T E M A 6

Implementación de UNIX

Contenido

TEMA 6

IMPLEMENTACIÓN DE UNIX

6.1. Introducción

Aunque la implementación de UNIX varía de versión en versión y de máquina a máquina, existen conceptos e ideas que son aplicables a todas. El estudio de la implementación de UNIX es útil por dos motivos. En primer lugar, muchas ideas empleadas en UNIX son usadas por sistemas operativos más recientes, como OS/2 o Windows NT. En segundo lugar, el conocimiento de la implementación permite sacar el máximo partido al sistema operativo a la hora de programar aplicaciones.

6.2. El Núcleo (*Kernel*) de UNIX

El *kernel* (núcleo) de UNIX es la parte del sistema que se carga desde disco a memoria cuando el computador es encendido y permanece siempre en memoria hasta que el sistema es apagado o ocurre algún error grave. Su mayor parte está escrita en C, aunque algunos componentes del mismo están escritos en ensamblador por motivos de eficiencia.

UNIX se basa en dos conceptos básicos: ficheros y procesos. Por este motivo, el *kernel* está diseñado para facilitar servicios relacionados con estos dos elementos (Figura 1). Los procesos son entidades "activas", lo que significa que realizan acciones por sí mismos, mientras que los ficheros son meros contenedores de información que son usados por los procesos.

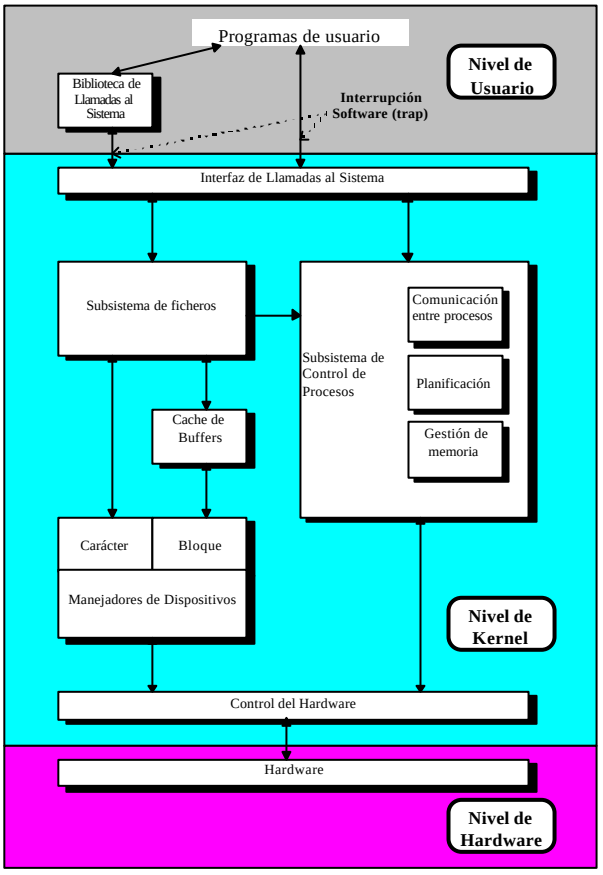

Figura 1.- Diagrama de bloques del *kernel*.

El *kernel* es accedido por los programas de usuario mediante un interface conocido como **llamadas al sistema**, que permiten a los programas de usuario gestionar procesos, ficheros y otros recursos. Toda llamada al sistema tiene asignado un identificador numérico. Los procesos realizan una llamada al sistema colocando los argumentos de la misma en registros o en la pila (*stack*) e invocando una instrucción especial (*trap*) que pasa de modo usuario a modo *kernel*. El identificador de la llamada al sistema se usa como índice para acceder a una tabla de llamadas al sistema que existe en el *kernel*. Esta tabla es un array de punteros a funciones que implementan cada llamada al sistema. Estas funciones acceden a estructuras de datos residentes en el *kernel*. Debido a que no existe una instrucción *trap* en C, se suministra una biblioteca de funciones escritas en ensamblador pero que pueden ser invocadas desde programas escritos en C.

El *kernel* tiene dos componentes principales: el subsistema de control de ficheros y el subsistema de control de procesos. El subsistema de control de ficheros realiza tareas relacionadas con la gestión de

ficheros, como la asignación de espacio en disco, administración del espacio libre, control del acceso a los ficheros e intercambio de información entre ficheros y programas de usuario. Los procesos interactúan con el subsistema de ficheros a través de un conjunto de llamadas al sistema específicas, como `open`, `close`, `read`, `write`, `stat`, `chown`, `chmod`, ... La comunicación entre el subsistema de ficheros y los dispositivos físicos se realiza mediante los manejadores (*drivers*) de dispositivo. Se distinguen dos tipos de dispositivos según la forma en que son accedidos:

- dispositivos de bloques (*block devices*).
- dispositivos de caracteres (*raw devices*).

El acceso a los dispositivos de bloques se realiza con la intervención de *buffers*, que permiten aumentar la velocidad de transferencia. Un mismo dispositivo físico puede ser manejado tanto en modo bloque como en modo carácter, dependiendo del manejador que use para acceder a él.

El subsistema de control de procesos lleva a cabo la comunicación y sincronización entre procesos, gestiona la memoria y la planifica los procesos para su ejecución. Algunas llamadas al sistema para control de proceso son `fork`, `exec`, `exit`, `wait`, `brk` y `signal`. El módulo de gestión de memoria controla la asignación de la memoria. Así, por ejemplo, cuando en un momento dado no existe memoria física suficiente para contener a todos los procesos, el gestor de memoria se encarga de mover algunos procesos a almacenamiento secundario para permitir que otros se puedan ejecutar. Para ello utiliza algunos mecanismos como *swapping* (intercambio) y demanda de páginas. El planificador (*scheduler*) asigna la CPU a los procesos. Entra en ejecución cada cierto tiempo y determina si el proceso en ejecución puede seguir su ejecución o, por el contrario, hay que quitarle la CPU (*preemption*) para asignársela a otro proceso. El módulo de comunicación entre procesos permite que procesos arbitrarios intercambien

El control de hardware es la parte del *kernel* que se encarga del manejo de las interrupciones y de la comunicación con la máquina. Dispositivos como los discos o terminales pueden interrumpir a la CPU mientras ésta está ejecutando un proceso. En estos casos, el *kernel* interrumpirá al proceso en ejecución, atenderá la interrupción y posteriormente reanudará la ejecución del proceso interrumpido.

El *kernel* contiene varias estructuras de datos (tabla de procesos, tabla de nodos índice, tabla de ficheros, ...) que se implementan como tablas de tamaño fijo en lugar de estructuras de datos dinámicas. La *kernel* es muy simple. El inconveniente es que limita el número de entradas de una estructura de datos al número con el que fue originariamente configurada cuando se generó el sistema. Si se agotan las entradas de una estructura de datos el *kernel* no puede asignar nuevo espacio y devolverá un error al proceso correspondiente. Por otro lado, el espacio reservado para las tablas que no es usado no se puede utilizar para otras tareas. No obstante, en UNIX se considera más importante la simplicidad de los algoritmos del *kernel* que tener que aprovechar al máximo toda la memoria del sistema. De este modo, los algoritmos son más eficientes y fáciles de comprender.

6.3. Gestión de Procesos

Cuando un programa es compilado se almacena en un fichero con un formato especial. Este formato consta de cuatro partes:

- Una cabecera primaria que incluye, entre otra información, el número de secciones del fichero y el *magic number*), que es usado por el *kernel* para identificar el tipo de fichero ejecutable (por ejemplo, si los dos primeros caracteres son "#!", se trata de un fichero de texto que contiene un *shell script*).
- Un conjunto de cabeceras (*headers*) de sección, que describen cada sección el fichero. Esta descripción incluye el tipo y tamaño de la sección y la dirección de memoria virtual que debería ocupar la sección cuando el programa sea ejecutado.
- Un conjunto de secciones, que contienen el código y los datos que se han de cargar en memoria en el espacio de direcciones del futuro proceso. Los datos constituyen la representación en lenguaje máquina de las variables que tienen un valor inicial cuando el programa inicia su ejecución, y la *kernel* debería de reservar para las variables sin inicializar. Esta cantidad se denomina `bss` y es inicializada cero por el *kernel* al inicio de la ejecución del programa.
- Otras secciones, como información sobre la tabla de símbolos e información útil para depuración.

Las cabeceras son usadas por el *kernel* para cargar el programa en memoria e iniciar su ejecución, lo que da lugar a un proceso. El *kernel* carga un fichero ejecutable en memoria durante la ejecución de una llamada al sistema `exec`. Un proceso en memoria consiste en tres regiones o segmentos: código, datos y pila. Los segmentos de código y datos se corresponden con las secciones de código y datos del fichero ejecutable. La pila se crea de forma automática y su tamaño varía en tiempo de ejecución y es controlado por el *kernel*. La pila consiste en marcos de pila (*stack frames*) que son añadidos cuando se llama a una función y extraídos cuando ésta termina. La profundidad de la pila la indica un registro denominado puntero de pila (*stack pointer*). Un marco de pila contiene los parámetros de la función, sus variables locales y los datos necesarios para recuperar el anterior marco de pila, que incluyen los valores del contador de programa y el *stack pointer* que existían cuando se llamó a la función. Un proceso en UNIX se puede ejecutar en dos modos, usuario y *kernel*. Para cada uno de estos modos existe una pila. La pila de usuario contiene los argumentos, variables locales y los datos de las funciones ejecutadas en modo usuario, mientras que la pila de *kernel* contiene los marcos de pila de las funciones que se ejecutan en modo *kernel*.

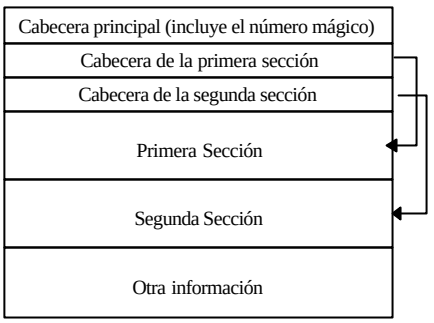

Figura 2.- Estructura de un fichero ejecutable.

En la Figura 3 se muestra un programa en C que copia un fichero. El programa se invoca de la siguiente forma: `copy oldfile newfile` , donde `oldfile` es el nombre de un fichero existente y `newfile` es el nombre del nuevo fichero. En la Figura 4 se muestran las pilas de usuario y de *kernel* de un proceso resultante de ejecutar este programa, tras la llamada al sistema `write`.

```
/*************************************************************************
*****/
/*  Programa copy.c: copia dos ficheros usando llamadas al sistema.
*/
/*************************************************************************
*****/
#include <fcntl.h>
char buffer[2048] ;

main(argc, argv)
int    argc ;   /* Contiene el número de argumentos
*/
char  *argv[] ; /* Contiene los argumentos en forma array de strings
*/
{
    int fdold, fdnew ; /* Descriptores de los ficheros */

    if (argc != 3) {
        printf("Error: son necesarios dos argumentos\n") ;
        exit(-1) ;
    }
    fdold = open(argv[1], O_RDONLY) ; /* Apertura del primer fichero
en */
                                      /* modo solo-lectura
*/
    if (fdold == -1) {
        prinf("Error: no se puede abrir el fichero %s\n", argv[1]) ;
        exit(-1) ;
    }

    /* Creación del fichero y comprobación de errores */
    if ((fdnew = creat(argv[2], 0666)) == -1) {
        printf("Error: no se puede crear el fichero %s\n", argv[2]) ;
        exit(-1) ;
    }
    copy(fdold, fdnew) ; /* Llamada a la función que copia ficheros */
    exit(0) ; /* Fin del programa */
```

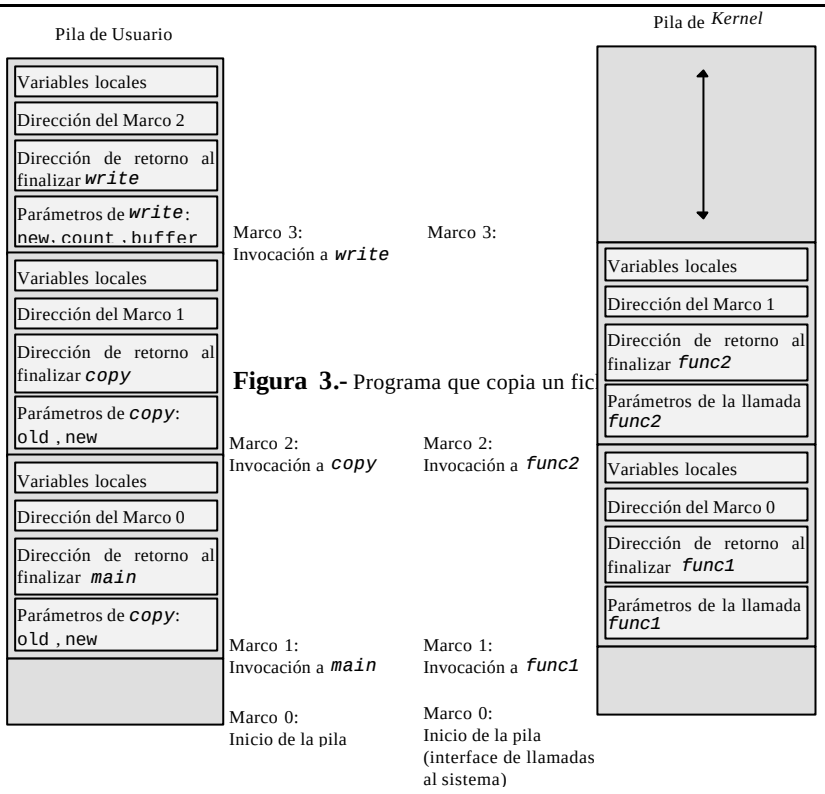

Figura 3.- Programa que copia un fic

Figura 4.- Pilas de usuario y *kernel* para el programa de la Figura 3.

```
}

copy(old, new)
int old ; /* Descriptor del fichero origen  */
int new ; /* Descriptor del fichero destino */
{
    int count ; /* Cuenta los bytes leidos del fichero origen */
    while ((count == read(old, buffer, sizeof(buffer)))>0)
        write(new, buffer, count) ;
}
```

Los procesos se pueden ejecutar, en apariencia, simultáneamente, encargándose el *kernel* de planificarlos para su ejecución. Varios procesos pueden ser instancias de un mismo programa. Cada proceso se ejecuta siguiendo una secuencia de instrucciones que se contienen en el segmento de texto y no puede saltar a la secuencia de instrucciones de otro proceso. Los procesos leen y escriben en sus propias zonas de datos y pila, pero no puede acceder a los datos y pila de otros procesos.

En términos prácticos, un proceso en UNIX es una entidad que ha sido creada mediante la llamada al sistema `fork`. Todo proceso excepto el proceso 0 se crea cuando otro proceso ejecuta la llamada al sistema `fork`. El proceso que ejecuta la llamada `fork` se denomina proceso padre, y el creado es un proceso hijo. Todo proceso posee un único proceso padre, pero un proceso puede tener varios hijos. El *kernel* identifica cada proceso mediante su número de proceso (*Process ID* o *PID*). El proceso 0 es un proceso especial que se crea cuando el *kernel* inicia su ejecución. Después de crear dos procesos hijo, el 1 y el 2, el proceso 0 se convierte en el proceso *swapper* (intercambiador). El proceso 1, denominado *init*, es el antepasado del resto de los futuros procesos del sistema. El proceso 2 es el demonio de páginas (*page daemon*).

6.3.1. Estructuras de Control de Procesos

El *kernel* mantiene varias estructuras de datos relacionadas con los procesos. Las más importantes son la **tabla de procesos** y la estructura o **área de usuario** (*u area*). La tabla de procesos es una tabla global del

kernel que contiene una entrada por cada proceso existente en el sistema. Cada entrada contiene la siguiente información sobre cada proceso:

- El PID del proceso y el PID de su proceso padre (PPID).
- El identificador de usuario real (UID) y efectivo y el identificador de grupo (GID).
- El estado del proceso.
- Puntero a la tabla de regiones del proceso.
- Información sobre las señales pendientes de ser enviadas al proceso, así como máscaras que indican qué señales son aceptadas y cuáles son ignoradas
- Parámetros de planificación (prioridad del proceso, uso de CPU consumido recientemente, etc.), que permiten al *kernel* decidir sobre qué proceso es el que se ha de ejecutar

La tabla de procesos reside en memoria y contiene información sobre todos los procesos, aunque éstos no estén en memoria. Las entradas en la **tabla de regiones del proceso** apunta a entradas de la **tabla de regiones**, que es global. Una región o segmento es una zona contigua del espacio de direcciones del proceso, y puede contener el código, los datos o la pila. Cada entrada de la tabla de regiones describe los atributos de la región, tales como si contiene código o datos, si es compartida o privada, o la localización de la región en memoria.

El nivel extra de indirección que supone acceder a la tabla de regiones por medio de la tabla de regiones por proceso tiene como fin el permitir que procesos diferentes puedan compartir regiones.

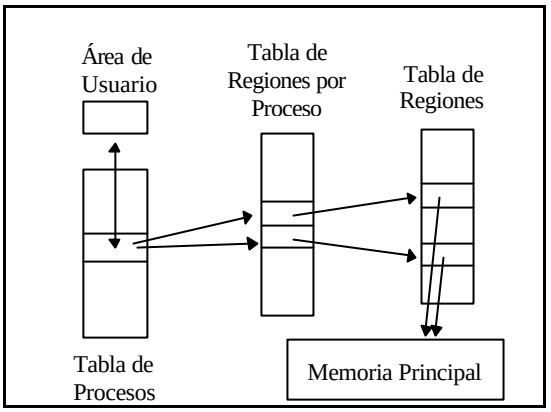

Figura 5.- Estructuras de datos para la gestión de procesos.

La estructura o área de usuario es una extensión de la tabla de procesos. Cada proceso tiene una estructura de usuario privada, que contiene información que no es necesaria cuando el proceso no está físicamente en memoria. Por ejemplo, cuando un proceso reside en disco puede recibir señales, pero obviamente no puede realizar operaciones de lectura sobre un fichero. Por este motivo, la información sobre las señales se almacena en la tabla de procesos, que siempre está en memoria, y la información sobre descriptores de ficheros se almacena en la estructura de usuario, que sólo es utilizada por el *kernel* cuando el proceso está en memoria y se está ejecutando.

La estructura de usuario contiene, entre otra, la siguiente información:

- Un puntero a la entrada de la tabla de procesos del proceso que se está ejecutando.
- Parámetros de la llamada al sistema que se está ejecutando, incluyendo los parámetros de la misma y los resultados.
- Tabla de descriptores de ficheros.
- Directorio actual y directorio raiz.

El *kernel* puede acceder directamente a los campos del área de usuario del proceso que está en ejecución, pero no al área de usuario de otro proceso Para el *kernel* es como si únicamente existiese una única área de usuario en el sistema, que se corresponde con la del proceso en ejecución. Para identificar *kernel* no tiene más que seguir el puntero del área de usuario que indica la entrada de la tabla de procesos de dicho proceso.

6.3.2. Contexto de un proceso

El contexto de un proceso es el conjunto del contenido de su espacio de direcciones, los contenidos de los registros hardware y las estructuras del *kernel* relativas al proceso. Concretamente, es el conjunto de los siguientes elementos:

- Segmentos de código, datos y pila.
- Valores de las variables globales de usuario y sus estructuras de datos.
- Valores de los registros de la CPU (contador de programa, puntero de la pila, etc.).
- Valores almacenados en su entrada de la tabla de procesos y en su área de usuario.
- Pila de *kernel*.

El código del sistema operativo y sus estructuras de datos globales son compartidos por todos los procesos, pero no forma parte del contexto de los mismos. Cuando se ejecuta un proceso se dice que el sistema se está ejecutando en el contexto de dicho proceso. Cuando el *kernel* decide ejecutar otro proceso realiza un **cambio de contexto**. Un cambio de contexto requiere que el *kernel* almacene la información necesaria para posteriormente poder ejecutar el proceso original. De forma similar, cuando un proceso pasa de modo usuario a modo *kernel*, éste almacena la información necesaria para posteriormente poder volver a modo usuario. Los cambios de modo usuario a modo *kernel* y viceversa se denominan **cambios de modo**.

6.3.3. Estados de un proceso

El tiempo de vida de un proceso se puede dividir en un conjunto de estados, cada uno con unas características propias. Los estados por los que puede atravesar un proceso se muestran en el diagrama Figura 6.

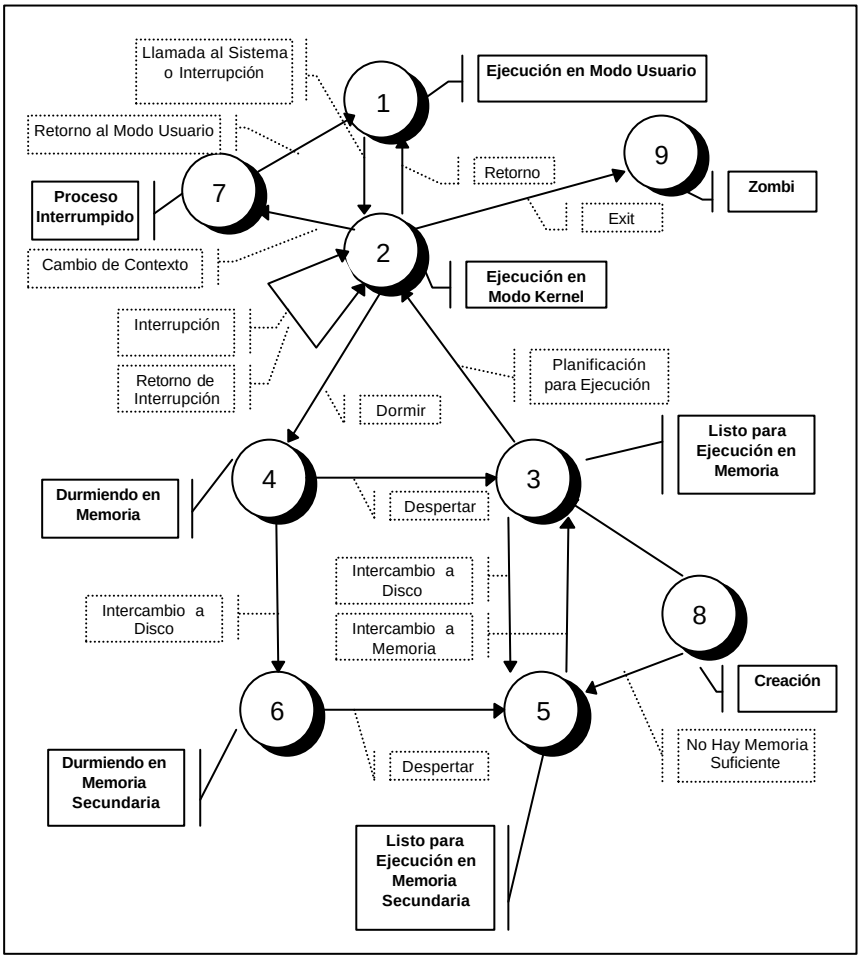

Figura 6.- Diagrama de estados de un proceso.

Los estados son los siguientes:

1. El proceso está en ejecución en modo usuario.
2. El proceso está en ejecución en modo *kernel*.
3. El proceso no está en ejecución pero está listo para ser ejecutado cuando lo decida la estrategia de *kernel*.
4. El proceso está durmiento y reside en memoria.
5. El proceso está listo para ser ejecutado, pero el proceso *swapper* (proceso 0) debe de llevar el proceso a memoria antes de que el *kernel* pueda planificar su ejecución.

6. El proceso está durmiendo y el *swapper* ha llevado el proceso a disco para permitir que otros procesos quepan en la memoria principal.
7. El proceso está pasando de modo *kernel* a modo usuario, pero el *kernel* lo interrupe (*preemption*) y realiza un cambio de contexto para ejecutar otro proceso.
8. El proceso acaba de ser creado, pero no está listo para ser ejecutado ni está durmiendo. Este estado es el estado de inicio de todo proceso, excepto el proceso 0.
9. El proceso ha ejecutado la llamada al sistema `exit` y entra en estado *zombie*. En este estado el proceso no existe, pero mantiene un registro que ha de ser leído por su proceso padre, y contiene un código de salida y algunas estadísticas. Este es el estado final de todo proceso.

Como un procesador únicamente puede ejecutar un proceso en un instante dado, únicamente puede haber un proceso en los estados 1 o 2.

Un proceso que se está ejecutando en modo usuario pasa a modo *kernel* cuando realiza una llamada al sistema o cuando el reloj del sistema interrumpe a la CPU. En este último caso, cuando el manejador de la *kernel* puede volver a ejecutar el mismo proceso en modo usuario o puede decidir planificar a otro proceso para ejecución (realmente, los estados 3 y 7 son el mismo estado). Eventualmente, el planificador elegirá el proceso para ejecución, con lo que pasará al estado 1, de

Cuando un proceso ejecuta una llamada al sistema abandona el estado de ejecución en modo usuario y pasa al estado de ejecución en modo *kernel*. Si la llamada al sistema requiere realizar una operación de entrada/salida, el proceso debe esperar a que ésta se realice y pasa al estado 4 (durmiendo en memoria). Cuando la operación de entrada/salida termina, el hardware interrumpe la CPU y el manejador de interrupciones despierta el proceso, que pasa al estado de listo para ejecución en memoria (estado 3).

swapper puede decidir que un proceso sea intercambiado a disco para permitir que algún proceso que esté en el estado 5 (listo para ejecución en disco) pueda pasar el

Un proceso tiene control sobre algunas transiciones entre estados cuando se ejecuta en modo usuario. En primer lugar, cuando un proceso crea otro proceso, el nuevo proceso se encuentra en el estado 8. En segundo lugar, un proceso puede entrar voluntariamente en estado de ejecución en modo *kernel* (estado 2) mediante una llamada al sistema. Por último, un proceso puede terminar voluntariamente mediante la llamada al sistema `exit`, con lo que pasa al estado *zombie*, aunque acontecimientos externos pueden forzar al proceso a terminar sin que éste invoque la llamada al sistema `exit`. El resto de las transiciones siguen un modelo rígido establecido por el *kernel*.

Muchos procesos se pueden ejecutar de forma concurrente y muchos de ellos pueden estar ejecutándose en modo *kernel*. Si se permitiese la ejecución en modo *kernel* sin ningún tipo de restricción, los procesos podrían corromper estructuras de datos globales del *kernel*. Esto se soluciona prohibiendo cambios de contexto arbitrarios e inhabilitando las interrupciones cuando un proceso está dentro de una *kernel* permite un cambio de contexto únicamente cuando un proceso pasa del estado 2 (ejecución en modo *kernel*) al 4 (durmiendo). Los procesos en ejecución en modo *kernel* no pueden ser apropiados (*preempted*) por otros procesos. De esta forma se resuelve el problema de la exclusión mutua dentro del *kernel*.

6.3.4. Interrupciones

Cuando un proceso realiza una llamada al sistema, la ejecución del proceso no puede ser interrumpida, tal como se vio en la sección anterior. Esto significa que el planificador de procesos no puede asignar la CPU a otro proceso durante la ejecución de la llamada al sistema. Sin embargo, algunas llamadas al sistema realizan operaciones de entrada/salida, que pueden tardar en realizarse una cantidad de tiempo apreciable. Para evitar que la CPU esté inactiva durante la realización de una operación de entrada/salida, el *kernel* envía al proceso al estado 4 del diagrama de estados (dormir en memoria) y no lo despierta hasta que no se produce una interrupción hardware indicando que la operación ha finalizado. Mientras el proceso está durmiendo, el planificador puede asignar a la CPU a otro proceso con la certeza de que no se producirán problemas, ya que los algoritmos codificados en el *kernel* aseguran que las estructuras de datos globales están en estado consistente cuando un proceso pasa al estado de dormir.

Las interrupciones son el mecanismo por el cual los dispositivos hardware notifican al *kernel* que requieren atención por parte de éste. De la misma forma que los procesos compiten por la CPU, los dispositivos hardware también compiten por el procesamiento de interrupciones. Los dispositivos tienen asignados niveles de prioridad basados en la importancia de los mismos. Por ejemplo, las interrupciones procedentes del reloj del sistema tiene prioridad sobre las originadas por el teclado.

Cuando se produce una interrupción, el proceso en ejecución es suspendido y el *kernel* determina la fuente de la interrupción. Cuando el *kernel* recibe una interrupción obtiene un número que se usa para acceder a una tabla denominada **vector de interrupciones**. Cada entrada suele contener la dirección de la rutina o procedimiento manejador de interrupción correspondiente a la interrupción recibida. Esta rutina es ejecutada y, cuando finaliza, el proceso puede continuar su ejecución. El procesamiento de una interrupción puede ser, a su vez, ser interrumpido por la llegada de otra de mayor prioridad. Sin embargo, si durante la ejecución del manejador de una interrupción se produce otra de igual o menor prioridad, ésta es ignorada y olvidada. Por este motivo, los manejadores de interrupciones han de ser diseñados para que sean muy rápidos, de forma que la posibilidad de que pierda una interrupción sea lo menor posible.

6.3.5. Planificación de procesos

Al ser UNIX un sistema de tiempo compartido, su algoritmo de planificación tiene como función principal el proporcionar un buen tiempo de respuesta a los procesos interactivos. El algoritmo de planificación es ***round-robin*** con múltiples colas, cada una de las cuales tiene asociada un nivel de prioridad. Los valores bajos indican mayor prioridad y los valores altos menor prioridad. Cada cola es una lista enlazada de los procesos listos para ser ejecutados que tienen la misma prioridad. Únicamente los procesos que están en memoria y están preparados para su ejecución están en estas colas.

Las prioridades de los procesos que se ejecutan en modo usuario se encuentran por encima de un valor umbral, mientras que las de los procesos que se ejecutan en modo *kernel* se encuentran por debajo de dicho umbral. Por ejemplo, si la prioridad umbral es cero, los procesos que se ejecutan en modo usuario tienen valores positivos, mientras que los procesos en modo *kernel* tienen valores negativos.

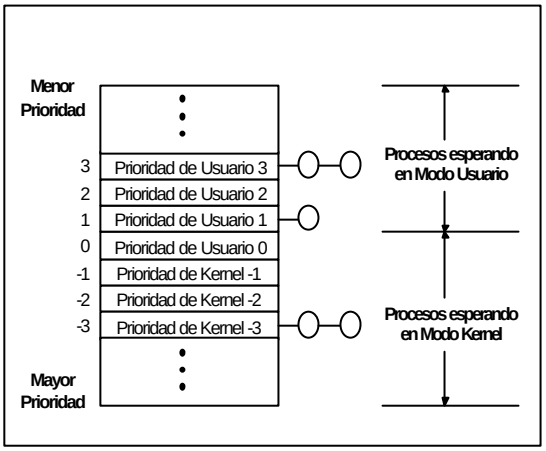

Figura 7.- Esquema de prioridades en UNIX.

A los procesos que se van a ejecutar se les asigna un tiempo de CPU denominado **quantum**. Un proceso se ejecuta hasta que consume su quantum de tiempo o se bloquea voluntariamente. La duración del quantum suele ser de 100 milisegundos.

El algoritmo de planificación de procesos es el siguiente:

- Cada segundo el planificador calcula las prioridades de todos los procesos del sistema que estan listos para ser ejecutados y los organiza entre las diferentes colas.
- Cada décima de segundo, el planificador selecciona el proceso de mayor prioridad y le asigna la CPU.
- Si un proceso consume su quantum de tiempo es colocado al final de su cola de prioridad, siguiendo la estrategia *round-robin*.
- Si un proceso pasa al estado de dormido durante su quantum de tiempo, el planificador selecciona inmediatamente otro proceso y le asigna la CPU.
- Si un proceso acaba una llamada al sistema durante su quantum de tiempo y un proceso de mayor prioridad está listo para ser ejecutado, al proceso de menor prioridad se le expropia la CPU y se le asigna al de mayor proridad.
- Cada vez que se produce una interrupción del reloj del sistema (*tick* de reloj), el contador del uso de CPU del proceso se incrementa.

El sistema operativo mide el tiempo mediante *ticks* de reloj, que son interrupciones hardware de la CPU que ocurren a intervalos fijos, normalmente 50 ó 60 veces por segundo, por lo que un quantum suele durar 5 o 6 *ticks* de reloj.

En UNIX System V las prioridades se recalculan cada segundo usando, en primer lugar, una función de envejecimiento que divide el uso de CPU de los procesos por 2, y, en segundo lugar, la siguiente fórmula:

```
prioridad = prioridad_base + (uso_de_CPU/2)
```

La prioridad base suele ser cero, pero puede alterarse para favorecer (base negativa) o penalizar (base positiva) la prioridad del proceso mediante la llamada al sistema `nice`. Únicamente el superusuario puede asignar una prioridad base negativa. En este caso, la fórmula para calcular prioridades sería igual a la anterior más la suma del valor que se pasa mediante `nice`. El efecto neto de recalcular las prioridades es el movimiento de los procesos de unas colas de prioridad a otras.

Problema:

Obtener la planificación de tres procesos A, B, y C en UNIX System V asumiendo las siguientes suposiciones:

- los tres procesos se crean a la vez y con prioridad inicial 60.
- el mayor nivel de prioridad a nivel de usuario es 60.
- el tick de reloj es cada 1/60 segundos.
- los procesos no hacen llamadas al sistema.
- no existen otros procesos preparados para ejecutarse.

6.4. Sistema de Ficheros

El sistema de ficheros de UNIX se basa en dos tipos de objetos: ficheros y directorios. Los directorios son también ficheros, pero tienen un formato especial.

Los sistemas de ficheros suelen estar situados en dispositivos de almacenamiento modo bloque, como discos o cintas. Un mismo sistema UNIX puede tener varios discos físicos, cada uno de los cuales puede contener uno o varios sistemas de ficheros. En este caso, los sistemas de ficheros son particiones lógicas del disco. El *kernel* trabaja con el sistema de ficheros a nivel lógico. Será el manejador (*driver*) del disco el que transforme las direcciones lógicas a físicas.

Un sistema de ficheros se compone de una secuencia de bloques lógicos, cada uno de los cuales tiene longitud fija. El tamaño de cada bloque es el mismo en todo el sistema de ficheros y suele ser múltiplo de 512 bytes. El tamaño del bloque influye en las prestaciones del sistema y en el aprovechamiento del espacio en disco. Si el tamaño de bloque es grande, aumenta la velocidad de transferencia entre disco y memoria, pero aumenta el espacio desperdiciado debido a que el último bloque de un fichero no suele estar lleno (fragmentación interna).

6.4.1. Estructura del Sistema de Ficheros

Todos los sistemas de ficheros de UNIX System V presentan la estructura que aparece en la Figura 8. El bloque de arranque suele ser el bloque 0 de la pista 0 de la superficie 0 de un disco y no es usado por UNIX. Contiene un programa que se cargar en memoria cuando se arranca el sistema y tiene como misión cargar el sistema operativo. El superbloque está a continuación del bloque de arranque y contiene información crítica sobre el sistema de ficheros:

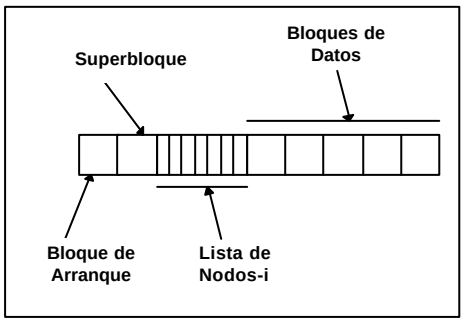

Figura 8.- Estructura de un sistema de ficheros.

- Tamaño del sistema de ficheros (en bloques).
- Número de bloques libres.
- Lista de bloques libres.
- Comienzo de la lista de bloques libres
- Número de nodos-i.
- Número de nodos-i libres.
- Lista de nodos-i libres.

- Comienzo de la lista de nodos-i libres.
- Campos para bloquear la lista de bloque libres y la lista de nodos-i libres.
- Un indicador que indica si el superbloque ha sido modificado.

La lista de nodos-i (nodos índice o ***inodes***) sigue al superbloque y su tamaño lo determina el administrador del sistema cuando configura el sistema de ficheros. Por último, los bloques de datos contienen los ficheros y directorios del sistema de ficheros.

El superbloque y la lista de nodos-i residen de forma permanente en disco, pero durante el proceso de arranque del sistema operativo y con el fin de aumentar el rendimiento del sistema, el *kernel* los copia en memoria. Esto da lugar a que produzcan inconsistencias de datos, ya que las modificaciones que se realizan en memoria no son realizadas inmediatamente sobre la copia en disco. Para solucionar este problema existe un *daemon*, llamado *syncer*, que periódicamente actualiza la información en disco.

6.4.2. Nodos-i

La representación interna de un fichero en UNIX está determinada por un nodo-i, que contiene la descripción de los bloque de disco que forman el fichero y otro tipo de información (atributos), como el propietario de fichero, permisos de acceso, etc. Los nodos-i están numerados y cada uno de ellos ocupa 64 bytes. El nodo-i número 2 se corresponde con el del directorio raíz.

fichero, y un fichero se identifica, independientemente de su nombre, por el número de su nodo-i. Todo fichero tiene asociado un nodo-i, pero puede tener varios nombres, cada uno de los cuales se refieren al mismo nodo-i. Cada nombre se denomina **enlace** (*link*).

Un nodo-i en disco contiene la siguiente información:

- Identificador del propietario del fichero.
- Identificador del grupo.
- Tipo de fichero.
- Permisos de acceso.
- Fecha y hora de última modificación, último acceso y última modificación del nodo-i.
- Número de enlaces del fichero.
- Tamaño del fichero, si es ordinario o directorio, o descripción del dispositivo, si es un fichero especial.
- Tabla de direcciones de los bloques de datos del fichero.

La modificación del contenido de un fichero implica modificar el contenido de su nodo-i, pero un nodo-i se puede modificar sin que se modifique el fichero. Cuando el nodo-i está en memoria posee algunos campos de información adicionales, que son los siguientes:

- El estado del nodo-i en memoria, que indica si
  - el nodo-i está bloqueado,
  - un proceso espera a que el nodo-i se desbloquee,
  - un indicador que indica si nodo-i en memoria difiere del mismo en disco debido a una cambio de los datos del nodo-i,
  - un indicador que indica si nodo-i en memoria difiere del mismo en disco debido a una cambio de los datos del fichero,
  - el fichero es un punto de montaje.
- El número de dispositivo lógico del sistema de ficheros que contiene el nodo-i.
- El número de nodo-i, que identifica la posición del nodo-i dentro de la lista de nodos-i en disco. El nodo-i en disco no necesita este campo.
- Punteros a otros nodos-i en memoria.
- Un contador de referencias, que indica el número de instancias del fichero que están activas.

6.4.3. Estructura de un fichero regular

Un nodo-i contiene una tabla de los bloques de datos del fichero en disco. Como cada bloque en disco se direcciona por su número, esta tabla consiste en un conjunto de números de bloques de disco. Si los ficheros de almacenasen de forma contigua, sería suficiente con almacenar la dirección del primer bloque y el tamaño del fichero en el nodo-i para poder acceder a todos los bloques del fichero. Este esquema

kernel contiene un algoritmo que convierte un desplazamiento en bytes a un número de bloque y un desplazamiento dentro de ese bloque.

Problema:

UNIX System V utiliza bloques de datos de 1024 bytes y se requieren 32 bits para direccionar cada bloque. Determinar el tamaño máximo de puede tener un fichero utilizando los 10 punteros directos, el puntero 11, el 12 y el 13.

6.4.4. Directorios

Los directorios, al igual que los ficheros regulares, están representados por un nodo-i y su contenido se almacena en bloques de datos. Únicamente el campo de tipo de fichero en el nodo-i permite distinguir entre ficheros y directorios. Sin embargo, mientras que los ficheros regulares no tienen una estructura (en el sentido de que consideran como meras secuencias de bytes), los directorios tienen una estructura específica. En las primeras versiones de UNIX, los nombres de ficheros estaban limitados a un máximo de 14 caracteres, por lo un directorio era una lista de grupos de 16 bytes: 2 para el número de nodo-i y 14 para el nombre del fichero. A partir de la versión 4.2BSD, los nombres de fichero en UNIX son de longitud variable y pueden tener hasta 255 caracteres, por lo que las entradas de directorio son, asimismo, de longitud variable. Cada entrada contiene la longitud de la misma, el nombre del fichero y el número de nodo-i. Este esquema es más complejo pero permite mucha mayor flexibilidad al usuario a la hora de elegir nombres significativos para los ficheros.

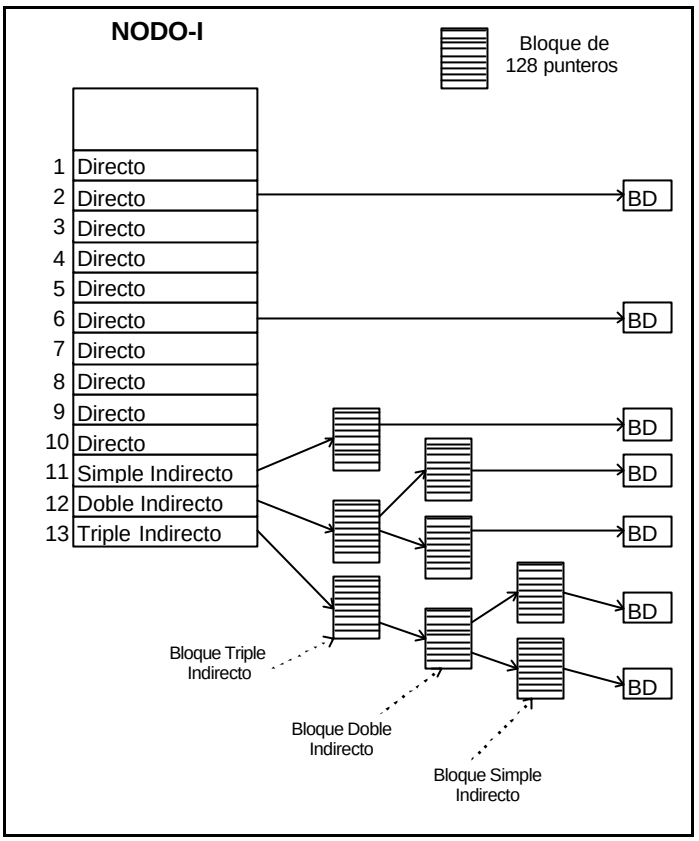

Figura 9.- Estructura de un fichero regular.

Cuando se crea un directorio, automáticamente se le asignan entradas para el directorio padre (conocido como '`..`') y él mismo (denominado '`.`'). Cada par [nombre de fichero, número de nodo-i] vincula un nombre a un fichero y es lo que se conoce como enlace (*hard link*).

Muchas llamadas al sistema obtienen el nodo-i de un fichero a través de su ruta (*pathname*) siguiendo los siguientes pasos:

1. Se busca el nodo-i del primer elemento de la ruta. Si se trata del directorio raiz (conocido como '`/`'), la búsqueda comienza en el nodo-i número 2. Si la ruta es relativa, la búsqueda comienza en el nodo-i correspondiente al directorio actual del proceso.
2. Los componentes de la ruta son procesados de izquierda a derecha. Todo componente menos el último se debe corresponder con un directorio o un enlace simbólico.
3. Si el nodo-i que se está procesando se corresponde con un directorio, el componente de la ruta se busca en el directorio correspondiente a dicho nodo-i. Si no se encuentra, se produce un error. En caso contrario, el nodo-i a procesar pasa a ser el nodo-i asociado con el componente localizado de la ruta.
4. Si el nodo-i que se está procesando se corresponde con un enlace simbólico, se sustituye la parte de la ruta que comprende el principio de la misma hasta el componente actual por el contenido del enlace simbólico, y la ruta es reprocesada.
5. El nodo-i correspondiente al último componente de la ruta es el del fichero referenciado por la ruta completa.

Por ejemplo, supongamos que queremos traducir el nombre "`/bin/ls`" a su número de nodo-i. Los pasos serían los siguientes:

1. La ruta es absoluta, luego el nodo-i de partida es el número 2.
2. En el directorio correspondiente al nodo-i número 2 se busca el componente "`bin`". Supongamos que la entrada es localizada y que indica que el nodo-i asociado a dicha entrada es el 8.

3. En el directorio correspondiente al nodo-i número 8 se busca el componente "`ls`". Supongamos que se encuentra y que el nodo-i correspondiente es el 25.
4. "`ls`" es el último componente de la ruta, por lo que el algoritmo devuelve el nodo-i número 25.

6.4.5. El Sistema de Ficheros de Berkeley

Las versiones de UNIX de la universidad de Berkeley aportaron nuevas características al sistema de La primera variación que tiene BSD UNIX es modificar los directorios mediante el aumento del tamaño máximo de los nombres de los ficheros de 14 caracteres a 255.

En segundo lugar, se divide cada disco en grupos de cilindros, cada uno con su superbloque, nodos-i y bloques de datos. Con este esquema se trata de mantener los bloques de nodos-i y bloques de datos físicamente cercanos para reducir los tiempos de búsqueda.

Por último, se establecen dos tamaños posibles de bloque de datos, en lugar de uno. De esta forma, se emplean bloques de mayor tamaño para ficheros grandes, lo que es más eficiente que usar bloques pequeños. Por otro lado, la mayoría de los ficheros suelen ser pequeños, para los cuales se emplea el tamaño de bloque menor, con la consiguiente reducción de la pérdida de espacio en comparación con el uso de bloques de gran tamaño. El inconveniente de este método es la complejidad de su implementación.

6.4.6. Tablas de control de acceso a ficheros

La copia de la lista de nodos-i en memoria se denomina **tabla de nodos-i**. Además de ésta, el *kernel* mantiene en memoria otras dos tablas que contienen información necesaria para la gestión de los ficheros: la **tabla de ficheros** y la **tabla de descriptores de ficheros**. La tabla de ficheros es global y la tabla de descriptores de ficheros es local a cada proceso. El motivo de usar tres tablas es permitir varios grados de acceso compartido a un mismo fichero.

Cada proceso tiene una tabla de descriptores de ficheros a la que se accede mediante un descriptor de fichero, y contiene un entrada para cada fichero abierto por el proceso (hasta un máximo de 20 entradas). La idea es que el proceso tenga que acceder al nodo-i del fichero en cuestión a través de un descriptor de fichero.

Cuando un proceso abre o crea un fichero, el *kernel* asigna una entrada para cada una de las tablas. Concretamente, el *kernel* devuelve un descriptor de fichero para el fichero creado o abierto, que es un índice a la tabla de descriptores de ficheros del proceso. La entrada de la tabla de descriptores de fichero apunta a una entrada en la tabla de ficheros, que contiene, entre otra información, el desplazamiento en bytes del fichero a partir del cual comenzará la siguiente operación de lectura o escritura, así como los derechos de acceso permitidos sobre el fichero que se ha abierto. Cada entrada de la tabla de ficheros apunta a una entrada de la tablas de nodos-i.

Concretamente, cuando un proceso ejecuta una llamada al sistema `open` o `create`, el *kernel* devuelve un descriptor de fichero. A continuación, cuando se invocan las llamadas `read` o `write`, el *kernel* usa el descriptor de fichero para acceder a la tabla de descriptores de fichero del proceso, sigue los punteros a la tabla de ficheros y a la tabla de nodos-i y, a partir de éste, accede a los datos del fichero.

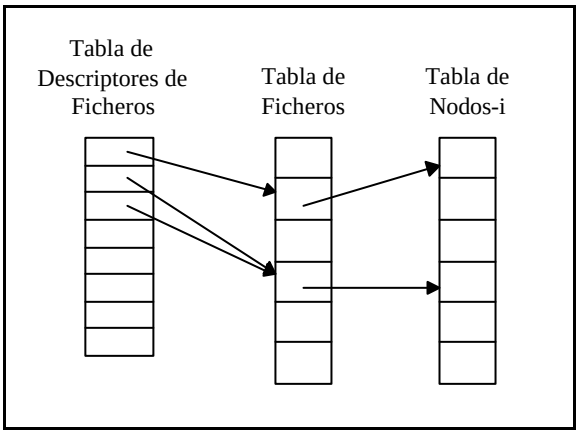

Figura 10.- Tablas de control de acceso a ficheros.

Cada entrada de la tabla de ficheros contiene, entre otros campos, un puntero a una estructura de datos que contiene información sobre el estado actual del fichero:

- Nodo-i asociado a la entrada de la tabla.
- Desplazamientos (*offsets*) de lectura y escritura, que indican sobre qué byte del fichero van a tener efecto las siguientes operaciones de lectura y escritura.
- Permisos de acceso para el proceso que ha abierto el fichero.

- *Flags* (indicadores) del modo de apertura del fichero, que se verificarán cada vez que se realice una

- Un contador que indica el número de entradas de tablas de descriptores que tiene asociada esta entrada de la tabla de ficheros.

Cuando se crea un proceso UNIX el sistema abre, por defecto, tres ficheros que ocupan las tres primeras entradas de la tabla de descriptores:

- entrada estándar (descriptor 0)
- salida estándar (descriptor 1)
- error estándar (descriptor 2)

Supongamos que un proceso A quiere abrir el fichero */etc/passwd* dos veces, una para lectura y otra para escritura, y el fichero *.cshrc* para lectura y escritura. Para ello, ejecutaría las siguientes llamadas al sistema:

```
fd1 = open("/etc/passwd", O_RDONLY) ;
fd2 = open(".cshrc", O_RDWR) ;
fd3 = open("/etc/passwd", O_WRONLY) ;
```

A continuación, otro proceso B abre el fichero */etc/passwd* para lectura:

```
fd1 = open("/etc/passwd", O_RDONLY) ;
```

Las relaciones entre las diferentes tablas se muestra en la Figura 11. La justificación de este esquema con tres tablas es la siguiente. Cada descriptor de fichero debe de tener asociado el desplazamiento (*offset*) que indica la siguiente posición del fichero sobre la cual leer o escribir. Si no existiese la tabla de ficheros, una posibilidad sería incluir el desplazamiento en el nodo
válida cuando dos procesos no relacionados abren un mismo fichero, ya que cada uno debe de tener su propio valor de desplazamiento (ver el ejemplo del fichero */etc/passwd*, en la Figura 11). La otra posibilidad sería situar el desplazamiento en el descriptor de fichero, pero esto impediría compartir ficheros entre procesos que tienen una relación padre-hijo. Supongamos que el proceso A comienza a escribir en el fichero *.cshrc* y, cuando termina de escribir, crea un proceso hijo C para que continúe escribiendo en dicho fichero a partir de la última posición escrita por el proceso padre. Las relaciones entre las distintas tablas después de la creación del proceso C se muestra en la Figura 12. Ambos procesos comparten las mismas entradas en la tabla de ficheros, que es la que almacena los desplazamientos, por lo que cuando el proceso C comience a escribir en el fichero *.cshrc*, lo hará a partir del lugar en el que terminó de escribir el proceso A.

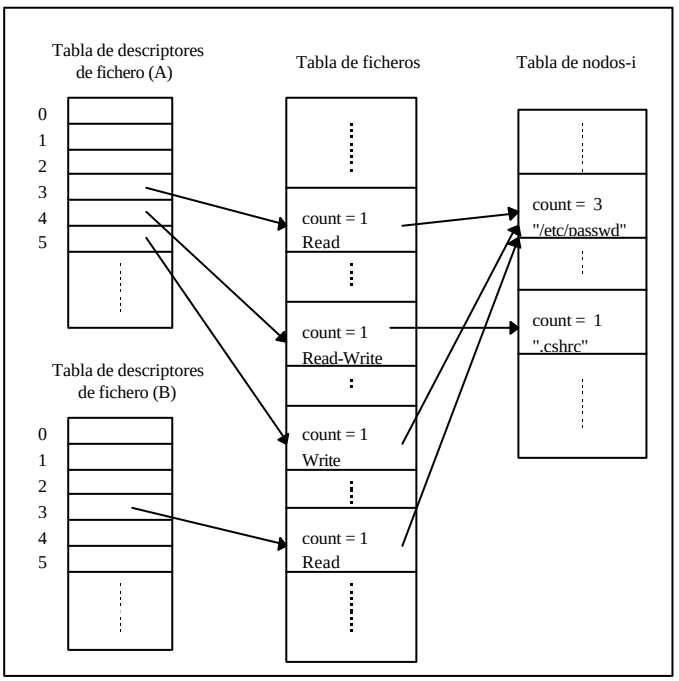

Figura 11.- Relaciones entre las tablas después de las operaciones de los procesos A y B.

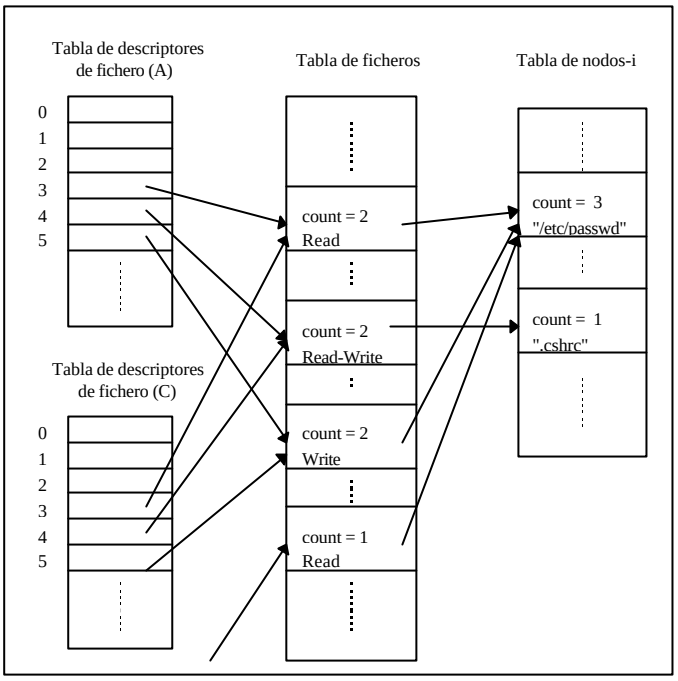

Figura 12.- Relaciones entre las tablas después de las crear el proceso C.

6.5. Gestión de Memoria

Las primeras versiones de UNIX se ejecutaban sobre máquinas que tenían muy poca memoria física. Debido a la poca cantidad de recursos disponibles, era injustificado el empleo de complejos algoritmos de gestión de memoria. Por este motivo, la mayoría de las versiones de UNIX anteriores a 3BSD gestionaban la memoria utilizando un simple mecanismo de intercambio o *swapping*: si en un instante dado existen más procesos de los que caben físicamente en memoria, algunos son llevados a disco (*swapped out*). Los procesos son intercambiados enteros a disco, de forma que un proceso o está en memoria o está en disco (excepto el segmento de código, en el caso de que esté compartido).

6.5.1. Intercambio (*Swapping*)

El movimiento entre memoria y disco es gestionado por un proceso planificador conocido como intercambiador o *swapper*. Este proceso tiene el PID 0, y despierta al menos una vez cada cuatro segundos para verificar qué procesos hay que descargar o reincorporar, en el caso de que sea necesario. Cuando por algún motivo es necesario intercambiar un proceso a disco (no se pueden crear nuevos procesos, un proceso pide memoria, un proceso lleva mucho tiempo en disco y es necesario hacerle sitio en memoria, etc.) el *swapper* tiene que decidir qué proceso ha de ser intercambiado. El método de elección busca entre aquellos procesos que están bloqueados, han estado en memoria más tiempo o si son grandes. Para reincorporar un proceso a memoria se busca entre aquellos que llevan más tiempo en disco o son pequeños. Para impedir que se produzca trasiego (*thrashing*) se prohibe que un proceso sea descargado a disco antes de que lleve un cierto tiempo en memoria Si el *swapper* elige un proceso para ser reincorporado a memoria y no existe suficiente espacio, intercambia procesos a disco hasta que haya el espacio libre necesario.

Para aumentar la eficacia del intercambio, los segmentos de datos del proceso a nivel de *kernel* (área de usuario y pila de *kernel*) y los segmentos de datos a nivel de usuario (código, pila, datos) se almacenan contiguamente en memoria. El inconveniente de esta solución es que la fragmentación externa puede originar problemas.

6.5.2. Paginación

En la actualidad, tanto las versiones de UNIX basadas en 4BSD como System V emplean, además del intercambio, políticas de demanda de página. La idea se basa en el hecho de que un proceso no necesita estar entero en memoria para poder ejecutarse. Únicamente se requieren la estructura de usuario y la tabla de páginas. Cuando un proceso necesita una página que no está en memoria se produce un fallo de página, se asigna una celda en memoria principal y se lee a ella la página adecuada desde disco. De este forma, se elimina el problema de la fragmentación externa (la fragmentación interna existe, pero es despreciable cuando se eligen tamaños de página razonables). La frecuencia del proceso de intercambio queda reducida a un mínimo porque con paginación caben más procesos en memoria.

La demanda de páginas se realiza de la siguiente forma:

1. cuando un proceso hace referencia a una página y ésta no está en memoria, se produce una falta
2. el proceso pasa a modo *kernel*,
3. el *kernel* asigna un marco de página en memoria principal, y
4. la página es leída de disco a ese marco de página.

BSD UNIX no usa el modelo de espacio de trabajo porque este método requiere conocer las páginas que están en uso y cuales no, lo que no ofrecen los sistemas VAX (sobre los que se implementó 3BSD) mediante hardware (no tiene bits de página referenciada o usada). La paginación la realiza el *kernel* y un proceso llamado demonio de páginas (*page daemon* o *page stealer*), que es el proceso 2. Cuando este demonio despierta comprueba si el número de celdas o marcos de página en memoria principal es menor que un cierto umbral llamado *lostfree*, que típicamente es 1/4 de la memoria. Si ello ocurre, el demonio de páginas aplica el algoritmo de reemplazamiento de páginas para transferir páginas de memoria a disco hasta que *lostfree* marcos de página quedan libres. Si hay más marcos libres que *lostfree*, el demonio de páginas no es despertado. El algoritmo de reemplazamiento original en 4BSD es una variante del LRU, con reloj simple global. La memoria principal en 4BSD se divide en tres partes:

- ***kernel***: memoria del *kernel*
- ***core map***: contiene información sobre el contenido de los marcos de página
- **marcos de página**: celdas en las que colocan las páginas de memoria virtual.

Tanto la zona de *kernel* como la de *core map* siempre están en memoria, nunca se pueden llevar a disco. El mecanismo del reloj consiste en recorrer los marcos de página de la memoria de forma lineal y continua. En una primera pasada, cuando la manecilla del reloj apunta a un marco de página, el bit de uso es puesto a cero. Si en la siguiente pasada el bit de uso sigue a cero, el marco de página será añadido a la lista de marcos libres. El marco de página retiene su contenido original, que puede ser recuperado si la página es necesitada de nuevo. Los sistemas VAX no tienen bits de página referenciada por hardware, por lo que cuando la manecilla del reloj apunta a un marco de página en la primera pasada, un bit de página referenciada software es puesto a cero y la página es marcada como inválida en la tabla de páginas. Cuando la página vuelva a ser accedida ocurrirá una falta de página, lo que permitirá poner a uno el bit de uso software. Así se consigue el mismo efecto que con el bit de uso por hardware, pero a un coste mayor.

El mecanismo del reloj simple tiene el inconveniente de que si la memoria es muy grande se tarda mucho tiempo en realizar las pasadas. Por este motivo, 4BSD emplea un algoritmo LRU modificado con reloj de dos manecillas. Ahora, el *page daemon* tiene dos punteros de forma que se pone a cero el bit de uso de la página apuntada con la primera manecilla, a continuación se comprueba valor del bit de uso de la página apuntada con la segunda, y después se avanzan ambas manecillas. Si las dos manecillas se mantienen próximas únicamente las páginas muy usadas serán accedidas entre el paso de las dos manecillas. Si la segunda manecilla está 359 grados por detrás de la primera, el comportamiento es como el del mecanismo del reloj simple.

La paginación en System V presenta algunas diferencias respecto a 4BSD. Se usa el mecanismo del reloj simple, pero en lugar de añadir las páginas no usadas en la segunda pasada, lo hace si la página no ha sido usada en las siguientes n pasadas. En lugar de una variable *lostfree*, System V usa dos variables, *min* y *max*. Cuando el número de marcos de página libres es menor que *min*, el *swapper* es despertado y libera tantos marcos de página como sean necesario hasta que el número de marcos libres sea igual a *max*.

6.6. Entrada/Salida

Uno de los principales objetivos de diseño de un sistema operativo es ocultar las peculiaridades de los diferentes dispositivos hardware al usuario. Por ejemplo, el sistema de ficheros presenta al usuario una misma entidad lógica, el fichero, que es independiente del hardware de disco subyacente. En UNIX, las características específicas de los dispositivos de entrada/salida permanecen ocultas a través del sistema de entrada/salida del *kernel*.

El sistema de entrada/salida se compone del subsistema de gestión de buffers (*buffer cache*), del código general para manejo de dispositivos y de los manejadores (*drivers*) de dispositivos hardware específicos, que son los que conocen las peculiaridades de dichos dispositivos. Este esquema permite que la mayor parte del sistema de entrada/salida pueda estar en la parte del *kernel* que es independiente del hardware. Existe normalmente un manejador por dispositivo. Los manejadores se incluyen en el sistema operativo cuando se genera el *kernel*, por lo que no pueden ser añadidos o eliminados posteriormente.

El sistema de entrada/salida se subdivide en dos: gestión de dispositivos de bloques y gestión de dispositivos de caracteres. Las principales diferencias entre ambos radican en que los primeros usan *buffers* durante las operaciones de entrada y de salida, y, además, estas operaciones se realizan sobre secuencias de bytes (bloques). Por el contrario, los segundos no usan *buffers* y las operaciones de entrada y de salida se realizan carácter a carácter.

6.6.1. Gestión de los dispositivos de bloque

Los dispositivos de bloques incluyen a los discos y cintas, que se caracterizan por ser accedidos a través de bloques de tamaño fijo (normalmente, 512 bytes). Pueden ser utilizados a través de los ficheros especiales de dispositivo, aunque normalmente son accedidos indirectamente a través del sistema de ficheros.

El objetivo principal de la gestión de dispositivos de bloques es minimizar el número de transferencias entre un dispositivo y la memoria. Para conseguirlo, UNIX dispone de lo que se denomina *buffer cache* entre los manejadores de los discos y el sistema de ficheros, que es una tabla en el *kernel* que almacena algunos de los bloques usados más recientemente.

Cuando se requiere un bloque de disco se comprueba primero si está en el *buffer cache*. Si está se toma directamente y si no está se lee de disco al *buffer cache* y de aquí se copia donde sea necesario. En el *buffer cache* existe capacidad para un determinado número de bloques. Los bloques se suelen gestionar mediante una lista encadenada. Cuando se usa un bloque, se mueve a la cabeza de la lista. Si hay que eliminar un bloque para hacer sitio a uno nuevo, se toma el de la cola de la lista, que es el último en haber sido usado. Cuando un programa escribe un bloque, lo hace en el *buffer cache*, no en el disco. Sólo cuando la cache se llena y el buffer es reclamado, se escribe el bloque en disco. Para evitar que los bloques estén demasiado tiempo en la cache sin ser salvados a disco, los bloques modificados se escriben cada 30 segundos.

6.6.2. Gestión de los dispositivos de caracteres

Los dispositivos de caracteres no mueven bloques entre disco y memoria, por lo que no necesitan acceder al buffer cache. En su lugar se utilizan unas estructuras de datos llamadas listas-C (*C-lists*). Cada elemento de una lista-C es un pequeño bloque de caracteres (normalmente 28 bytes) más un contador y un puntero al siguiente bloque.

Cuando un proceso lee de un dispositivo de caracteres, los caracteres no se envían directamente de la lista-C al proceso, sino que pasan a través de un filtro del *kernel* llamado *line discipline* que procesa el flujo de caracteres tal como viene del dispositivo. Existen tres tipos básicos de procesamiento:

- modo *raw*: no se realiza ningún tipo de procesamiento. Se usa en aplicaciones como editores de texto, que gestionan por sí mismas el procesamiento de caracteres.
- modo *cbreak*: se procesan algunas combinaciones de teclas. Por ejemplo, *Control-C* genera una señal que se envía al proceso que se está ejecutando en primer plano.

- modo *cooked*: se procesan los caracteres tanto durante las operaciones de entrada como de salida. Por ejemplo, las pulsaciones de la tecla de borrado se tienen en cuenta. Únicamente cuando se pulsa le tecla de retorno de carro los caracteres de entrada son enviados al proceso.

6.7. Comunicación entre Procesos (*pipes*)

UNIX System V tiene dos tipos de *pipes*: *pipes* con nombre (*FIFOs*) y *pipes* sin nombre. Los primeros se crean mediante la llamada al sistema `mknod()` y los segundos mediante `pipe()`.

Los datos que se escriben en un *pipe* se almacenan en el sistema de ficheros, por lo que cuando se crea un *pipe* el *kernel* le asigna un nodo-i, además de dos entradas en la tabla de ficheros y dos entradas en la tabla de descriptores de fichero. Si el *pipe* es con nombre, se realiza un enlace al nodo-i de *pipe*. Si es sin nombre no se crea ningún enlace y el *pipe* permanece anónimo. El *kernel* mantiene el puntero de escritura y el puntero de lectura del *pipe* en el nodo-i en lugar de en la tabla de ficheros. El motivo es que de esta forma el *kernel* puede controlar que cada byte del *pipe* sea leído únicamente por un proceso. También permite conocer el número de procesos que leen y escriben en el *pipe*.

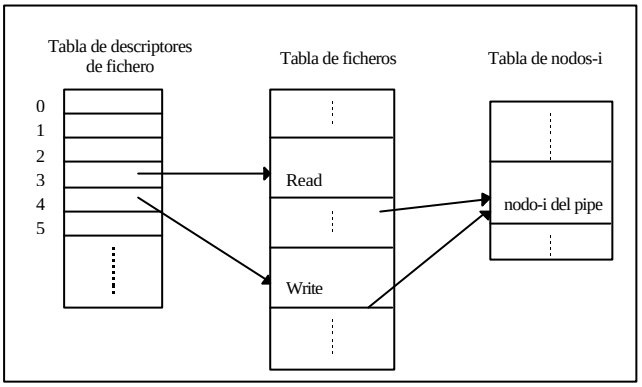

Figura 13.- Relaciones entre las tablas cuando se crea *pipe*.

Cuando se escribe en un *pipe*, el *kernel* asigna bloques de disco e incrementa el puntero de escritura. Por razones de simplicidad y eficiencia, a los *pipes* no les asigna bloques indirectos, por lo que el tamaño de un *pipe* está limitado a 40K (este límite depende del tamaño de bloque del sistema de ficheros). Si una escritura en un *pipe* sobrepasa este límite, el proceso que escribe se bloquea hasta que otro proceso lea. Cuando se llega al límite del *pipe*, se comienza a escribir por el principio del mismo, como si se tratase de un buffer circular. Cuando se realiza una operación de lectura, el *kernel* actualiza la posición del puntero de lectura, controlando que no sobrepase al de escritura. Si el *pipe* está vacío, el proceso que lee se bloquea hasta que otro escriba.

Cuando se cierra un descriptor de un *pipe*, el *kernel* hace lo siguiente:

1. Actualiza el contador de procesos lectores y escritores.
2. Si el contador de procesos escritores llega a cero y existen procesos que intentan leer, la siguiente lectura de éstos devolverá un error.
3. Si el contador de procesos lectores llega a cero y existen procesos que intentan escribir, el *kernel* les envía una señal.
4. Si ambos contadores llegan a cero, se liberan los bloque de disco y los punteros de lectura y escritura del nodo-i son puestos a cero. Si el *pipe* es sin nombre, se libera el nodo-i.

6.8. Referencias

- “Modern Operating Systems”. A.S. Tanenbaum. Prentice-Hall. 1992.
- “The Design of the UNIX Operating System”. M.J. Bach. Prentice-Hall. 1986.
- "Operating System Concepts". A. Silberschatz, P. B. Galvin. Addison-Wesley. 1994.
- "UNIX for Programmers and Users". G. Glass. Prentice-Hall. 1993.

Ejercicios

1. Un sistema UNIX posee un tamaño de bloque de 1024 bytes y usa 32 bits para direccionamiento de bloques. Sabiendo que las direcciones lógicas comienzan por cero, convertir los desplazamientos 9000 y 350.000 a direcciones físicas, indicando qué punteros directos e indirectos y desplazamiento dentro

2. Una versión de UNIX usa la siguiente fórmula de envejecimiento (*decay*):

   $$decay(CPU) = 0.8*CPU$$

   La prioridad de los procesos se recalcula cada segundo mediante la siguiente fórmula:

   $$prioridad = prioridad\_base + CPU / 16$$

   Planificar la ejecución de tres procesos A, B y C durante ocho segundos sabiendo que se crean en este orden con prioridad base 60, no se emplea quantum de tiempo, los *ticks* de reloj son cada 1/60 segundos, ningún proceso realiza llamadas al sistema y no existen otros procesos listos para ser ejecutados.

3. Realizar el ejercicio anterior con siete procesos usando las funciones de envejecimiento y cálculo de prioridades de UNIX System V suponiendo que se producen 100 *ticks* por segundo y que siempre que existen dos procesos con igual prioridad se elige para su ejecución el que creó en primer lugar. ¿Qué es lo que sucede?.