside por completo en el espacio del núcleo y sus operaciones son implementadas en el núcleo. Consta de la cabeza del stream, el extremo del driver y cero o varios módulos entre ellos. La *cabeza del stream* sirve de interfaz con el nivel de usuario y permite a las aplicaciones acceder al stream a través de la interfaz de llamadas al sistema. El *extremo del driver* comunica con el propio dispositivo (alternativamente, puede ser un driver de psudodispositivo, en cuyo caso puede comunicarse con otro stream). Los módulos se encargan del procesamiento intermedio de los datos.

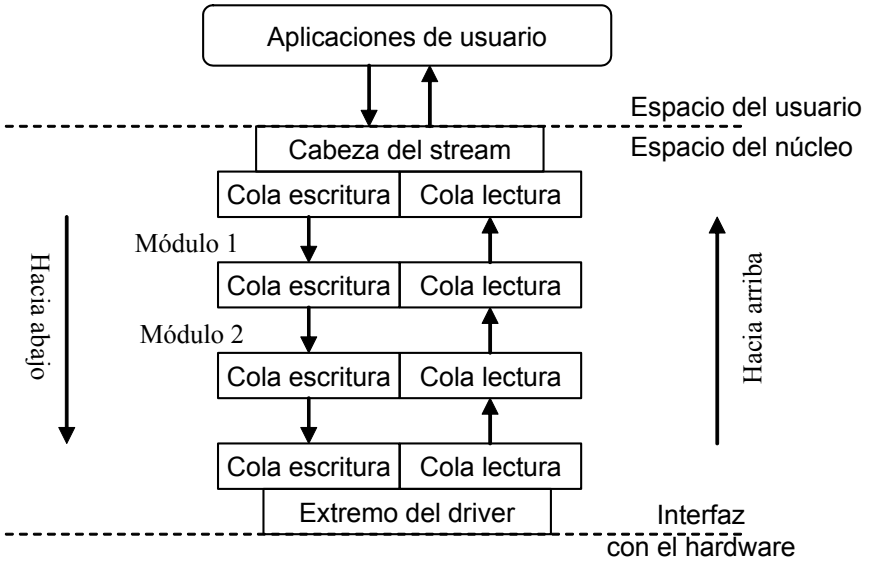


Figura 10.5: Un stream típico.

Cada módulo contiene una cola de lectura y una cola de escritura. Tanto la cabeza del stream como el extremo del driver también contienen estas colas. El stream transfiere datos poniendo en las colas mensajes. Las colas de escritura envían mensajes hacia las partes de abajo del stream, desde la aplicación al driver. Las colas de lectura pasan los

mensajes hacia arriba del stream, desde el driver hacia la aplicación. Aunque la mayoría de los mensajes se originan en la cabeza del stream o en el driver, los módulos intermedios también pueden generar mensajes y pasarlos arriba o abajo del stream.

Cada cola puede comunicarse con la siguiente cola del stream. Por ejemplo, en la Figura 10.5, la cola de escritura del módulo 1 puede enviar mensajes a la cola de escritura del módulo 2 (pero no al contrario). La cola de lectura del módulo 1 puede enviar mensajes a la cola de lectura de la cabeza del stream. Una cola puede también comunicarse con su cola compañera. Así, la cola de lectura del módulo 2 puede pasar mensajes a la cola de escritura del mismo módulo, la cual podría entonces enviarlo hacia abajo del stream. Una cola no necesita conocer si la cola con la que está comunicándose pertenece a la cabeza del stream, al extremo del driver o a otro módulo intermedio.

Los STREAMS utilizan el paso de mensajes como su única forma de comunicación. Los mensajes transfieren datos entre las aplicaciones y los dispositivos. También transportan información de control al driver o a un módulo. Los módulos y los drivers generan mensajes para informar al usuario, o unos a otros, de condiciones de error o eventos inesperados. Una cola puede tratar un mensaje entrante de varias formas. Puede pasarlo a la siguiente cola, sin modificar o después de realizar algún procesamiento sobre el mismo. La cola puede planificar el mensaje para aplazar su procesamiento. Alternativamente, puede pasar el mensaje a su compañera, para así enviar el mensaje de vuelta en la dirección opuesta. Finalmente una cola puede incluso descartar un mensaje.

Cada módulo puede ser escrito independientemente, quizás por diferentes fabricantes. Los módulos pueden ser mezclados y ajustarse de diferentes formas, de forma análoga a como se combinan varios comandos mediante el uso de tuberías en un intérprete de comandos de UNIX.

♦ **Ejemplo 10.2:**

La Figura 10.6 muestra como diferentes streams pueden ser formados con unos pocos componentes. Un fabricante que desarrolla software para red puede desear añadir el protocolo TCP/IP (Transmission Control Protocol/Internet Protocol) al sistema. Usando STREAMS, tendría que desarrollar un módulo TCP, un módulo UDP (User Datagram Protocol) y un módulo IP. Otros fabricantes que hagan tarjetas de red escribirán independientemente drivers STREAMS para la ethernet y el token ring.

Una vez que estos módulos están disponibles pueden ser configurados dinámicamente para formar diferentes tipos de streams. En la Figura 10.6a, un usuario ha formado un stream TCP/IP que se conecta a un token ring. Mientras que la Figura 10.6b muestra una nueva configuración, con un stream UDP/IP conectado a un driver de ethernet.

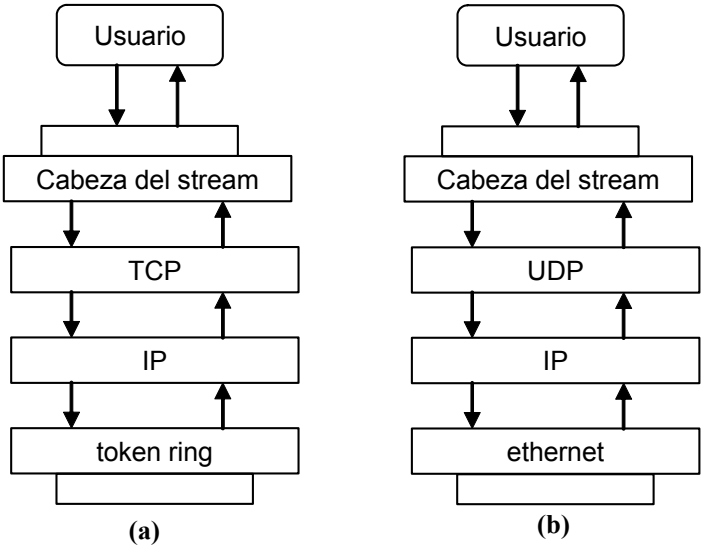


Figura 10.6: Módulos reutilizables

♦

Los STREAMS soportan una funcionalidad llamada multiplexión. Un driver de multiplexión puede conectar múltiples streams en lo alto o en lo bajo. Hay tres tipos de multiplexores:

- *Multiplexor superior o fan-in*. Puede conectar a varios streams sobre él.
- *Multiplexor inferior o fan-out*, Puede conectar múltiples streams por debajo de el.
- *Multiplexor de doble-sentido*. Soporta múltiples conexiones tanto por encima como por debajo de él.

♦ **Ejemplo 10.3:**

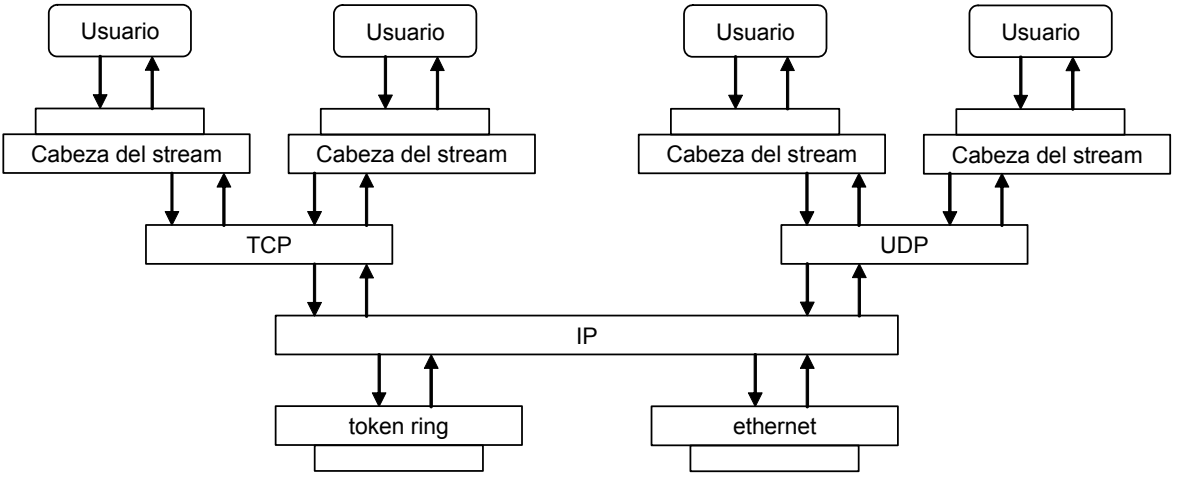


Figura 10.7: Streams multiplexados

Escribiendo los módulos TCP, UDP e IP como drivers multiplexados, se pueden combinar los streams en un único objeto componente que soporta múltiples caminos de datos. La Figura 10.7 muestra una configuración posible. TCP y UDP actúan como multiplexores superiores, mientras que IP sirve como multiplexor de doble sentido. Esto permite a las aplicaciones hacer varios tipos de conexiones de red y habilita a varios usuarios a acceder a cualquier combinación dada de protocolos y drivers. Los drivers multiplexados pueden gestionar todas las diferentes conexiones y encaminar los datos arriba o abajo del stream hacia la cola adecuada.

♦

APÉNDICE

A

Acerca del sistema operativo Linux

A.1 ORÍGENES

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 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 actualidad (`www.kernel.org`), Linux es por completo un sistema basado en UNIX. Es estable, pero continúa 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.

Afortunadamente 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.

A.2 DISTRIBUCIONES DE LINUX

A.2.1 Distribuciones tradicionales

Linux es un sistema de libre distribución por lo que es posible encontrar todos los ficheros y programas necesarios para su funcionamiento en multitud de servidores conectados a Internet. La tarea de reunir todos los ficheros y programas necesarios, así como instalarlos en una computadora y configurarlo, puede ser una tarea bastante complicada y no apta para muchos. Por esto mismo, nacieron las llamadas *distribuciones de Linux*, empresas y organizaciones que se dedican a hacer el trabajo “sucio” para beneficio y comodidad del usuario.

Una *distribución* no es otra cosa, que una recopilación de programas y ficheros, organizados y preparados para su instalación. Estas distribuciones se pueden obtener a través de Internet o comprando los CDs de las mismas, los cuales contendrán todo lo necesario para instalar un sistema operativo Linux bastante completo y en la mayoría de los casos un programa de instalación que ayudará en la tarea de una primera instalación. Casi todos los principales distribuidores de Linux ofrecen la posibilidad de bajarse sus distribuciones vía FTP (sin cargo alguno).

Existen muchas y variadas distribuciones creadas por diferentes empresas y organizaciones a unos precios bastantes asequibles (si se compran los CDs, en vez de bajársela vía FTP), las cuales se deberían poder encontrar en tiendas de informática y librerías especializadas. En el peor de los casos siempre es posible encargarlas directamente por Internet a las empresas y organizaciones que las crean. A veces, las revistas de informática sacan una edición bastante aceptable de alguna distribución.

Si se va a instalar el sistema Linux por primera vez, es recomendable hacerse con una de estas distribuciones y en el futuro usar Internet cuando se desee actualizar el sistema con las últimas versiones del núcleo y programas utilizados

A continuación se enumeran y comentan brevemente las distribuciones mas importantes de Linux:

- REDHAT ENTERPRISE (`http://www.redhat.com/`). Distribución que tiene muy buena calidad, contenidos y soporte a los usuarios por parte de la empresa que la distribuye. Es necesario el pago de una licencia de soporte. Enfocada a empresas.

- FEDORA (`http://fedora.redhat.com/`). Distribución patrocinada por RedHat y soportada por la comunidad de usuarios de Linux. Fácil de instalar y buena calidad.

- DEBIAN (`http://www.es.debian.org/`). Distribución con muy buena calidad. El proceso de instalación es quizás un poco mas complicado, pero sin mayores problemas. Gran estabilidad antes que últimos avances.

- OpenSuSE (`http://www.opensuse.org/`). Versión libre de la distribución comercial SuSE. Fácil de instalar.

- SuSE LINUX ENTERPRISE (`http://www.suse.com/`). Distribución de muy buena calidad, contenidos y soporte a los usuarios por parte de la empresa que la distribuye, Novell. Es necesario el pago de una licencia de soporte. Enfocada a empresas.

- SLACKWARE (`http://www.slackware.com/`). Esta distribución es de las primeras que existió. Es raro encontrar a algún usuario pionero de Linux que no se haya instalado esta distribución alguna vez.

- GENTOO (`http://www.gentoo.org/`). Esta distribución es una de las únicas que han incorporado un concepto totalmente nuevo en Linux. Es una sistema inspirado en BSD-ports. Es posible compilar/optimizar el sistema completamente desde cero. No es recomendable adentrarse en esta distribución sin una buena conexión a Internet, un ordenador medianamente potente y cierta experiencia en sistemas UNIX.

- UBUNTU (`http://www.ubuntu.com/`). Distribución basada en Debian que se caracteriza por estar centrada en el usuario final y su facilidad de uso. Muy

popular y con mucho soporte en la comunidad de usuarios de Linux. El entorno de escritorio por defecto es Gnome.

- KUBUNTU (`http://www.kubuntu.com/`). Distribución basada en Ubuntu que se caracteriza por estar centrada en el usuario final y su facilidad de uso. Muy popular y con mucho soporte en la comunidad de usuarios de Linux. El entorno de escritorio por defecto es KDE.

- MANDRIVA (`http://www.mandrivalinux.org/`). Esta distribución fue creada en 1998 con el objetivo de acercar el uso de Linux a todos los usuarios, en un principio se llamo Mandrake Linux. Facilidad de uso para todos los usuarios.

El elegir una distribución u otra es cuestión de gustos ya que la calidad de todas las distribuciones es alta. Sin embargo de forma general puede afirmarse que Suse y Ubuntu son muy buenas distribuciones para usuarios que no quieran complicarse la vida sin perder la potencia y versatilidad de Linux.

A.2.2 Distribuciones LiveCD

Para aquellos usuarios que desean probar como funciona y se utiliza un sistema Linux, sin necesidad de instalarlo en el disco duro, existen las *distribuciones LiveCD*.

Una *distribución LiveCD* no es otra cosa que una distribución de Linux que funciona al 100%, sin necesidad de instalarla en el ordenador donde se prueba. Utiliza la memoria principal de la computadora para instalar y arrancar la distribución en cuestión. En la memoria también se instala un *disco virtual* que emula al disco duro de un ordenador.

De esta forma solamente hace falta introducir el CD o DVD en la computadora y arrancarlo, al cabo de unos minutos se tendrá un sistema Linux funcionando en el mismo. Este tipo de distribuciones solamente sirve para demostraciones y pruebas, ya que una vez que apagamos el ordenador, todo lo que se haya hecho desaparece.

Algunas distribuciones LiveCD también vienen con una opción de instalación una vez que se ha probado. Existen muchas distribuciones de este tipo, algunas solamente en versión LiveCD, otras como demostraciones de distribuciones que se pueden instalar de la manera tradicional.

A continuación se enumeran y comentan brevemente las distribuciones LiveCD mas importantes de Linux:

- UBUNTU DESKTOP LIVECD (`http://www.ubuntu.com/`). Distribución basada en Debian que se caracteriza por estar centrada en el usuario final y su facilidad de uso. La imagen ISO versión DESKTOP de esta distribución, es del tipo 'LiveCD' con posibilidad de instalación.

- DISTRIBUCION GNOPPIX - LIVECD (`http://www.gnoppix.org/`). Esta distribución está basada en Ubuntu y usa Gnome como gestor de ventanas.

- DISTRIBUCION SuSE LIVE (`http://www.suse.com/`). Versión LiveCD de la distribución SuSE.

- DISTRIBUCION KNOPPIX - LIVECD (`http://www.knopper.net/knoppix/index-en.html`). Distribución LiveCD basada en Debian.

- DISTRIBUCION CENTOS - LIVECD (`http://www.centos.org`). Versión LiveCD de la distribución Centos. Basada en Redhat Enterprise.

- DISTRIBUCION GENTOO - LIVECD (`http://www.gentoo.org`). Versión LiveCD de la distribución Gentoo.

- DISTRIBUCION SLAX - LIVECD (`http://www.slax.org`). Distribución LiveCD basada en Slackware

A.3 INSTALACIÓN DE UNA DISTRIBUCIÓN

Uno de los principales problemas que se le plantea a un usuario cuando quiere empezar a usar Linux, es que no tiene muy claro que es lo que necesita y que pasos debe seguir para instalar y configurar este sistema operativo. Hace unos años el proceso de instalación y configuración de un sistema Linux era un poco complicado para usuarios sin conocimientos. Afortunadamente esto ha cambiando bastante en los últimos años. Ahora casi todas las distribuciones vienen con unos programas de instalación y configuración del sistema muy fáciles de usar para usuarios con conocimientos básicos de informática.

El usuario debe elegir la distribución de Linux que desea instalar y descargársela de Internet. A continuación conviene leerse el manual de instalación antes de empezar el proceso de instalación. Además conviene planificar un poco dicho proceso de instalación. La primera pregunta que conviene hacerse es si Linux va a ser el único sistema operativo que exista en el ordenador. Si es así la instalación será más sencilla. Por el contrario si va a convivir con otros sistemas operativos como por ejemplo Windows, conviene leerse

las secciones del manual que explican como instalar o tener varios sistemas operativos en la computadora junto con Linux.

Asimismo antes de iniciar el proceso de instalación también conviene recopilar toda la información técnica de la computadora: tarjeta gráfica, monitor, etc. Los principales problemas de instalación del sistema Linux son debidos al hardware de la computadora donde se desea instalar el sistema, ya que puede no estar soportado o necesitar un tratamiento especial para funcionar. La mayoría de las distribuciones de Linux tiene documentación sobre el hardware que soportan.

Siguiendo las instrucciones del asistente y del manual de instalación, no se debería tener ningún problema para instalar Linux, siempre que el hardware de la computadora esté soportado. Es en el proceso previo de planificación y en los ajustes posteriores a la instalación, donde quizás se necesite mas ayuda.

Una vez que se haya terminado la instalación y el sistema arranque sin problemas, hay una serie de pasos que se deben seguir. Dependiendo de la distribución que se haya instalado, algunos de estos pasos ya se habrán hecho en el proceso de instalación. Además existen programas gráficos que simplifican bastante su realización. Los pasos en cuestión son:

- Abrir una cuenta de usuario para usar el sistema. El superusuario (root) sólo se debe utilizar para tareas de administración del sistema.
- Hacer funcionar el sistema de ventanas X-Windows. Obviamente es mucho mas cómodo utilizar el sistema en modo gráfico que en modo texto.
- Configurar la conexión a Internet.
- Instalar programas que no vengan con la distribución.

APÉNDICE

B Funciones de biblioteca de uso más frecuente

Función	Tipo de salida	Propósito	Archivo de cabecera
abs(i)	int	Devuelve el valor absoluto de i.	stdlib.h
acos(d)	double	Devuelve el arco coseno de d.	math.h
asin(d)	double	Devuelve el arco seno de d.	math.h
atan(d)	double	Devuelve el arco tangente de d.	math.h
atan(d1,d2)	double	Devuelve el arco tangente de d1/d2.	math.h
atof(s)	double	Convierte la cadena s a una cantidad en doble precisión.	stdlib.h
atoi(s)	int	Convierte la cadena s a un entero.	stdlib.h
atol(s)	long	Convierte la cadena s a un entero largo.	stdlib.h
calloc(u1,u2)	void*	Reserva memoria para un array de u1 elementos, cada uno de u2 bytes. Devuelve un puntero al principio del espacio reservado.	malloc.h stdlib.h
ceil(d)	double	Devuelve un valor redondeado por exceso al siguiente entero mayor.	math.h
cos(d)	double	Devuelve el coseno de d.	math.h
cosh(d)	double	Devuelve el coseno hiperbólico de d.	math.h
difftime(t1,t2)	double	Devuelve la diferencia de tiempo t1-t2, donde t1 y t2 representan el tiempo transcurrido después de un tiempo base.	time.h
exp(d)	double	Eleva el número e (2.7182818...) a la potencia d.	math.h
fabs(d)	double	Devuelve el valor absoluto de d.	math.h
fclose(f)	int	Cierra el fichero f. Devuelve 0 si el archivo se ha cerrado con éxito.	stdio.h
feof(f)	int	Determina si se ha encontrado el fin del archivo f. Si es así, devuelve un valor distinto de cero; en otro caso devuelve 0.	stdio.h
fgetc(f)	int	Lee un carácter del archivo f.	stdio.h
fgets(s,i,f)	char*	Lee una cadena s, con i caracteres, del archivo f.	stdio.h
floor(d)	double	Devuelve un valor redondeado por defecto al entero menor más cercano.	math.h

Función	Tipo de salida	Propósito	Archivo de cabecera
`fopen(s1,s2)`	`file*`	Abre un archivo llamado `s1`, del tipo `s2`. Devuelve un puntero al archivo.	`stdio.h`
`fprintf(f,...)`	`int`	Escribe datos en el archivo `f` de acuerdo a un determinado formato especificado en los restantes argumentos.	`stdio.h`
`fputc(c,f)`	`int`	Escribe un carácter en el archivo `f`.	`stdio.h`
`fputs(s,f)`	`int`	Escribe una cadena de caracteres `s` en el archivo `f`.	`stdio.h`
`fread(s,i1,i2,f)`	`int`	Lee `i2` elementos, cada uno de tamaño `i1` bytes, desde el archivo `f` hasta la cadena `s`.	`stdio.h`
`free(p)`	`void`	Libera un bloque de memoria reservada cuyo principio está indicado por `p`.	`malloc.h` `stdlib.h`
`fscanf(f,...)`	`int`	Lee datos del archivo `f` de acuerdo a un determinado formato especificado en los restantes argumentos.	`stdio.h`
`fseek(f,l,i)`	`int`	Mueve el puntero al archivo `f` una distancia de `l` bytes desde la posición `i`.	`stdio.h`
`ftell(f)`	`long int`	Devuelve la posición actual del puntero dentro del archivo `f`.	`stdio.h`
`fwrite(s,i1,i2,f)`	`int`	Escribe `i2` elementos, cada uno de tamaño `i1` bytes, desde la cadena `s` hasta el archivo `f`.	`stdio.h`
`getc(f)`	`int`	Lee un carácter del archivo `f`.	`stdio.h`
`getchar()`	`int`	Lee un carácter desde el dispositivo de entrada estándar.	`stdio.h`
`gets(s)`	`char*`	Lee una cadena de caracteres desde el dispositivo de entrada estándar.	`atdio.h`
`isalnum(c)`	`int`	Determina si el argumento es alfanumérico. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isalpha(c)`	`int`	Determina si el argumento es alfabético. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isascii(c)`	`int`	Determina si el argumento es un carácter ASCII. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`iscntrl(c)`	`int`	Determina si el argumento es un carácter ASCII de control. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isdigit(c)`	`int`	Determina si el argumento es un dígito decimal. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`

Función	Tipo de salida	Propósito	Archivo de cabecera
`islower(c)`	`int`	Determina si el argumento es una minúscula. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isodigit (c)`	`int`	Determina si el argumento es un dígito octal. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isprint(c)`	`int`	Determina si el argumento es un carácter ASCII imprimible (hex `0x20-0x70`; octal `040-176`). Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`ispunct(c)`	`int`	Determina si el argumento es un carácter de puntuación. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isspace(c)`	`int`	Determina si el argumento es un espacio en blanco. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isupper(c)`	`int`	Determina si el argumento es una mayúscula. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`isxdigit (c)`	`int`	Determina si el argumento es un dígito hexadecimal. Devuelve un valor distinto de cero si es cierto; en otro caso devuelve 0.	`ctype.h`
`labs(l)`	`long int`	Devuelve el valor absoluto de `l`.	`math.h`
`log(d)`	`double`	Devuelve el logaritmo natural de `d`.	`math.h`
`log10(d)`	`double`	Devuelve el logaritmo en base 10 de `d`.	`math.h`
`malloc(u)`	`void*`	Reserva `u` bytes de memoria. Devuelve un puntero al principio del espacio reservado.	`stdlib.h`
`pow(d1,d2)`	`double`	Devuelve `d1` elevado a la potencia `d2`.	`math.h`
`printf(...)`	`int`	Escribe datos en el dispositivo de salida estándar de acuerdo a un determinado formato especificado en los restantes argumentos.	`stdio.h`
`putc(c,f)`	`int`	Escribe un carácter en el archivo `f`.	`stdio.h`
`putchar(c)`	`int`	Escribe un carácter en el dispositivo de salida estándar.	`stdio.h`
`puts(s)`	`char*`	Escribe una cadena de caracteres en el dispositivo de salida estándar.	`atdio.h`
`rand()`	`int`	Devuelve un entero positivo aleatorio.	`stdlib.h`
`rewind(f)`	`void`	Mueve el puntero al principio del archivo `f`.	`stdio.h`
`scanf(...)`	`int`	Lee datos en el dispositivo de entrada estándar de acuerdo a un determinado formato especificado en los restantes argumentos.	`stdio.h`

Función	Tipo de salida	Propósito	Archivo de cabecera
sin(d)	double	Devuelve el seno de d.	math.h
sinh(d)	double	Devuelve el seno hiperbólico de d.	math.h
sqrt(d)	double	Devuelve la raíz cuadrada de d.	math.h
srand()	void	Inicializa el generador de números aleatorios.	stdlib.h
strcmp(s1,s2)	int	Compara dos cadenas de caracteres lexicográficamente. Devuelve un valor negativo si s1 < s2; 0 si s1 y s2 son idénticas; y un valor positivo si s1 > s2.	string.h
strcmpi(s1,s2)	int	Compara dos cadenas de caracteres lexicográficamente, sin diferenciar entre mayúsculas y minúsculas. Devuelve un valor negativo si s1 < s2; 0 si s1 y s2 son idénticas; y un valor positivo si s1 > s2.	string.h
strcpy(s1,s2)	char*	Copia la cadena de caracteres s2 en la cadena s1.	string.h
strlen(s)	int	Devuelve el número de caracteres de una cadena.	string.h
strset(c,s)	char*	Copia todos los caracteres de s a c (excluyendo el carácter nulo al final \0).	string.h
strset(c,s)	char*	Copia todos los caracteres de s a c (excluyendo el carácter nulo al final \0).	string.h
system(s)	int	Pasa la orden al intérprete de comandos. Devuelve 0 si la orden se ejecuta correctamente; en otro caso devuelve un valor distinto de 0, típicamente -1.	stdlib.h
tan(d)	double	Devuelve la tangente de d.	math.h
tanh(d)	double	Devuelve la tangente hiperbólica de d.	math.h
toascii(a)	int	Convierte el valor del argumento a ASCII.	ctype.h
tolower(c)	int	Convierte una letra a minúscula.	ctype.h stdlib.h
toupper(c)	int	Convierte una letra a mayúscula.	ctype.h stdlib.h

APÉNDICE

C

Recopilación de llamadas al sistema

En este apéndice se recopilan las llamadas al sistema que han ido apareciendo en el texto, más otras adicionales relacionadas de forma significativa con las anteriores o con determinadas órdenes del intérprete de comandos. Conviene recordar que las llamadas al sistema suelen devolver algún resultado. El valor `0` o un valor positivo indican que la llamada se ha ejecutado satisfactoriamente, mientras que el valor `-1` indica que se ha producido algún error. En caso de error, la variable global `errno` contendrá un número que codifica el tipo de error producido. Este número tendrá un significado u otro dependiendo de cada llamada concreta.

- **alarm**

```
unsigned long alarm (unsigned long sec);
```

Activa un temporizador regresivo en tiempo real de `sec` segundos. Cuando el temporizador llega a 0, el proceso que realizó la llamada recibirá la señal SIGALARM.

- **brk, sbrk**

```
int brk (char *endds);
char *sbrk (int incr);
```

Llamadas para cambiar el tamaño del segmento de datos de un proceso. `brk` cambia la posición del fin del segmento de datos, haciendo que tome el valor `endds`. `Sbrk` incrementa el tamaño del mismo segmento en la cantidad de bytes especificados por `incr`.

- **chdir**

```
int chdir (char *path);
```

Cambia el directorio de trabajo actual asociado a un proceso. El nuevo directorio será el asociado a la ruta apuntada por `path`.

- **chmod, fchmod**

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod (char *path, mode_t mode);
int fchmod (int fildes, modet_t mode);
```

Llamadas para cambiar la máscara de modo de un fichero de acuerdo con el valor de `mode`. `chmod` trabaja con la ruta de fichero `path`; y `fchmod` lo hace con el descriptor de un fichero ya

abierto, `fildes`. En ambos casos, el argumento `mode` se puede formar a partir de las siguientes constantes:

`S_ISU1D`	`04000`	Cambiar el identificador de usuario al ejecutar.
`S_1SGID`	`02000`	Cambiar el identificador de grupo al ejecutar.
`S_ISVTX`	`01000`	Mantener el segmento de texto en memoria después de ejecutar.
`S_IRUSR`	`00400`	Permiso de lectura para el propietario.
`S_IWUSR`	`00200`	Permiso de escritura para el propietario.
`S_IXUSR`	`00100`	Permiso de ejecución (búsqueda) para el propietario.
`S_IRGRP`	`00040`	Permiso de lectura para el grupo.
`S_IWGRP`	`00020`	Permiso de escritura para el grupo.
`S_IXGRP`	`00010`	Permiso de ejecución (búsqueda) para el grupo.
`S_IROTH`	`00004`	Permiso de lectura para otros.
`S_IWOTH`	`00002`	Permiso de escritura para otros.
`S_IXOTH`	`00001`	Permiso de ejecución (búsqueda) para otros.

- **chown, fchown**

```
#include <sys/types.h>
chown (char *path, uid_t owner, gid_t group);
fchown (int fildes, uid_t owner, gid_t group);
```

Cambian el propietario y el grupo al que pertenece un fichero de acuerdo con los valores de `owner` (*uid* del nuevo propietario) y grupo (*gid* del nuevo grupo). `chown` trabaja con el nombre de un fichero y `fchown` lo hace con el descriptor de un fichero previamente abierto.

- **chroot**

```
int chroot(char *path);
```

Cambia el directorio raíz asociado a un proceso. El nuevo directorio raíz pasa a ser el referenciado por el puntero `path`.

- **close**

```
int close (int fildes);
```

Libera el descriptor de fichero `fd` y cierra su fichero asociado en el caso de que no haya más procesos que lo tengan abierto.

- **creat**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int creat (char *path, mode_t mode);
```

Llamada para crear un nuevo fichero cuya ruta está referenciada por `path`. El fichero se creará con la máscara de modo que se especifica en `mode`. Si el fichero ya existe, `creat` trunca su longitud a 0 bytes. Si la llamada se ejecuta con éxito devolverá un descriptor de fichero.

- **dup**

```
int dup (int fildes);
```

Llamada para duplicar una entrada en la tabla de descriptores de ficheros de un proceso, devuelve un nuevo descriptor que va a tener en común con `fildes` los siguientes campos: apuntará al mismo objeto de fichero abierto, tendrá Igual modo de acceso (lectura, escritura, lectura/escritura) e igual indicador de estado. El descriptor devuelto es el menor de entre los que haya disponibles.

- **execl, exev, execle, execve, execlp, execvp**

```
int execl (path, arg0, argl, ... , argn, (char *)0)
char *path, *arg0, *argl, ..., *argn;

int execv (path, argv)
char *path, *argv[];

int execle (path, arg0, argl, ... , argn, (char *)0, envp)
char *path, *arg0, *argl, ...• *argn, *envp[];

int execve (path, argv, envp)
char *path, *argv[], *envp[];

int execlp (file, arg0, argl, ... , argn, (char *)0)
char *file, *arg0, *argl, ..., *argn;

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

La llamada al sistema `exec` sirve para invocar desde un proceso a otro programa ejecutable (programa compilado o shell script). Básicamente `exec` carga las regiones de código, datos y pila del nuevo programa en el contexto de usuario del proceso que la invoca. Existe toda una familia de funciones de librería asociadas a esta llamada al sistema: `execl`, `execv`, `execle`, `execve`, `execlp` y `execvp`.

En todas estas funciones, `path` es la ruta del fichero ejecutable que es invocado. `File` es el nombre de un fichero ejecutable, la ruta del fichero se construye buscando el fichero en los directorios indicados en la variable de entorno `PATH`.

`Arg0`, `arg1`,...,`argN` son punteros a cadenas de caracteres y constituyen la lista de argumentos o parámetros que se le pasa al nuevo programa. Por convenio, al menos `arg0` está presente siempre y apunta a una cadena idéntica a `path` o al ultimo componente de `path`. Para indicar el final de los argumentos siempre a continuación del último argumento `argN` se pasa un puntero nulo `NULL`.

`Envp` es un array de punteros a cadenas de caracteres terminado en un puntero nulo que constituyen el *entorno* en el que se va ejecutar el nuevo programa.

- **exit**

```
#include <stdlib.h>
void exit (int status);
```

Termina la ejecución del proceso que la invoca y devuelve `status` a su proceso padre para que lo pueda examinar para identificar la causa por la que finalizó el proceso hijo de acuerdo a unos criterios que haya previamente establecido el usuario.

- **fork**

```
#include <sys/types.h>
pid_t fork();
```

Crea un nuevo proceso. Si se ejecuta correctamente devuelve al proceso padre el *pid* que le asigne el sistema al proceso hijo. Asimismo devuelve `0` al proceso hijo.

- **getitimer, setitimer**

```
#include <time.h>
getitimer (int wich, struct itimerval *value);
setitimer (int wich, struct itimerval *value,
           struct itimerval *ovalue);
```

Estas llamadas se utilizan para controlar los tres temporizadores asociados a un proceso. `getitimer` se utiliza para leer el estado del temporizador especificado por `wich`. Los valores que puede tomar este argumento son:

`ITIMER_REAL`	Temporizador en tiempo real.
`ITIMER_VIRTUAL`	Temporizador que contabiliza el tiempo que el proceso se ejecuta en modo usuario (tiempo virtual).
`ITIMER_PROF`	Temporizador que contabiliza el tiempo que el proceso se ejecuta en modo usuario y en modo núcleo.

El valor del temporizador es devuelto a través de los campos de la estructura apuntada por `value`. Esta estructura se define como sigue:

```
struct itimerval {
      struct timeval it_interval; /*Intervalo del temporizador. */
      struct timeval it_value; /* Valor actual del temporizador. */
};
```

La estructura `timeval` se define así:

```
struct timeval {
      unsigned long tv_sec; /* Segundos transcurridos desde el día
                               1 de Enero de 1970 */
```

```
        long  tv_usec;    /* Microsegundos. Su rango está comprendido
                             entre O y 999.999 */
    };
```

`setitimer` se utiliza para definir el valor del temporizador especificado por `wich`. `Value` es un puntero a una estructura con los nuevos valores del temporizador y `ovalue` es un puntero a una estructura donde se devuelven los antiguos valores del temporizador.

- **getpid, getppid**

```
    <sys/types.h>
    pid_t getpid ();
    pid_t getppid ();
```

`getpid` devuelve el *pid* del proceso que la invoca. Por su parte, `getppid` devuelve el *pid* del proceso padre del proceso que realiza la llamada.

- **gettimeofday, settimeofday**

```
    <time.h>
    int gettimeofday (struct timeva *tp, struct timezone *tzp);
    int settimeofday (struct timeval *tp, struct timezone *tzp);
```

Estas dos llamadas se utilizan para manipular el reloj interno del sistema. Con `gettimeofday` se puede leer la fecha del sistema expresada en segundos y microsegundos con respecto al día 1 de Enero de 1970 GMT. `settimeofday` se utiliza para fijar una nueva fecha del sistema.

`tp` y `tzp` son punteros a estructuras del tipo `timeval` y `timezone`, respectivamente. Estas estructuras se definen como sigue:

```
    struct timeval {
        unsigned long tv_sec; /* Segundos desde el día 1 de Enero
                                 de 1970 */
        long  tv_usec;        /* Microsegundos. */
    };
    struct timezone {
        int tz_minuteswest; /* Corrección con respecto al Meridiano
                               de Greenwich */
        int tz_dsttime;     /* Corrección anual. */
    };
```

- **getuid, geteuid, getgid, getegid**

```
    <sys/types.h>
    uid_tgetuid ();
    uid_t geteuid ();
    gid_t getgid ();
```

```
gid_t getegid ();
```

Las llamadas al sistema `getuid`, `geteuid`, `getgid` y `getegid` permiten determinar qué valores toman los identificadores *uid*, *euid*, *gid* y *egid*, respectivamente

- **kill**

```
#inc1ude <signa1.h>
int kill (pid_t pid, int sig);
```

Permite a un proceso enviar una señal a otro proceso o a un grupo de procesos. `pid` es un número entero que permite identificar al proceso o conjunto de procesos a los que el núcleo va a enviar una señal. Si `pid` >0, el núcleo envía la señal al proceso cuyo *pid* sea igual a `pid`. Si `pid` = 0, el núcleo envía la señal a todos los procesos que pertenezcan al mismo grupo que el proceso emisor. Si `pid` = -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 `pid` < -1, el núcleo envía la señal a todos los procesos cuyo *gid* sea igual al valor absoluto de `pid`. `sig` 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.

- **link**

```
int link (char *path1, char *path2);
```

Crea una entrada de directorio cuya ruta será igual a la cadena apuntada por `path2`. Esta nueva entrada va a ser un enlace con el fichero cuyo nodo-i es el correspondiente a la ruta apuntada por `path1`.

- **lseek**

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek (int fildes, off_t offset, int whence);
```

Permite realizar accesos aleatorios mediante la configuración del puntero de lectura/escritura a un valor especifico. `fildes` es el descriptor del fichero, `offset` es el número de bytes que se va desplazar el puntero y `whence` es la posición desde donde se va desplazar el puntero, que puede tomar los siguientes valores constantes:

`SEEK_SET`	El puntero avanza offset bytes con respecto al inicio del fichero. El valor de esta constante es `0`.
`SEEK_CUR`	El puntero avanza offset bytes con respecto a su posición actual. El valor de esta constante es `1`.
`SEEK_END`	El puntero avanza offset bytes con respecto al final del fichero. El valor de esta constante es `2`.

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.

- **mkdir**

```
int mkdir (char *path, mode_t mode);
```

Crea un fichero de directorio con una ruta igual a la cadena apuntada por `path`. `mode` codifica los la máscara de modo del directorio.

- **mknod**

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod (char *path, mode_t mode, dev_t dev);
```

Crea un fichero de dispositivo, una tubería con nombre o fichero FIFO o un directorio. `path` apunta a una cadena de caracteres que contiene la ruta del fichero a crear. `mode` es la máscara de modo del fichero, en la que sólo uno de los siguientes bits deberá estar activo:

`S_IFIFO`	Crear una tubería con nombre o fichero FIFO.
`S_IFCHR`	Crear un fichero de dispositivo modo carácter.
`S_IFBLK`	Crear un fichero de dispositivo modo bloque.
`S_IFDIR`	Crear un directorio.

Los 9 bits menos significativos de `mode` codifican los permisos del fichero. Si el fichero a crear es un dispositivo, `dev` debe codificar los números principal y secundario del mismo.

- **mount**

```
int mount (char *spec, char *dir, int rwflag);
```

Monta un sistema de ficheros en un determinado directorio. `spec` 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 `rwflags` 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.

- **msgctl**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl (int msqid, int cmd, struct msqid_ds *buf);
```

Permite leer y modificar la información estadística y de control de una cola de mensajes. `msqid` es el identificador de la cola, `cmd` es un número entero o una constante simbólica que especifica la operación a efectuar y `buf` 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. (Más información relativa a esta llamada al sistema en la sección 7.3.3).

- **msgget**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget (key_t key, int msgflg);
```

Crea una cola de mensajes o bien permite acceder a una cola ya existente usando una clave dada. `key` es la clave de la cola de mensaje y `msgflg` es una máscara de indicadores (ver sección 7.3.1.4). 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.

- **msgrcv**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int msqid, void *msgp, int msgsz, long msgtyp,
            int msgflg);
```

Extrae un mensaje de una determinada cola de mensajes. `msqid` es un identificador de una cola de mensajes, `msgp` es un puntero a la variable del espacio de direcciones del usuario donde se va almacenar el mensaje, `msgsz` es la longitud del texto del mensaje en bytes, `msgtyp` indica el tipo del mensaje que se desea extraer. Si `msgtyp`=0 se extrae el primer mensaje que haya en la cola independientemente de su tipo. Corresponde al mensaje más viejo. Si `msgtyp` > 0 se extrae el primer mensaje del tipo `msgtyp` que haya en la cola. Por último si `msgtyp` < 0 se extrae el primer mensaje que cumpla que su tipo es menor o igual al valor absoluto de `msgtyp` y a la vez sea el más pequeño de los que hay. `msgflg` 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.

- **msgsnd**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int msqid, void *msgp, int msgsz, int msgflg);
```

Envía un mensaje a una determinada cola de mensajes. `msqid` es un identificador de una cola de mensajes, `msgp` puntero a 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 `msgflg` 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.

- **nice**

```
int nice (int incr);
```

Permite aumentar o disminuir el factor de amabilidad actual del proceso que la invoca. `incr` 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.

- **open**

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open (char *path, int oflag [, mode_t mode]);
```

Permite abrir un fichero ya existente. `path` es la ruta del fichero y `flags` puede ser o una máscara de modo octal o una máscara de bits que permiten especificar los permisos de apertura de dicho fichero. Cuando el argumento `flags` se especifica mediante una máscara de bits, ésta típicamente se implementa como una combinación de constantes enlazadas con el operador OR a nivel de bit ('|'). Algunas de las constantes utilizadas más frecuentemente son:

`O_RDONLY`	Abrir en modo sólo lectura.
`O_WRONLY`	Abrir en modo sólo escritura.
`O_RDWR`	Abrir para leer y escribir.
`O_CREAT`	Crear el fichero si no existe con la máscara de modo especificada en `mode`.
`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.

De las constantes `O_RDONLY`, `O_WRONLY` y `O_RDWR` solo una de ellas debe estar presente al componer la máscara `flags`, de lo contrario, el modo de apertura quedaría indefinido. Si `open` se ejecuta con éxito devuelve de un descriptor de fichero.

- **pause**

```
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.

- **pipe**

```
int pipe (int fildes [2]);
```

Crea una tubería sin nombre y devuelve dos descriptores a través de los cuales se puede leer y escribir en la tubería. Para leer de la tubería hay que usar el descriptor `fildes[0]`, mientras que para escribir en la tubería hay que usar el descriptor `fildes[1]`.

- **ptrace**

```
#include <sys/ptrace.h>
int ptrace (int request, int pid, int addr, int data);
```

Permite a un proceso padre (proceso depurador) controlar la ejecución de un proceso hijo (proceso depurado). `pid` 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 `request`. El argumento `request` permite al padre realizar las siguientes operaciones:

`0`	Habilita la depuración en el proceso hijo (éste parámetro lo utiliza sólo el proceso hijo).
`1, 2`	Devuelve el contenido de la dirección de memoria virtual referenciada por `addr` en el proceso hijo.
`3`	Devuelve el contenido de la posición `adrr` del área de usuario del proceso hijo.
`4, 5`	Escribe, con el valor `data`, el contenido de la dirección de memoria virtual referenciada por `addr` en el proceso hijo.
`6`	Escribe, con el valor `data`, en la posición `addr` del área de usuario del proceso hijo.
`7`	Le indica al proceso hijo que continúe con su ejecución. El proceso hijo estará parado debido a la recepción de una señal.
`8`	Fuerza a que el proceso hijo termine su ejecución con una llamada al sistema `exit`.
`9`	Le indica al proceso hijo que continúe su ejecución, pero activando el bit de traza del procesador. Esto hace que el proceso interrumpa su ejecución después de cada instrucción máquina. Esta orden se emplea para ejecutar un proceso paso a paso.

- **raise**

```
#include <signal.h>
int raise(int sig);
```

Permite a un proceso enviarse una señal a sí mismo. `sig` es una constante entera que identifica a la señal que se desea enviar. También se puede introducir directamente el número asociado a dicha señal.

- **read**

```
int read (int fildes, char *buf, unsigned nbyte);
```

Permite leer datos de un fichero. `fildes` es el descriptor de fichero, `buf` es el array de caracteres donde se almacenarán los datos que se lean en el fichero y `nbyte` es el número de bytes que se desea leer. Si la llamada al sistema se ejecuta devuelve el número de bytes leídos.

- **rename**

```
#include <stio.h>
rename (const char *source, const char *target);
```

Hace que el fichero cuya ruta indica `source` pase a llamarse como indica `target`.

- **rmdir**

```
rmdir (char *path);
```

Borra el directorio cuyo nombre viene indicado por `path`. Para que pueda ser borrado no debe contener ningún fichero.

- **semctl**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl (int semid, int semnum, int cmd,
      union semun
      {
            int val;
            struct semid ds *buf;
            ushort *array;
      }arg);
```

Permite acceder a la información administrativa y de control que posee el núcleo sobre un cierto conjunto de semáforos. `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 que va a realizar la llamada al sistema `semctl` y `arg` se utiliza para almacenar los argumentos o los resultados de la operación.

- **semget**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget (key_t key key, int nsems nsems, int semflg);
```

La llamada al sistema `semget` crea u obtiene un array o conjunto de semáforos. `key` es una llave numérica del tipo predefinido `key_t` o bien la constante IPC_PRIVATE que obliga a crear un nuevo identificador, `nsems` es el número entero de semáforos del conjunto o array asociados a `key` y `semflg` 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. Si la llamada al sistema `semget` se ejecuta con éxito devolverá 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.

- **semop**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop (int semid, struct sembuf *sops, int nsops);
```

Realiza operaciones sobre los elementos de un determinado conjunto de semáforos. `semid` es un identificador de un array o conjunto concreto de semáforos, `sops` es un puntero a un array de estructuras del tipo `sembuf` (ver sección 7.3.2.2) 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.

- **setuid, setgeid**

```
#include <sys/types.h>
int setuid (uid_t uid);
int setgid (gid_t gid);
```

La llamada al sistema `setuid` permite asignar el valor `uid` al *euid* y al *uid* del proceso que invoca a la llamada. Si el identificador de usuario efectivo del proceso que efectúa la llamada es el del superusuario, entonces en este caso *uid*=`uid` y *euid*=`uid`. Si el identificador del usuario efectivo del proceso que efectúa la llamada no es el del superusuario, entonces en este caso *euid*=`uid` si el valor del parámetro `uid` coincide con el valor del *uid* del proceso o si esta llamada se está invocando dentro de la ejecución de un programa que tiene su bit `S_ISUID` activado y el valor del parámetro `uid` coincide con el valor del *uid* del propietario del programa.

La explicación del funcionamiento de la llamada al sistema `setguid` es análoga a la explicada para `setuid` pero referido a los identificadores *gid* y *egid*.

- **shmctl**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl (int shmid, int cmd, struct shmin_ds *buf);
```

Permite realizar operaciones de control sobre una zona de memoria compartida creada previamente por `shmget`. `shmid` es el identificador de una región de memoria compartida, `cmd` es un número entero o una constante simbólica que especifica la operación a efectuar y `buf` es un puntero a una estructura del tipo predefinido `shmid_ds` que contiene los argumentos de la operación. (Más información en la sección 7.3.4.5).

- **shmget**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget (key_t key, int size, int shmflg);
```

Crea un segmento de memoria compartida o accede a uno que ya existe. `key` es la clave de acceso a un segmento de memoria compartida, `size` especifica el tamaño en bytes del segmento de memoria solicitado y `shmflg` es una máscara de indicadores (ver sección 7.3.1.4). Si la llamada al sistema se ejecuta con éxito devolverá el identificador entero de la zona de memoria compartida asociada a la llave `key`.

- **shmat**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat (int shmid, char *shmaddr, int shmflg);
```

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. `shmid` es un identificador de una región de memoria compartida, `shmaddr` es la dirección virtual del proceso donde se desea que empiece la región de memoria compartida. Si `shmaddr` = 0, el sistema selecciona la dirección. Es la opción más adecuada si se desea conseguir portabilidad. Si `shmaddr` ≠ 0, el valor de la dirección devuelto depende si se especificó o no el bit SHM_RND del parámetro `shmflg`. Si se especificó el segmento de memoria es enlazada en la dirección especificada por el parámetro `shmaddr` 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 `shmaddr`. `shmflg`, 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 devuelve la dirección a la que está unido el segmento de memoria compartida `shmid`.

- **shmdt**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt (char *shmaddr);
```

Desenlaza un segmento de memoria compartida del espacio de direcciones de un proceso. `shmaddr` es la dirección virtual del segmento de memoria compartida que se quiere separar del proceso.

- **sigblock**

```
#include <signal.h>
long sigblock (long mask);
```

Permite añadir nuevas señales bloqueadas a la máscara actual de señales. `mask` que es un entero largo que se utilizará como operando junto con la máscara actual de señales para realizar una operación lógica de tipo OR a nivel de bits. Se considera que la señal número *j* está bloqueada si el j-ésimo bit de `mask` está a 1. Este bit puede ser fijado con la macro `sigmask(j)`. Si la llamada se ejecuta con éxito devuelve la máscara de señales que se tenía especificada antes de ejecutar esta llamada al sistema

- **signal**

```
#include <signal.h>
void (*signal (int sig, void (*action)()))();
```

Permite especificar el tratamiento de una determinada señal recibida por un proceso. `sig` 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. `action` 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`	Es la dirección del manejador de la señal definido por el usuario.

Si `signal` se ejecuta con éxito devuelve 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).

- **sigpause**

```
#include <signal.h>
long sigpause (long mask);
```

Bloquea la recepción de señales de acuerdo con el valor de `mask`, de la misma forma que hace `sigsetmask`. A continuación se pone a esperar a que llegue alguna de las señales no bloqueadas. Si no se desea que `sigpause` bloquee ninguna señal, se le debe pasar la máscara `OL`.

Cuando `sigpause` termina su ejecución, restaura la máscara de señales que había antes de llamarla. La ejecución de `sigpause` termina cuando es interrumpida por una señal. Después de tratar la señal, `sigpause` hace que `errno` tome el valor EINTR y devuelve el valor -1.

- **sigsetmask**

```
#include <signal.h>
long sigsetmask (long mask);
```

Fija la máscara actual de señales, es decir, permite especificar qué señales van a estar bloqueadas. Obviamente, aquellas señales que no pueden ser ignoradas ni capturadas, tampoco van a poder ser bloqueadas. `mask` 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 `mask` está a 1. Este bit puede ser fijado con la macro `sigmask(j)`. Si la llamada se ejecuta con éxito devuelve la máscara de señales que se tenía especificada antes de ejecutar esta llamada al sistema.

- **sigvector**

```
#include <signal.h>
sigvector (int sig, struct sigvec *vec, struct sigvec *ovec);
```

`sigvector` se utiliza para especificar la forma de tratar una señal. `sig` 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.

Tanto `vec` como `ovec` son punteros a estructuras del tipo `sigvec`. Esta estructura está definida con los siguientes campos:

```
struct sigvec
{
       void (*sv_handler) ();
       long sv_mask;
       long sv_flags;
};
```

El campo `sv_handler` es un puntero a una función que devuelve `void` y tiene el mismo significado que el parámetro `action` de la llamada `signal`. Este campo se utiliza para indicar cuál será la rutina de tratamiento de la señal. Al igual que en el caso de `signal`, puede tomar tres tipos de valores con diferente significado:

`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`	Es la dirección del manejador de la señal definido por el usuario.

El campo `sv_mask` codifica, en cada uno de sus bits, las señales que no se desen que sean tratadas si son recibidas mientras se está ejecutando la rutina de tratamiento actual. Normalmente, este campo vale 0, lo que indica que mientras se está tratando una señal, cualquier otra señal puede interrumpir. Si alguno de los bits de `svmask` vale 1, se impide el anidamiento cuando se recibe esa señal.

El campo `sv_flags` codifica cuál va a ser la semántica (significado) que se emplee en la recepción de la señal. Los siguientes bits están definidos para este campo:

`SV_BSDSIG`	Usar la semántica de BSD. Esto significa, entre otras cosas, que cuando se instala una rutina de tratamiento, permanecerá instalada hasta que se haga otra llamada a `sigvector` que instale una rutina nueva.
`SV_RESETHAND`	Impone las misma semántica que la llamada `signal` del UNIX System V. Es decir, siempre se restaura la rutina de tratamiento por defecto antes de entrar en la rutina de tratamiento suministrada por el usuario.

`vec` apunta al que va a ser el nuevo vector de señal y `ovec` devuelve **un** puntero al vector que había, por si desea restaurarlo posteriormente.

- **stat, fstat**

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (char *path, struct stat *buf);
int fstat (int fildes, struct stat *buf);
```

`stat` devuelve, a través de `buf`, información sobre el estado del fichero cuya ruta es `path`. `fstat` hace lo mismo con el fichero descrito por `fildes`. La estructura de `buf` es la siguiente:·

```
struct stat {
      dev_t st_dev     /* Número de dispositivo del disco donde
                          se encuentra el fichero. * /
      ino_t st_ino     /* Nodo-i del fichero. */
      ushort st_mode   /* Máscara de modo*/
      ushort st_nlink* /* Número de enlaces al fichero. */
      uid_t st_uid     /* Identificador de usuario (uid) del
                          propietario del fichero. */
```

```
gid_t st_gid     /* Identificador del grupo (gid) del
                    propietario del fichero. */
dev_t st_rdev    /* Número principal y número secundario.
                    Tiene significado únicamente para los
                    ficheros especiales. */
off_t st_size    /* Tamaño, en bytes, de un fichero.*/
time_t st_atime  /* Fecha del último acceso al fichero
                    (lectura).*/
time_t st_mtime  /*Fecha de la última modificación del
                   fichero.*/
time_t st_ctime  /*Fecha del último cambio de la información
                   administrativa del fichero*/
}
```

- **stime**

```
int stime (long *tp);
```

Permite fijar la fecha y la hora actuales del sistema con el valor apuntado por `tp` contiene los segundos transcurridos desde las 00:00:00 GMT del día 1 de enero de 1970.

- **sync**

```
void sync();
```

Copia en el disco aquellos bloques de la caché de buffers de bloques de disco cuyo contenido ha sido modificado. Se asegura de este modo la consistencia de los datos del sistema.

- **time**

```
#include <time.h>
time_t time (time_t *tloc);
```

Permite leer la fecha y la hora actuales que almacena el sistema. Si la llamada se ejecuta con éxito en `tloc` se almacenarán los segundos transcurridos desde las 00:00:00 GMT del día 1 de enero de 1970. Además esta llamada también devuelve esta misma información.

- **times**

```
#include <sys/times.h>
clock_t times (struct tms *buffer);
```

Permite conocer el tiempo empleado por un proceso en su ejecución. Si `times` se ejecuta con éxito almacena en `tbuffer` la información estadística relativa a los tiempos de ejecución empleados por el proceso, desde su inicio hasta el momento de invocar a `times`. (Más información en sección 6.2.4.3).

- **umount**

```
int umount(char *name);
```

Desmonta el sistema de ficheros que se encuentra montado sobre el fichero de dispositivo indicado en `name`.

- **unlink**

```
int unlink(char *path);
```

Borra la entrada del directorio especificada en la ruta apuntada por `path`.

- **wait**

```
pid_t wait (int *stat_loc);
```

Suspende la ejecución del proceso actual hasta que alguno de sus procesos hijos finalice. `stat_loc` 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. Si la llamada se ejecuta con éxito devuelve el *pid* del proceso hijo que ha terminado.

- **write**

```
int write (int fildes, ehar *buf, unsigned nbyte);
```

Esta llamada permite escribir datos en un fichero. `fildes` es el descriptor de fichero, `buf` es el array de caracteres donde se encuentran almacenados los datos que se van a escribir en el fichero y `nbyte` es el número de bytes que se desea escribir. Si `write` se ejecuta con éxito, devuelve el número de bytes escritos.

BIBLIOGRAFÍA

- 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.

ÍNDICE

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z