Diseño de Sistemas Operativos

Curso: 2001/2002
Cuatrimestre: Primero
Alumna: Laura M. Castro Souto
Profesor: Ramón Pérez Otero

Índice general

Capítulo 1

Introducción

Un **sistema operativo** es un programa cuya principal finalidad es hacer que la ejecución del resto de programas sea independiente del hardware. Para conseguirlo, utiliza dos conceptos básicos: el concepto de *fichero* y el concepto de *proceso*; trata cualquier tipo de dispositivo igual que si fuese un fichero, salvo dos excepciones: la CPU y la memoria, con los que emplea el concepto de proceso.

En cuanto a la *arquitectura*, las dos características principales son:

- Las llamadas al sistema.
- El kernel no tiene procesos asociados.

Los sistemas operativos han ido evolucionando:

↪ Se empezó "incluyendo" el SO[1] en la aplicación, esto es, eran las propias aplicaciones las que accedían directamente al hardware.

↪ Más tarde se crea una aplicación que es la única que interactúa con el hardware. El resto de aplicaciones trabajan sobre ella. Ello hace necesaria la presencia de una *interfaz*. El código del SO debe hacer lo que necesita la aplicación del usuario. Para ello, en teoría, la aplicación debe llamar a las funciones del SO, incluyéndolas, pues, en su código. Esto implicaría varias copias del SO.

↪ Se dieron dos soluciones a la duplicación del código del SO:

→ Existencia de un proceso independiente del resto para el kernel; los otros procesos se comunican con él pidiéndole la ejecución de funciones. Ejemplo: antiguo SO de IBM.

→ El propio kernel es un trozo único de código que se une a todos los programas. La llamada a funciones del kernel es más compleja, pues hay que acceder a una zona de código que no pertenece al proceso. Ejemplo: UNIX (arquitectura con la que nos quedamos).

[1] Abreviaremos así *sistema operativo*.

La arquitectura UNIX, que estudiaremos, presenta las siguientes propiedades:

1. El SO es un programa que no acaba. Puede llamarse a cualquiera de sus funciones, aunque no a todas: a un grupo de ellas denominadas **llamadas al sistema**.

2. El kernel puede ser llamado por varios procesos a la vez. Se utilizan semáforos o bien modo exclusivo para regularlo.

El código del SO también está dividido en ficheros y procesos. El código se organiza desde el nivel hardware hasta el nivel del usuario:

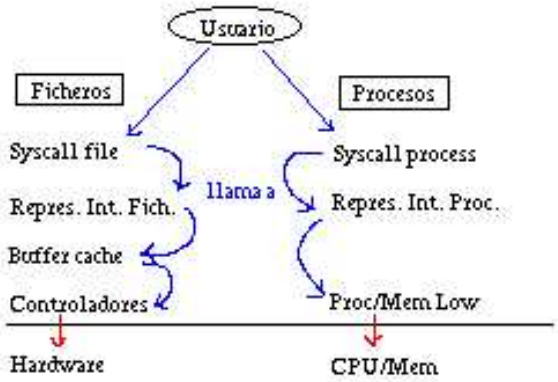

Figura 1.1: Organización de la interfaz usuario-hardware (sistema operativo).

Las zonas más cercanas al hardware llaman a controladores, que sí son ya, obviamente, dependientes de aquél. El resto del kernel es independiente.

Capítulo 2

El buffer caché

El **buffer caché** es una estructura software del sistema operativo formada por una serie de buffers organizados en colas hash (para agilizar su acceso) y enlazados en una lista de buffers libres o *free list* en la que se mantiene un orden de reemplazo LRU.

Todo buffer está siempre en una cola hash, y puede o no estar en la *free list*.

2.1. Tarea

La tarea del **buffer caché** es bastante concreta. Su presencia se debe a que el disco es mucho más lento que la memoria; para paliar esto, lo que hace es guardar los datos que ha leído del disco en un buffer, cuya velocidad de acceso es mucho mayor.

Estudiaremos las funciones del kernel y las llamadas al sistema que implementan las operaciones necesarias (ver figura 2.1).

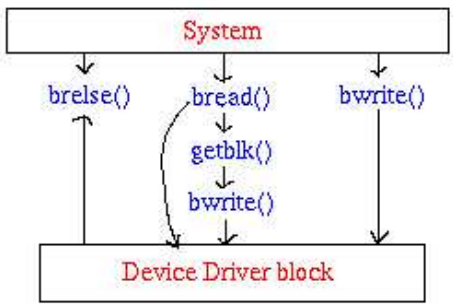

Figura 2.1: Funciones del buffer caché.

2.1.1. Algoritmo `bread()`

La función `bread()` es la función a la que llama el sistema como consecuencia de cualquier operación/instrucción en cualquier aplicación o programa que produzca como resultado la demanda de una lectura de datos. Su esquema es el que se expone en la tabla 2.1.1, página 10.

```
entrada: identificación de un bloque
salida: buffer ocupado con los datos del bloque
{
  obtener buffer -getblk()-;
  if (datos validos)
    return (buffer);
  iniciar lectura en disco;
  sleep (hasta que termine la lectura);
  marcar datos como válidos;
  return (buffer);
}
```

Cuadro 2.1: Algoritmo `bread()`.

2.1.2. Algoritmo `breada()`

El algoritmo `breada()` (tabla 2.1.2, página 11), también llamado **de lectura anticipada** (`bread ahead`) es una ampliación al algoritmo `bread()` que además de leer el bloque solicitado, ordena la lectura asíncrona de otro bloque más, normalmente, el bloque siguiente. La razón de esto es la gran probabilidad que existe de que un proceso que necesita leer un bloque necesite más tarde leer el que le sigue[1] *–localidad de datos, secuencialidad de programas–*.

2.1.3. Algoritmo `getblk()`

La función `getblk()`, que hemos visto referenciada en el código anterior, se describe en la tabla 2.1.3, página 12.

La existencia del flag `ocupado` es necesario, independientemente de que tengamos la lista de libres, porque aunque un bloque estará en la lista de libres si y sólo si no está ocupado, ésta nos sirve para mantener un orden LRU a la hora de seleccionar un bloque para ser utilizado (para traer datos de un bloque en disco que no se encuentre en cache y cuya lectura haya sido demandada).

2.1.4. Algoritmo `bwrite()`

De acuerdo con la política de eficiencia que venimos estudiando, las escrituras a disco sólo se realizan cuando es estrictamente necesario. Esto evitará muchas veces que escribamos a disco bloques de buffers que aún van a ser modificados más veces, aunque también es la razón de por qué los apagados no ordenados de las máquinas pueden dañar los sistemas de ficheros.

[1] No nos referimos al bloque con siguiente número de bloque necesariamente, sino al siguiente bloque correlativamente, es decir, el siguiente bloque que le corresponde a un fichero del que se esté haciendo una lectura secuencial, por ejemplo.

```
entrada: identificacion bloque lectura inmediata
         identificacion bloque lectura asincrona
salida: buffer ocupado con datos bloque lectura inmediata
{
  if (primer bloque no en cache)
  {
    obtener buffer para primer bloque -getblk-;
    if (datos buffer no validos)
      iniciar lectura en disco;
  }
  if (segundo bloque no en cache)
  {
    obtener buffer para segundo bloque -getblk-;
    if (datos buffer validos)
      liberar buffer -brelse-;
    else
      iniciar lectura en disco;
  }
  if (primer bloque estaba en cache)
  {
    leer primer bloque -bread-;
    return (buffer);
  }
  sleep (hasta que primer buffer tenga datos validos);
  return (buffer); /* del primer bloque */
}
```

Cuadro 2.2: Algoritmo `breada()`.

```
entrada: identificación de un bloque
salida: buffer ocupado
{
  while (no se haya hecho)
  {
    if (bloque en cola hash)
    {
      if (bloque ocupado)
      {
        sleep (hasta que quede libre);
        continue;
      }
      marcar buffer ocupado;
      quitar buffer de la free list;
      return (buffer);
    }
    else
    {
      if (la free list está vacía)
      {
        sleep (hasta que haya algún buffer libre);
        continue;
      }
      marcar primer buffer ocupado;
      quitar primer buffer de la free list;
      if (buffer modificado)
      {
        comenzar escritura asíncrona a disco;
        continue;
      }
      quitar buffer de su cola hash;
      poner en nueva cola hash;
      marcar datos no validos;
      return (buffer);
    }
  }
}
```

Cuadro 2.3: Algoritmo `getblk()`.

Así pues, la gestión de la escritura de bloques a disco corre totalmente por cuenta del sistema operativo, de suerte que una operación de lectura en un fichero *nunca* va a provocar directamente una llamada a `bwrite()` (¡si lo hiciese, no se usaría el caché!); a quien se llama al final es a `bread()`, pues, al fin y al cabo, escribir es leer, modificar y escribir.

El `sleep` en el que la función `bwrite()` se queda esperando, puede implementarse de dos maneras:

1. Mediante **espera activa** o lo que se denomina **E/S programada**, que consiste en el chequeo mediante un bucle `while` de algún tipo de variable hasta que el disco termine de escribir, momento en que dicha variable cambiará su valor (lo hará el S.O. a instancias del dispositivo o bien el propio dispositivo). Esta solución data de 1955.

2. Mediante **espera con interrupción**, que permite que la CPU quede libre y no tenga que estar ejecutando continuamente el bucle de la opción anterior, pudiendo ejecutar otros procesos en caso de que los hubiese. En este caso, además, el algoritmo sleep se hace independiente del hardware, pues el sleep se va a hacer ahora en el buffer y no en el disco. Una **interrupción** llegará proveniente del disco cuando éste termine (lo hace cada vez que acaba de leer/escribir un sector), haciendo que la ejecución de la CPU salte a una posición de memoria determinada y se notifique el suceso de la forma que sea conveniente.

```
entrada: buffer
{
  iniciar escritura en disco;
  if (escritura síncrona)
  {
    sleep (hasta que la escritura se complete);
    liberar buffer -brelse()-;
  }
  else
    if (buffer se libró de ser reemplazado -marda d/w-)
      marcar buffer como 'viejo';
}
```

Cuadro 2.4: Algoritmo `bwrite()`.

El motivo de la existencia de la *marca de* viejo es para controlar que un bloque que no haya sido escogido en `getblk` porque debía ser escrito a disco (y lo ha sido), en lugar de ir a parar, tras terminar de ser escrito, al *final* de la lista de libres (recordemos que es una lista LRU), vaya a parar al *principio*, pues la realidad es que no fue usado. Si no lo

hiciésemos así, estaríamos dando más posibilidades de permanecer en caché a los bloques modificados, y es absurdo que sea así.

Véase el código de `brelse` (tabla 2.1.5, página 14) para comprobar cómo se mira el estado de la marca de *viejo*.

2.1.5. Algoritmo `brelse()`

Esta última función (tabla 2.1.5, página 14) tiene la misión de quitar la marca de *ocupado* a los bloques, una vez que han terminado de ser escritos a disco.

```
entrada: buffer ocupado
{
  despertar procesos esperando por buffer libre;
  if (buffer válido y no viejo)
    poner al final de la free list;
  else
    poner al principio de la free list;
  marcar buffer libre;
}
```

Cuadro 2.5: Algoritmo `brelse()`.

Cuando el sistema operativo pierde el control, por cualquier motivo, la situación que se deriva se conoce como `panic`. En un caso así, es más que probable que el sistema de ficheros se vuelva inconsistente. ¿Cómo se arregla este problema? De una manera sencilla: el SO lleva cuenta (como en las transiciones de una base de datos) de las operaciones que realiza con los ficheros (bloques), información que se almacena en el `superbloque`. Gracias a ella, puede recuperarse de faltas de corriente, apagados no ordenados, etc.

2.2. Peculiaridades de BSD frente a UNIX

BSD también tiene buffer caché, evidentemente, pero además tiene *memoria virtual* e introduce ciertas mejoras al propio buffer caché. Veremos cómo interacciona todo ello:

2.2.1. Mejoras que introduce BSD

Entre las características del buffer caché propio de BSD, encontramos:

- BSD introduce bloques (y por tanto, buffers) de *tamaño variable*, pero no tiene un array de cabeceras y un array de buffers, sino un array de cabeceras + espacio

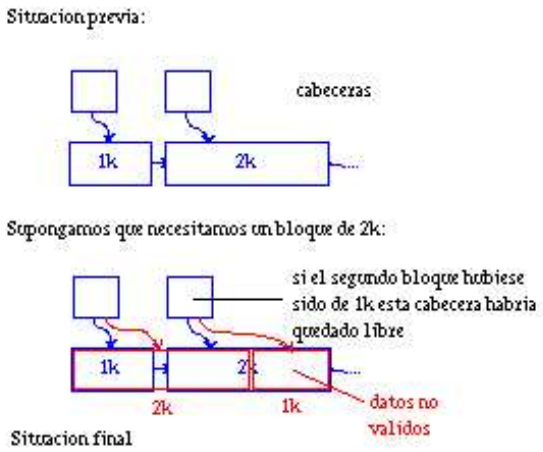

Figura 2.2: Buffer caché con tamaños de bloque variables (BSD).

de memoria, pues lo que se necesita con esta nueva organización es conocer dónde empieza el buffer y su tamaño.

Para ello se modifica `getblk` añadiendo a la estructura que representa los bloques un campo nuevo, `tambuf`, que almacene dicho tamaño de bloque. Además, al recorrer la lista de libres, no basta con simplemente escoger el primero, hay que comprobar que el tamaño sea suficiente.

El problema es que si procedemos de esta manera podemos no cumplir con la filosofía LRU, podemos elegir un bloque liberado recientemente si su tamaño se adecúa al que necesitamos y los que le preceden en la lista son todos demasiado pequeños. Lo que se hace en realidad es ir cogiendo buffers de la lista de libres en secuencia hasta que sumen suficiente espacio libre para satisfacer la petición actual.

Ello obliga al sistema a poseer algoritmos de *defragmentación de bloques*, para juntar los bloques[2] y un flag de `datos inválidos` para los "trozos que sobren".

- BSD maneja *varias listas de asignación* (en concreto, tiene cuatro listas con diferentes "tipos" de bloques libres: lista `LRU`, lista `AGE`, lista de `no validos` y lista de `EMPTY` –cabeceras–).

El hecho de que buffer caché tenga que convivir con un sistema de memoria virtual no es en realidad un gran problema. La principal característica de la memoria virtual es que una misma dirección virtual se puede traducir, en diferentes momentos del tiempo, por diferentes direcciones de memoria reales (físicas).

[2] Aunque se puede solucionar también con *memoria virtual*.

2.3. Ventajas e inconvenientes del uso de buffer caché

El uso de buffer caché reporta las siguientes consecuencias:

- Se tendrán *accesos más rápidos* sin mayor carga del sistema.
- Garantiza un control efectivo en el acceso al *device driver* al tenerlo centralizado en las funciones `bread` y `getblk`; buffer caché protege, pues, contra inconsistencias en los dispositivos porque garantiza acceso exclusivo.
- Evita problemas de alineamiento debido a que el DMA no tiene la capacidad de direccionamiento de la CPU[3].

[3]El DMA es un sistema que habilita la escritura entre la memoria y el disco sin la supervisión/intervención de la CPU. La DMA sólo escribe en el primer mega si se coloca en direcciones bajas; si se coloca en direcciones altas, escribe en todo el disco pero sólo en múltiplos de ciertas posiciones.

Capítulo 3

El sistema de ficheros

El sistema operativo, y en particular el kernel, puede considerarse integrado por varios *módulos* con diferentes funciones, de los cuales ya hemos visto uno de ellos (buffer caché), y que seguiremos viendo a lo largo del curso:

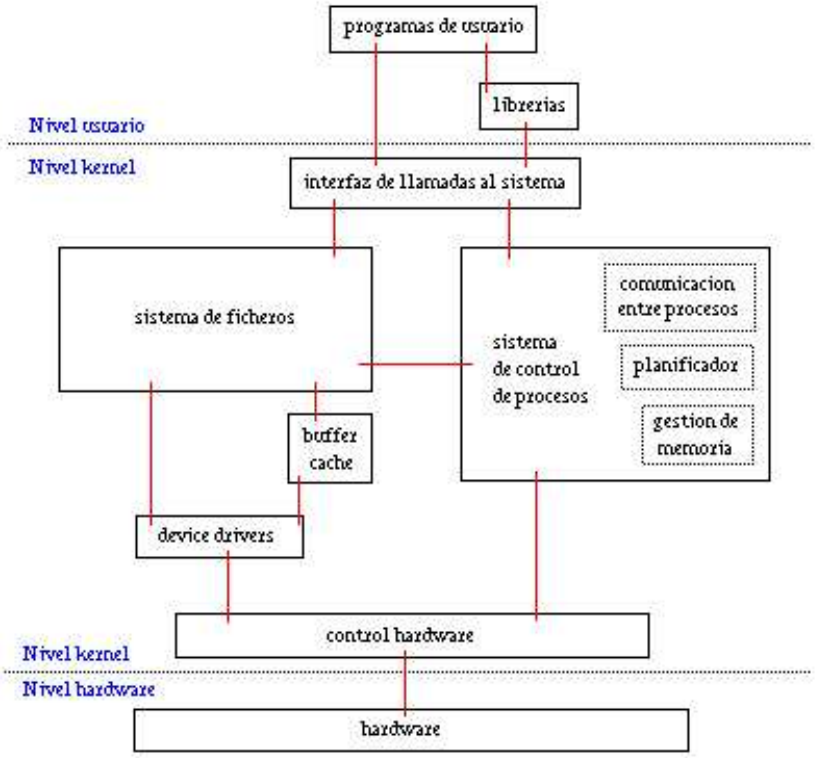

Figura 3.1: Diagrama de bloques del kernel de UNIX.

En concreto, en este tema nos centraremos en el **sistema de ficheros**, tanto en la representación interna de los ficheros como en el módulo de **inode caché**.

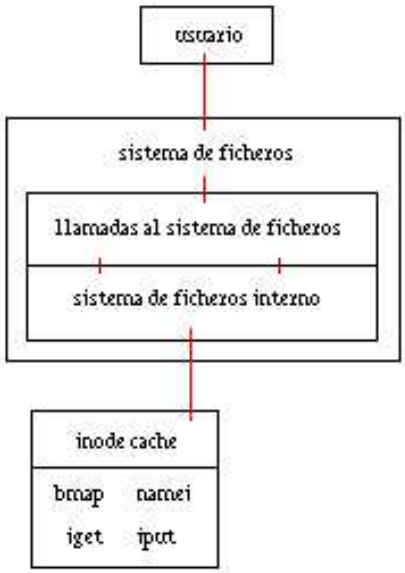

Figura 3.2: Diagrama de bloques del sistema de ficheros de UNIX.

3.1. Representación interna de los ficheros

El concepto de **fichero** se crea como una abstracción de los sistemas de almacenamiento, una entidad con finalidad integradora.

Un programador de alto nivel (ya no digamos un usuario), no necesita tener que saber cómo están almacenadas físicamente los datos en el disco para acceder a ellos. Con la aparición de los *ficheros*, sólo necesita conocer cómo funcionan, cómo se manejan éstos en el sistema operativo. Se desligan los términos ***organización*** y ***acceso***:

Organización En la mayoría de los sistemas, los *ficheros* son arrays de bytes[1] o bien arrays de registros (campos, estructuras).

Acceso Se realiza comúnmente por medio de una serie de funciones de nombre intuitivo:

- `open(fichero)`
- `read(bytes, fichero)`
- `write(bytes, fichero)`
- `close(fichero)`

Buffer caché, visto en el tema anterior, es el *nexo* entre el concepto abstracto de fichero y el device driver que accede al dispositivo físico. En referencia al contenido, ¿cómo determinaremos qué bloques pertenecen a un fichero y cuáles a otro? Es necesaria una

[1]Podemos considerar un *byte* como un *string* de 8 bits.

manera de identificar cuáles son los bloques que pertenecen a un fichero.

La primera solución que se plantea es la de mantener una *lista ordenada de bloques* para cada fichero, con los bloques en que está almacenado en disco (sus números, identificadores). Ahora bien ¿cómo la guardamos? Lo mejor es que la lista resida en el propio dispositivo, claro está, pues de lo contrario éste estaría ligado a una máquina concreta. De modo que el primer bloque del fichero contiene la lista de bloques de ese propio fichero.

Esta solución es mejor, además, que la de tener sólo un puntero al primer bloque y el número de bloques que componen el fichero, puesto que no obliga, como en este caso, a una contigüidad física de los bloques integrantes del fichero, porque establece un *mapping* que ayuda a romper dicha contigüidad.

También es mejor que tener una lista enlazada, esto es, tener en el bloque inicial un puntero al primer bloque y en cada bloque un puntero al siguiente, ya que aunque esta organización permite no contigüidad y no desperdicia espacio en guardar la lista, obliga a recorrer todo el fichero para alcanzar partes que estén próximas al final del mismo.

De modo que la solución óptima es la apuntada al principio: el primer bloque del fichero contiene una lista de punteros a bloques; si fuese necesario, el último puntero señalaría un bloque que a su vez contendría una lista con más punteros (continuación).

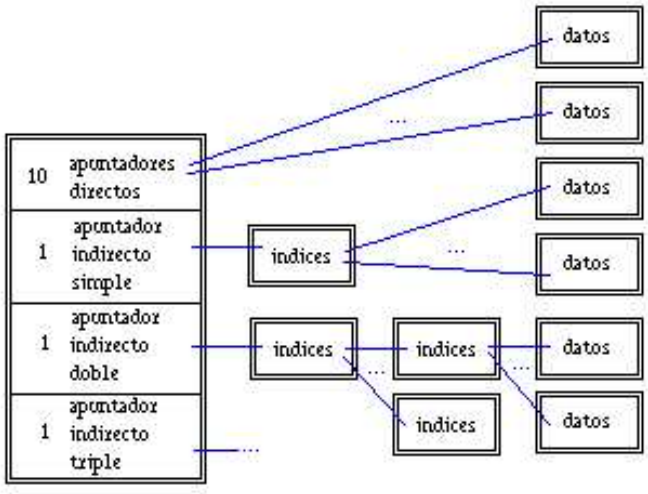

Figura 3.3: Lista de bloques de un fichero.

Con esta organización se reservan a priori todos los bloques del primer nivel; aún así, se pierde de media (al pasar del tamaño mínimo) sólo la mitad del último bloque en fragmentación interna debido a los índices.

Otros sistemas de ficheros emplean una filosofía distinta: reservan un espacio al principio del disco para indicar cada bloque a qué fichero pertenece (es lo que hace el sistema de ficheros **FAT**). Uno de los inconvenientes que tiene es que manejar toda la FAT siempre

supone majenar tantos índices como si el disco estuviera lleno; parece que es un desperdicio de espacio, también, pero no es del todo cierto porque, con el disco lleno, ambos sistemas emplean el mismo espacio en guardar la información de los bloques. No obstante, el almacenamiento distribuido siempre es beneficioso, porque no se accede siempre y continuamente a los mismos bloques como hace la FAT, algo que puede acabar dañándolos. El riesgo de perder toda la información es además mucho menor porque los bloques *informativos* no sólo están distribuidos, sino que lo están de acuerdo con un algoritmo, de modo que si se pierde alguno siempre se puede seguir sabiendo dónde está el siguiente, mientras que en FAT, en caso de fallo se asume que se pierde completamente todo. En UNIX-BSD, concretamente, se tiene un bloque de lista en cada pista, que además guarda información de su propia pista (mapa) de suerte que si resulta dañado ese bloque y/o esa pista, no afecta al resto del sistema[2].

Ahora bien, a la hora de modificar algo, la FAT es mucho más rápida, puesto que sólo tiene que cambiar un bit; en la lista, por el contrario, hay que modificar 2 punteros, lo que supone leer y escribir 3 bloques.

En cuanto a los bloques libres, se podría considerar que pertenecen todos a un mismo fichero y se tratalos de igual manera. No obstante, en UNIX no se hace así, sino que se tiene una lista enlazada (ver figura 3.1). En BSD, por su pate, mantiene un mapa de bits de todo el espacio con ceros y unos.

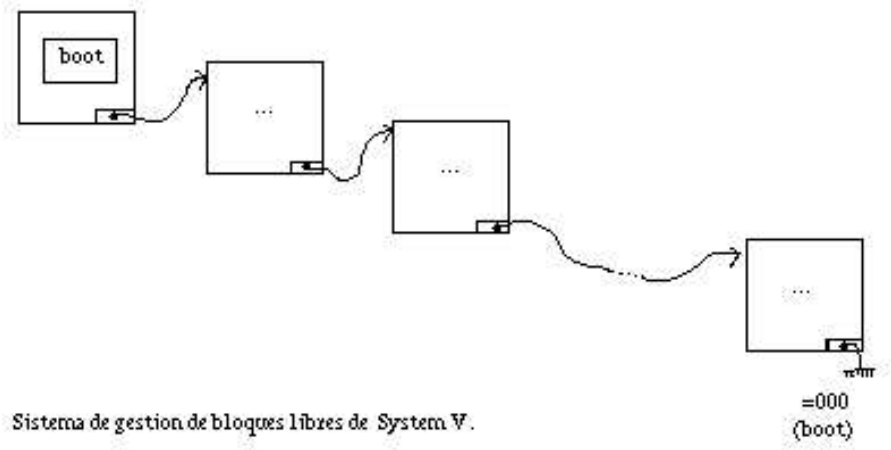

Figura 3.4: Lista de bloques libres en UNIX.

3.1.1. Tamaño de bloque: ¿grande o pequeño?

La cuestión del tamaño de bloque más adecuado no es trivial:

- Un **tamaño de bloque grande** ahorra tiempo en lecturas y escrituras al tratar más datos de una sola vez; ahora bien, esto no es cierto siempre, porque la situación

[2]BSD tiene otras características al respecto, como es que guarda una copia del `boot` en cada cilindro y mantiene una lista de bloques libres –un mapa de ceros y unos–.

empeora cuando dicho tamaño se hace mayor que el tamaño de una pista. El motivo es que al hardware (disco) le lleva el mismo tiempo leer un sector ¡que leer la pista entera![3] Por otra parte, cuanto más grande sea el tamaño de bloque, más espacio físico se va a "desperdiciar" en la gestión de los propios bloques y ficheros.

- Un **tamaño de bloque pequeño** perjudica si hay que leer mucho, ya que se pierde mucho más tiempo en acceso al dispositivo.

El sistema de bloques permite evitar *fragmentación externa*; en cuanto a la *fragmentación interna*, cuanto menor sea el tamaño de bloque, menor será. Claro que como hemos visto, es mejor un tamaño de bloque más bien grande, pues entre consumir espacio o tiempo de acceso, es una razón de mayor peso la segunda, de modo que se suele escoger un *tamaño de bloque grande*[4].

Algunos sistemas (entre ellos, UNIX BSD) han intentado solventar este dilema tamaño de bloque grande vs. tamaño de bloque pequeño permitiendo **dos tamaños de bloque**, uno grande y otro pequeño. De este modo se intenta aproximar la velocidad de acceso media a la que se tendría si el tamaño del bloque fuese único y grande y el desperdicio de espacio medio al que se produciría si el tamaño de bloque fuese único y pequeño.

Para manejar dos tamaños de bloque diferentes es necesario un *mecanismo de fragmentación-compactación dinámico*. Los bloques se numeran asignando los números de *fragmento*[5]. Cada vez que un fichero pide un fragmento, se le intentan asignar contiguos (de forma que si llega a tener todos los de un bloque, se reagrupen en uno solo). Para la gestión y contabilidad, se mantiene un mapa de bits de fragmentos/bloques.

3.1.2. Algoritmo `bmap()`

Este algoritmo (tabla 3.1.2, página 22), dado un *offset* en un fichero o directorio cuyo inodo se encuentre en la tabla de inodos de memoria (veremos qué es esto en la sección 3.2, página 22), devuelve el número de bloque en disco en el que se encuentra dicho *offset*.

3.1.3. Algoritmo `namei()`

El algoritmo `namei()` (tabla 3.1.3, página 23) identifica (asocia) números de bloque con nombres de directorios o ficheros. El sistema tiene una tabla denominada *directorio* que guarda las corresondencias.

Formalmente, lo que hace esta función es, a partir del *path* de un fichero (o directorio), colocar su inodo en la tabla de inodos en memoria (ver sección 3.2, página 22).

---

[3] Esto no es estrictamente cierto; en realidad, se tarda el mismo tiempo en leer un sector que en aproximadamente leer media pista.

[4] Aunque sin excesos: si el tamaño de bloque es demasiado grande, el 80 % del espacio se pierde en fragmentación interna.

[5] El *fragmento* es el tamaño de bloque pequeño; el tamaño de bloque grande es un múltiplo del tamaño de fragmento, pues cada bloque estará formado por n fragmentos.

```
entrada: inodo, offset
salida: numero de bloque del sistema, offset dentro del bloque,
        bytes e/s en bloque
{
  calcular bloque logico a partir de offset;
  calcular byte inicio e/s;
  calcular numero de bytes e/s;
  determinar indireccion;
  while (queden por resolver indirecciones)
  {
    calcular indice en inodo o b.i. a partir de b.l.;
    obtener numero bloque fisico (inodo o b.i.);
    liberar bloque lectura previa (si hay) -brelse-;
    if (no hay mas indirecciones)
      return (numero de bloque);
    leer bloque -bread-;
    ajustar bloque logico segun nivel de indireccion;
  }
}
```

Cuadro 3.1: Algoritmo `bmap()`.

El problema es que según crece el número de ficheros, el *directorio* se hace muy poco manejable. Intentar hacer varias listas (*directorios*) no soluciona el problema. Sí lo hace tratar el *directorio* como un fichero más: lo único que hay que hacer es guardar el primer directorio (raíz). Esta sencilla idea habilitó la posterior aparición del concepto de subdirectorio y los actuales árboles de directorios y ficheros tal y como los conocemos y estamos acostumbrados a utilizarlos.

3.2. Inode caché

El sistema operativo tiene, además de buffer caché, otra pequeña caché, denominada **inode caché** donde guarda una copia de los índices de los ficheros en memoria. Su funcionalidad es la de cualquier caché: pretende evitar accesos a dispositivo.

Ahora bien, la presencia de *inode caché* no evita el acceso a buffer caché, de hecho *inode caché* accede a buffer caché. ¿No es, entonces, una estructura redundante, repetitiva? ¿Por qué si ya tenemos la información en buffer caché hacemos otra copia más? Las razones son varias, pero las dos principales son las funciones `open` y `close`.

open: lleva el inode del fichero a inode caché y lo deja fijo allí

close: borra el inodo del fichero de inode caché

La gran diferencia, pues, con buffer caché es que en inode caché los *bloques* no son reemplazables, permanecen fijos desde que llegan hasta que se indica explícitamente (con

```
entrada: pathname
salida: inodo ocupado
{
  if (pathname empieza con /)
    inodo_trabajo=inodo_raiz -iget-;
  else
    inodo_trabajo=inodo_directorio_actual -iget-;
  while (hay mas nombres)
  {
    leer siguiente componente (trozo del nombre);
    verificar que inodo_trabajo es directorio y permiso ejecucion;
    if (inodo_trabajo es root y componente "..")
      continue;
    leer datos directorio (bmap, bread, brelse);
    if (componente esta en el directorio)
    {
      obtener inodo_componente -iget-;
      liberar inodo_trabajo -iput-;
      inodo_trabajo=inodo_componente;
    }
    else
      return (no inodo); /* error de path not found */
  }
  return (inodo_trabajo);
}
```

Cuadro 3.2: Algoritmo `namei()`

`close`) que ya no se van a usar más. ¿Por qué no, de todos modos, dejarlos fijos en buffer caché en lugar de crear otra caché dedicada? La razón principal es que el inodo de un fichero ocupa mucho menos que un bloque completo.

Además, al permanecer fijos desde que llegan hasta que indican que pueden ser reemplazados, deducimos que el tamaño de esta inode caché marca el límite (número) de ficheros abiertos que puede sostener el sistema.

Los algoritmos relacionados con inode caché son `iget` (ver subsección 3.2.1, página 24) e `iput` (ver subsección 3.2.2, página 24).

3.2.1. Algoritmo `iget()`

Este algoritmo (tabla 3.2.1, página 25) hace en inode caché la misma función que hacen en buffer caché `bread`+`getblk`.

En este algoritmo, vemos que el inode se queda bloqueado entre dos llamadas, teniendo el programador el control cuando vuelve a la lista de libres. Además, se devuelve un error en lugar de quedarse esperando cuando no hay más espacio libre porque que alguno deje su sitio es algo que depende del usuario y no del sistema. Por norma, el sistema no puede consentir depender de algo que haga el usuario, no se va a arriesgar a provocar un bloqueo por hacer una espera en ese lugar.

3.2.2. Algoritmo `iput()`

El algoritmo `iput()` (tabla 3.2.2, página 26) libera un lugar en la tabla de inodos como consecuencia de un `close`.

3.3. Asignación de espacio

3.3.1. Asignación de bloques: algoritmos `alloc()` y `free()`

La asignación de bloques de disco a un fichero cuando éste crece tiene lugar mediante la aplicación del algoritmo `alloc()` (tabla 3.3.1, página 27). Por su parte, la liberación de bloques pertenecientes a un fichero que ya no los necesita, sea por la razón que fuere, la realiza el algoritmo `free()` (tabla 3.3.1, página 28).

Recordemos que la organización del disco se trata como si fuese un gran fichero, esto es, se usa un bloque para guardar las direcciones (números) de los bloques libres del disco.

3.3.2. Asignación de inodos: algoritmos `ialloc()` e `ifree()`

Una parte de los bloques de disco, como sabemos, se reserva para inodes. Los algoritmos que asignan y liberan inodes son, respectivamente, `ialloc()` e `ifree()`. Puede verse su pseudocódigo en sendas tablas: tabla 3.3.2 (página 29) y tabla 3.3.2 (página 30).

```
entrada: identificacion i-nodo
salida: i-nodo en memoria ocupado
{
  while (no se haya hecho)
  {
    if (i-nodo en tabla i-nodos)
    {
      if (i-nodo ocupado)
      {
        sleep (hasta que quede libre);
        continue;
      }
      if (i-nodo en FREE-LIST)
        quitar de FREE-LIST;
      marcar i-nodo ocupado;
      incrementar contador referencia;
      return (i-nodo);
    }
    /* no esta en la tabla de i-nodos */
    if (FREE-LIST vacia)
      return (error);
    quitar i-nodo de FREE-LIST;
    marcar i-nodo ocupado;
    poner nuevo numero i-nodo y dispositivo;
    poner en nueva cola hash;
    leer i-nodo de disco -bread-;
    inicializar contador de referencia;
    return (i-nodo);
  }
}
```

Cuadro 3.3: Algoritmo `iget()`.

```
entrada: inodo en memoria
{
  marcar i-nodo como ocupado (lock);
  decrementar contador referencia;
  if (contador referencia==0)
  {
    if (numero de links==0)
    {
      desasignar bloques de disco del fichero -free-;
      poner tipo fichero a 0;
      liberar inodo -ifree-; /* desasignar */
    }
    if (fichero accedido || cambio i-nodo)
      actualizar i-nodo en disco -bwrite-;
    poner i-nodo en FREE-LIST;
  }
  marcar i-nodo como no ocupado (unlock);
}
```

Cuadro 3.4: Algoritmo `iput()`.

Como anécdota, comentaremos que en System V no existe lista de inodos libres (de hecho, ni siquiera existe lista de bloques libres). La razón es que los cuatro algoritmos que acabamos de mencionar (`alloc()`, `free()`, `ialloc()` e `ifree()`) son simples optimizaciones realmente, pues para obtener un inodo/bloque libre, como tanto unos como otros están en un buffer y poseen una marca de *libre*, basta con recorrerlos todos y obtener uno que no esté ocupado.

```
entrada: identificacion sistema ficheros
salida: buffer para nuevo bloque puesto a 0
{
  while (lista de bloques sin asignar ocupada)
    sleep (lista de bloques sin asignar disponible);
  obtener numero de bloque de la lista superbloque;
  if (quitado el ultimo)
  {
    marcar lista bloques sin asignar ocupada -lock-;
    leer bloque cuyo numero se acaba de obtener -bread-;
    copiar contenidos en lista de bloques sin asignar;
    soltar buffer del bloque -brelse-;
    marcar lista bloques sin asignar disponible -unlock-;
    despertar procesos en espera por lista;
  }
  obtener buffer para el bloque -getblk-;
  poner a cero el buffer;
  disminuir la cuenta de bloques sin asignar;
  marcar el superbloque modificado;
  return (buffer);
}
```

Cuadro 3.5: Algoritmo `alloc()`.

```
entrada: identificacion sistema ficheros
{
  while (superbloque ocupado)
    sleep(superbloque libre);
  if (lista bloques sin asignar no llena)
    poner numero de bloque en lista sin asignar;
  else
  {
    marcar lista bloques sin asignar ocupada -lock-;
    hacer del bloque actual un bloque de enlaces;
    escribir la lista del superbloque en el nuevo bloque;
    escribir el bloque a disco -bwrite-;
    colocar el numero del nuevo bloque
      en la lista del superbloque (unico miembro de la lista);
    marcar lista bloques sin asignar disponible -unlock-;
    despertar procesos en espera por lista;
  }
  aumentar la cuenta de bloques sin asignar;
  marcar el superbloque modificado;
}
```

Cuadro 3.6: Algoritmo `free()`.

```
entrada: sistema de ficheros
salida: inodo en la tabla de inodos en memoria, ocupado
{
  while (no se haya hecho)
  {
    if (lista inodos sin asignar ocupada)
    {
      sleep(lista inodos sin asignar disponible);
      continue;
    }
    if (lista inodos sin asignar vacia)
    {
      marcar lista inodos sin asignar ocupada -lock-;
      obtener numero inodo recordado;
      empezar busqueda inodos hasta lista llena
        o no mas inodos sin asignar;
      marcar lista inodos sin asignar disponible -unlock-;
      despertar procesos en espera lista inodos sin asignar;
      if (no hay inodos sin asignar en disco)
        return (no inodo);
      establecer valor inodo recordado;
    }
    obtener numero inodo de lista inodos sin asignar;
    obtener inodo -iget-;
    if (inodo no esta libre)
    {
      escribir inodo a disco;
      soltar inodo -iput-;
      continue;
    }
    inicializar inodo;
    escribir inodo a disco;
    disminuir contador inodos sin asignar;
    marcar superbloque modificado;
    return (inodo);
  }
```

Cuadro 3.7: Algoritmo `ialloc()`.

```
entrada: identificacion inodo
{
  incrementar contador inodos libres;
  if (lista inodos sin asignar ocupada)
    return();
  if (lista inodos sin asignar llena)
  {
    if (numero inodo < numero inodo recordado)
      inodo recordado = numero inodo;
  }
  else
    almacenar nuevo inodo en lista inodos sin asignar;
  return();
}
```

Cuadro 3.8: Algoritmo `ifree()`.

3.4. Llamadas al sistema de ficheros

Las **llamadas al sistema de ficheros** son la *parte alta* del sistema de ficheros, el punto de interacción con el usuario o el programador. Veremos algunas de las principales llamadas al sistema de ficheros.

3.4.1. Llamada `open()`

Como ya hemos visto, el SO mantiene una copia del inode correspondiente a cada fichero abierto en memoria, de modo que, antes de que éste se use *efectivamente* es necesario saber cuándo va a ser usado, esto es, el usuario o programador (proceso, al fin y al cabo) ha de indicar su *intención* de leer o escribir en un fichero.

Para ello surge la llamada `open()`, que devuelve un número. Dicho número recibe el nombre de **descriptor de fichero** y es suficiente para identificar el mismo, pues representa la *entrada correspondiente al fichero en la tabla de ficheros abiertos del proceso.*

Cuando se crea un proceso, el sistema automáticamente crea la **tabla de ficheros abiertos del proceso** (***userfile descriptor table***) y abre tres entradas en ella:

0 `stdin`, entrada estándar

1 `stdout`, salida estándar

2 `stderr`, salida de error estándar

Además de esta tabla y la tabla de inodos, el sistema mantiene aún otra estructura ("intermedia" entre ambas, de algún modo), la **tabla de ficheros abiertos del sistema**. La tabla de ficheros abiertos del proceso es en realidad una extracción de ésta. De este

modo se evita la redundancia que se produciría cuando dos procesos abriesen el mismo fichero y se mantiene a la vez registro de las variables particulares de cada uno.

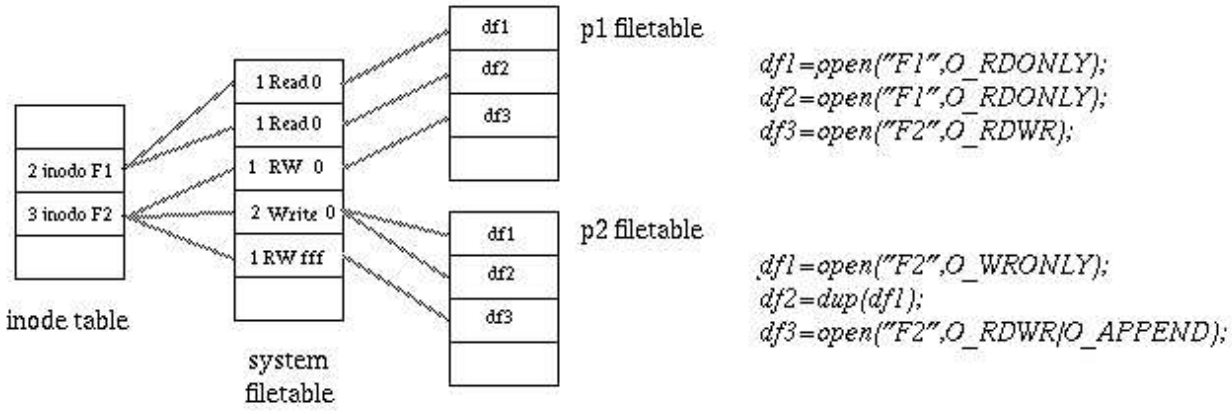

Figura 3.5: Estructura de tablas del sistema relativas a ficheros.

Además de los errores explícitos en este algoritmo, otros de los errores posibles pueden producirse en `namei` (si no existe la ruta), si no queda espacio en la tabla de ficheros del sistema (al intentar inicializar una entrada). Hay que hacer `unlock` el inodo (el `lock` está hecho en `namei`), que protegía el acceso concurrente; se marca el inodo como *no ocupado* (accesible), aunque sigue sin estar libre (reemplazable) –y lo seguirá hasta el `close`–.

3.4.2. Llamada `close()`

Esta llamada informa al SO que un determinado proceso no va a volver a usar un fichero concreto. El pseudocódigo de la llamada es, a grandes rasgos, la de la tabla 3.4.2 (página 32).

Como dato, la llamada `exit()`, que estudiaremos cuando veamos el tema de procesos, recorre la tabla de ficheros abiertos del proceso haciendo close de cada uno de ellos antes de

3.4.3. Llamada `read()`

Las llamadas `read` y `write` son muy similares. El pseudocódigo de `read` se presenta en la tabla 3.4.3 (página 33).

Nótese que los permisos se comprueban en cada acceso, pero no los del fichero, sino aquéllos con que fue abierto.

3.4.4. Llamada `write()`

Las modificaciones respecto al pseudocódigo de la llamada `read()` son sencillas, como puede verse en la tabla 3.4.4 (página 34).

```
entrada: nombre del fichero
         tipo de apertura
         permisos (si en modo crear)
salida: descriptor de fichero
{
  obtener inodo a partir de nombre -namei-;
  if (fichero no existe y modo crear)
    asignar inodo -ialloc-;
  if (fichero no existe o no permitido acceso)
    return (error);
  asignar entrada en tabla ficheros;
  inicializar entrada de tabla de ficheros
    {
      referencia inodo;
      contador de referencias;
      tipo de apertura;
      offset (segun tipo de apertura);
    }
  asignar entrada en tabla descriptores fichero usuario;
    establecer puntero a tabla ficheros;
  if (tipo de open supone truncar fichero)
    liberar bloques disco -ifree-;
  unlock inodo;
  return (descriptor de fichero);
}
```

Cuadro 3.9: Algoritmo de la llamada `open()`.

```
entrada: descriptor de fichero
{
  lock inodo;
  liberar puntero tabla usuario;
  if (no hay mas referencias a entrada tabla sistema)
  {
    liberar entrada tabla sistema;
    if (no hay mas referencias al inode)
      liberar inode -iput-;
  }
}
```

Cuadro 3.10: Algoritmo de la llamada `close()`.

```
entrada: descriptor de fichero;
         direccion transferencia datos;
         numero bytes a leer;
salida: numero bytes transferidos;
{
  obtener entrada T.F. a partir descriptor;
  comprobar accesibilidad en T.F.;
  establecer parametros en u-area
    (direccion, contador);
  obtener inodo a partir de T.F.;
  lock inodo;
  poner offset en u-area a partir del de T.F.;
  while (queden bytes por leer)
  {
    convertir offset a bloque -bmap-;
    calcular offset en bloque y bytes a leer;
    if (bytes a leer=0)
      break;
    leer bloque -bread-;
    transferir datos del buffer sistema
      a direccion usuario;
    actualizar campos u-area
      (offset, contador, direccion transferencia);
    liberar buffer -brelse-;
  }
  unlock inodo;
  actualizar offset en T.F.;
  return (numero bytes leidos);
}
```

Cuadro 3.11: Algoritmo de la llamada `read()`.

```
while (queden bytes por leer)
   {
     si hace falta otro bloque, asociarlo;
     ...
     transferir datos de direccion del usuario
       al buffer del sistema;
     marcar el bit del buffer modificado;
     ...
   }
```

Cuadro 3.12: Algoritmo de la llamada `write()`.

3.4.5. Llamada `lseek()`

Esta llamada es útil para posicionar el *offset* en cualquier lugar de un fichero, permitiendo de este modo acceso aleatorio tanto para lectura como para escritura. El pseudocódigo es tan sencillo como:

```
entrada: descriptor del fichero
{
  seguir el puntero de la tabla de ficheros del proceso
    hasta la T.F.;
  cambiar el campo offset en la T.F.;
}
```

Cuadro 3.13: Algoritmo de la llamada `lseek()`.

3.4.6. Llamada `cd()`

El algoritmo para el cambio de directorio también es muy sencillo, ya que en realidad no cambia nada salvo el valor de una variable (ver tabla 3.4.6, página 35).

3.4.7. Llamada `mknod()`

La llamada `mknod()` (`make inode`) crea un fichero sin contenido, es decir, crea una entrada en el directorio.

3.4.8. Llamada `link()`

Esta función crea un nuevo nombre para un fichero que ya existe.

```
entrada: nombre de directorio
{
  llamada a namei con el nombre de directorio
    -namei devuelve el inodo-;
  se comprueba que es un directorio;
  se guarda el numero de inodo en la variable
    numero de inodo del directorio actual;
```

Cuadro 3.14: Algoritmo de la llamada `cd()`.

```
entrada: nombre de fichero existente
         nombre de fichero nuevo
{
  obtener inodo nombre existente -namei-;
  if (demasiados links o link directorio sin ser root)
  {
    liberar inodo -iput-;
    return (error);
  }
  incrementar contador links inodo;
  actualizar copia disco;
  unlock inodo;
  obtener inodo dir padre -namei-; /* para nuevo nombre */
  if (ya existe nombre o esta en otro sst.fich.)
  {
    deshacer cambios anteriores;
    return (error);
  }
  crear nueva entrada de directorio en directorio padre;
  liberar inodo dir padre -iput-;
  liberar inodo fichero -iput-;
}
```

Cuadro 3.15: Algoritmo de la llamada `link()`.

3.4.9. Llamada `unlink()`

La tarea de `unlink` es tan simple como buscar la entrada asociada al nombre que se le pasa y borrarla, pero no sólo eso, sino que si es la última entrada que apuntaba a ese inodo, el último nombre de ese fichero, en ese caso además borrará el fichero (inode) –`iput`–. ¿Cómo lo sabe? En el inode se mantiene un campo que indica el número de referencias que están apuntando al mismo.

3.4.10. Llamada `mount()`

Tal y como hemos estudiado el tema hasta ahora, sólo podríamos tener *un sistema de ficheros* en el sistema (un disco, en términos de usuario). Para solucionar esto, surgen varias opciones:

- *Numerar los discos* de la siguiente manera:

  ```
  A:/.../.../...
  ```

  Habría que modificar también el algoritmo `namei` para tenerlo en cuenta. El sistema guardaría una tabla con las letras válidas y el dispositivo al que pertenecen. Es la solución de sistemas como MS-DOS.

- Permitir que *otros discos aparezcan como directorios del disco "inicial"*, haciendo que un determinado directorio del sistema de ficheros original pase a ser (se corresponda) con el raíz del otro. El sistema guarda en este caso una tabla con los nombres de estos directorios que sirven de "enlace" y los dispositivos con los que hacen de puente. También habría que modificar `namei` para que en el bucle en el que recorre los directorios compruebe si el que busca está en la mencionada tabla, y en caso afirmativo que pase a leer el directorio raíz del otro sistema de ficheros del otro disco. Por supuesto, esto permite hacerlo recursivo. Es la solución de UNIX.

El algoritmo que lleva a cabo la operación de asociar un directorio del sistema de ficheros actual con el directorio raíz de otro sistema de ficheros es `mount`.

3.4.11. Llamada `umount()`

Es el inverso al algoritmo `mount`: busca y borra de la tabla de montaje la entrada correspondiente, comprobando antes que no haya ningún proceso utilizando ningún fichero del sistema que se pretenda desmontar.

Como curiosidad, anotar que el directorio destino donde montamos no tiene por qué estar vacío; simplemente los ficheros que residan en él dejarán de ser accesibles (visibles, incluso, de hecho) hasta que se haga el `umount` del mismo.

```
entrada: nombre de dispositivo
         directorio de montaje
         opciones
{
  if (no superusuario)
    return (error);
  obtener inodo de dispositivo -namei-;
  comprobar que es correcto;
  obtener inodo de directorio en donde se monta -namei-;
  if (no es directorio o contador de referencias > 1)
  {
    liberar inodos -iput-;
    return (error);
  }
  encontrar entrada sin usar tabla de montaje;
  invocar rutina de apertura del manejador del dispositivo;
  obtener buffer del buffer cache;
  leer superbloque en ese buffer
    e inicializar campos necesarios;
  obtener inodo raiz del dispositivo montado -iget-
    y salvar en tabla de montaje;
  marcar inodo del directorio donde se monta: mount point;
  liberar inodo del dispositivo -iput-;
  unlock inodo del directorio punto de montaje;
}
```

Cuadro 3.16: Algoritmo de la llamada `mount()`.

3.5. Otras organizaciones de sistemas de ficheros

3.5.1. Sistemas de Ficheros de Bases de Datos

Este tipo de sistemas de ficheros (INFORMIX, DBII, ORACLE, IBM,...) tienen todos una misma estructura que puede ser:

1. ***Organización secuencial***, cuyas propiedades son:

   a) Acceso a la información lo más rápido posible.

   b) Contenido del fichero almacenado de forma continua físicamente.

   c) Etc.

   Los SO que manejan estos sistemas de ficheros son trivialmente un subconjunto de lo que ya hemos visto.

2. O bien ***organización mapeada*** (ya hemos visto algunas).

Los sistemas de gestión de bases de datos esbozan un nuevo tipo de fichero, distinto a todos los que hemos estudiado, que recibe el nombre de **ficheros de acceso por contenido**. Este nuevo tipo y sus características obligan a cambiar la política de implementación.

En una base de datos, lo normal no es acceder a la información por posición, sino a partir de una parte de ella, pretendiendo recuperar el resto. Ello implica modificar toda la definición y el concepto de fichero que hemos visto con anterioridad.

Lo que se hace es organizar el fichero en partes (**registros**) cada uno de los cuales se divide a su vez en trozos (**campos**). La implementación del sistema permite que, dado uno de esos *trozos* (**clave**), el SGDB[6] proporcione a cambio la otra. El tamaño de los *campos* puede ser fijo o variable, pueden tenerse varios campos clave, etc.

Las organizaciones que soportan esto:

- Leen secuencialmente los ficheros hasta encontrar la clave.
- Mantienen dos ficheros: uno de claves y otro de registros, cuyas posiciones se corresponden. La búsqueda es así más rápida (un fichero sólo de claves tiene una extensión menor) y el acceso al registro, una vez que se cuenta con la clave, es directo (gracias a dicha correspondencia). Para mejorar esto:
  - Se ordenan los ficheros de claves, de modo que la búsqueda que puede acometerse sobre ellos puede ser *binaria* (más rápida). El problema que ello presenta se percibe al intentar incluir una nueva clave (para solucionarlo, se propone como solución el punto siguiente).
  - Se mantienen índices de los índices (claves) en organizaciones con forma de árbol, de rápida carga en memoria y ágil manejo.
  - Otro tipo de técnicas varias: *hashing*,...

[6] Sistema Gestor de la Base de Datos.

Capítulo 4

Device Drivers

En el capítulo anterior hemos visto cómo se implementa el sistema de ficheros; éste, además de sus utilidades directas, provee a los programadores de un modo sencillo y simple para acceder a los dispositivos (salvo la CPU y la memoria), basado en el propio concepto de *fichero*.

Los **device drivers** son un conjunto de programas que están justo por encima del hardware y son llamados, por ejemplo, por el módulo de *buffer caché* (iniciar operación de lectura o escritura en un determinado dispositivo –disco–) y desde la *parte interna* del sistema de ficheros[1].

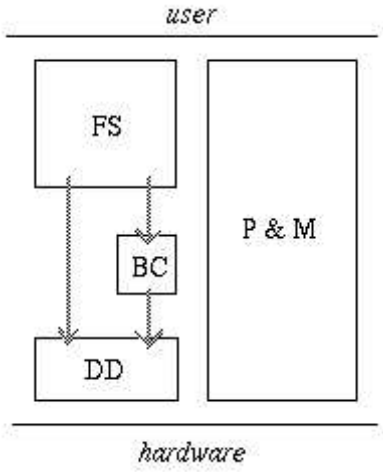

Figura 4.1: Lugar de los `device drivers` en la estructura del sistema operativo.

Las funciones de llamada a los **device drivers** son llamadas que ya no llaman a ningunas otras, actúan directamente sobre el hardware, completando de esta manera la estructura del sistema operativo.

El concepto de **device driver** en realidad no es necesario ni siquiera para los programadores (no necesitan conocerlo), sino sólo para el instalador del sistema operativo.

[1]En ocasiones, también pueden comunicar *avisos*, como por ejemplo la finalización de una operación de lectura o escritura en un dispositivo –disco–.

Antes de la existencia de los *device drivers*, la construcción de sistemas operativos era muy complicada, ya que, entre otras cosas, no se podía instalar el mismo sistema operativo en diferentes arquitecturas, se hablaba de *sistemas operativos de máquina*. Los grandes problemas que había que enfrentar eran, sobre todo:

- Que el hardware no es directamente accesible, no acepta las cosas como nos gustaría, "impone sus reglas".
- Que existen grandes diferencias entre el hardware de los distintos fabricantes, hay una gran variabilidad.

Es inmediato darse cuenta de que lo que se necesita son una serie de *funciones* o similar que conviertan la manera en que funciona un dispositivo (hardware) a la manera en que funciona el sistema operativo (software[2]), en concreto, buffer caché. Es más, no sólo debe ser posible para los dispositivos disponibles, sino para aquéllos que puedan aparecer en un futuro.

Sabemos que, llegado un punto, el hardware realmente *ejecuta funciones* (de carácter muy específico en su formato y a un nivel muy bajo, pero lo hace). Dichas *funciones* "son" el hardware, en cuanto son el lenguaje que éste acepta[3].

Al conectarse al ordenador, los dispositivos **se hacen aparecer en posiciones de memoria**, y el sistema guarda en una tabla la información que identifica una posición de memoria con el dispositivo mapeado en ella. Las operaciones utilizadas para el acceso a los dispositivos son entonces exactamente las mismas que se utilizan para el acceso a ficheros (en memoria). Este modelo, el más común actualmente, recibe el nombre de **E/S mapeada en memoria**. En otras arquitecturas se utiliza una filosofía diferente, en la cual existen una serie de operaciones especiales para el acceso a los dipositivos, y éstos se mapean al mismo rango de direcciones que memoria principal, de suerte que la CPU maneja dos tipos de direcciones (RAM y de E/S), debiendo conocer en todo momento a cuál de ambas se están refiriendo los datos que maneja. Esta otra perspectiva se denomina **E/S aislada**.

Cada device driver en realidad ha de asociarse con tres posiciones de memoria: en una se almacenará el sector a leer/escribir, en otra la posición de memoria del buffer origen/destino de la operación y en la última el código que identifique a dicha operación a realizar. Estas posiciones reciben el nombre de **puertos**. Realmente, esta cuestión es la que otorga su complejidad al problema de escribir un device driver para un dispositivo concreto: debido a la prácticamente nula estandarización del hardware, debemos conocer al detalle todas las características de este estilo (además de las referentes al cableado y conexión).

[2]Recordemos la máxima: un problema hardware siempre puede resolverse por software, pero el recíproco no es cierto.

[3]Identificaremos, pues, un dispositivo con las funciones que realiza.

Hoy en día hay sistemas que, al arrancar, realizan un chequeo de los dispositivos detectándolos mediante una serie de encuestas[4]. Además del puerto, para cada dispositivo debe indicarse la *interrupción* asociada y el DMA.

Todo lo visto no parece excesivamente complicado, ahora bien, **¿cómo se integra esto con el resto del kernel?** Precisamos más *funciones intermedias*, en este caso llamadas *funciones distribuidoras*. Cuando buffer caché llama a `iniciar_operacion_lectura` (o escritura), pasando un identificador de disco, número de buffer y número de bloque, la función que llama al device driver debe saber qué función del hardware asociar a dicha llamada según el identificador del dispositivo y el tipo de operación a realizar. Esto depende estrechamente de la configuración del sistema en cada momento, ¿cómo, entonces, se hace que esta función realice su tarea correctamente en cualquier caso? La primera solución histórica que se dio a este problema pasaba por recompilar la función en cuestión cada vez que el sistema sufría alguna modificación. Esto, por supuesto, no es soportable hoy en día.

Lo que hacen nuestros sistemas en la actualidad es mantener toda la información referente a este tema en una tabla, la **tabla de conmutación de los dispositivos**. En ella se tiene información para cada dispositivo de las cinco operaciones posibles sobre ellos: `read`, `write`, `open`, `close` y `ioctl`[5]. Dicha información puede residir en un fichero y ser cargada por el sistema al arrancar en esta estructura, de manera que en realidad la función intermedia entre buffer caché y el device driver no es necesaria, porque al tener este array con la información de los dispositivos (identificadores y funciones), buffer caché puede llamar directamente a la función que le interese del dispositivo requerido (en una sola línea y sin recompilar el kernel).

> **¿Qué es el *modelo de DD*?**
> Consiste en que todos los dispositivos son definidos de forma general mediante cinco operaciones y se mantiene la información referente a ellos en una tabla, la *tabla de conmutación*, funcionando todo ello sin necesidad alguna de recompilar el sistema.

No obstante, aunque de esta manera solucionamos cómo obtenemos las *referencias* a las funciones, no hemos explicado cómo se introduce el *código* del device driver en el kernel sin recompilarlo. Esta cuestión recibe el nombre de **integración dinámica**.

Como sabemos, todo código ejecutable de cualquier aplicación está dividido en una *sección de código*, que sólo se puede modificar al compilar, y una *sección de datos*, que puede, sin embargo, ser modificada en ejecución[6].

[4] Es lo que conocemos como *plug & play*.

[5] La única de las cinco que no es una función hardware sino una llamada al sistema no estándar; `ioctl` significa *input output control*.

[6] Las secciones de código y datos –como la de código dinámico y variables dinámicas– se separan por la misma razón: un motivo de seguridad, de protección del sistema y su integridad contra *malos* programadores.

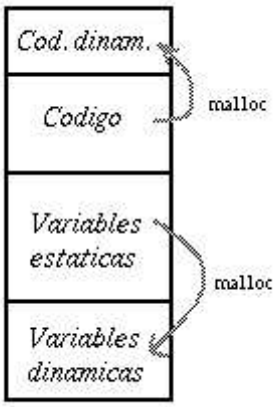

Figura 4.2: Diferenciación entre zonas de código y datos de un programa.

Podemos generar más variables para nuestro programa utilizando la llamada al sistema `malloc`, que reserva más memoria (*espacio de variables dinámico*) y manejarla mediante *punteros*. Existen un tipo especial de variables a las que, en lugar de direcciones de posiciones de memoria, se les pueden asignar direcciones de funciones. Utilizando este *mecanismo de llamada a través de variable* podremos entonces realizar lo que pretendíamos.

Ahora bien, no basta sólo con tener la referencia para poder llamar a la función adecuada, necesitamos incluir el código objeto como parte integrante del código dinámico del programa. Para ello es imprescindible que nos sea posible cargar un trozo de código de otro fichero a la mencionada zona de código dinámico (`load_code_function`). Por esta razón los device drivers son ficheros ejecutables (sin main) con un formato algo diferente: incluyen al principio del fichero una tabla que indica dónde empieza cada una de las cinco funciones que han de definir obligatoriamente. La función `load_code_function` lee ese fichero y lo copia a la parte de código del programa, donde las variables punteros a funciones ya pueden direccionarlos.

Obsérvese que esta técnica no sólo permite cargar el código al iniciar el sistema, sino también durante la ejecución normal del mismo[7].

4.1. Librerías dinámicas

Las **librerías dinámicas** permiten que el mismo código ejecutable/compilado funcione en arquitecturas diferentes y su filosofía está sacada de los DD, con una pequeña modificación: en el caso de los device drivers, cada uno de ellos sólo contiene cinco funciones de cabeceras predefinidas. En el caso de las funciones de librería, esto es totalmente diferente (cada librería puede contener un número amplio y variado de funciones de nombres y cabeceras adecuados a cada caso).

Lo que se hace es lo siguiente: además de la dirección donde empieza cada función,

[7]En Linux, una tarea similar la realizan los comandos `insmod` y `rmmod`.

en el *ejecutable* se mantiene el *nombre real* de las funciones, de modo que se las pueda llamar por nombre desde la *tabla de funciones sin resolver* que cada programa obtiene tras su compilación. Esta información (el emparejamiento de los strings de los nombres de función con la dirección *relativa* al fichero de la propia librería donde empieza cada una de ellas) se guarda en una tabla al principio del propio fichero ejecutable de la librería.

Existen **tres modos de resolver las llamadas a funciones de librería**:

- **Resolución en compilación**.-Programa y librería pueden compilarse por separado, pero existe una fase de *linkado* en la que se obliga a que se resuelvan todas las referencias. De este modo, el código ejecutable de la librería *queda incorporado* al del programa original[8].

- **Carga estática**.-El compilador hace una pasada por el programa haciendo *anotaciones* en las llamadas que no puede resolver (que serán, típicamente, las llamadas a funciones de librería). Al final del ejecutable, añade una *tabla de referencias no resueltas* donde guarda las anotaciones, identificadas con el *nombre* (string) de la función cuya llamada originó esa referencia. Al ejecutable del programa se le anexa el ejecutable de la librería, y es en el momento de la carga (`exec`) cuando se resuelven las referencias no resueltas presentes en la tabla.

  Ante esta situación, se presenta una mejora inmediata: ¿no se podría, acaso, conseguir que si dos programas utilizan la misma librería pueda haber sólo una copia del ejecutable de ésta en memoria (ya que, al fin y al cabo, es el mismo código)? Evidentemente, ello es perfectamente viable, y se consigue independizando los códigos ejecutables de programa y librería, algo que, dada la situación anterior, no es en absoluto complicado. El código de la librería estará entonces *compartido en memoria*. El comportamiento es el siguiente: cuando el programa es cargado, si la librería no está presente, se trae a memoria y se resuelven las llamadas pendientes; si ya está cargada, simplemente se resuelven las llamadas. Esta variante se denomina **compilado dinámico**.

- **Carga dinámica** o **carga bajo demanda**.-Presenta una última mejora con respecto al esquema de *compilado dinámico*, consistente en que la carga de la librería se realiza *sólo en caso necesario*, es decir, no se hace invariablemente en el momento de la carga del resto del programa, sino sólo si realmente se produce una llamada a funciones de dicha librería. Ello es posible porque las *anotaciones* que hacía el compilador en *carga estática* son sustituidas en este caso por llamadas a una función `cargar()` que recibe como argumento la referencia no resuelta. Será esta función, si llega a ejecutarse, la que desencadene el proceso de carga de la librería pertinente y resuelva la llamada[9].

[8] Es el sistema que suelen usar los usuarios cuando implementan sus propias librerías.

[9] Evidentemente, para poder hacer esto, el sistema guarda tablas con información acerca de librerías que están presentes en memoria, etc.

4.2. Device drivers para el usuario

Según el esquema visto hasta ahora, el usuario sólo puede llamar a las funciones de la *parte superior* del sistema de ficheros, no a las funciones de *buffer caché* y menos aún, por tanto, a las funciones de los *device drivers* (`open`, `close`, `read`, `write` y `ioctl`). Sin embargo, hay programas que requieren acceso directo a los dispositivos, aunque ello implique "saltarse" el kernel, el sistema de ficheros y toda la política de seguridad e integridad del sistema operativo. Son ejemplos, los sistemas de gestión de bases de datos (SGDB), que no usan el sitema de fichero del SO, sino que administran el suyo propio[10], programas de impresión que necesitan seleccionar parámetros como el tamaño del papel[11],...

Es por ello que el SO implementa *llamadas al sistema* que dan acceso a las funciones de los device drivers. En realidad no son llamadas al sistema especiales ni con nombres especiales, sino las mismas llamadas `open`, `read`, `write` y `close` que ya hemos visto, pero ligeramente modificadas. La modificación que se les incluye consiste en un `if` que comprueba que el fichero sobre el que se pretende realizar la operación pertenece al sistema de ficheros del SO o, por el contrario, es un dispositivo.

```
funcion X (id_fichero, ...)
/* X puede ser read, write, open o close */
{
  if (id_fichero corresponde a un DD)
  {
    /* se consulta la tabla de conmutacion */
    X[id_fichero](buffer, ...);
  }
  else
  {
    /* llamada al sistema de ficheros */
    X_FS(...);
  }
}
```

Cuadro 4.1: Modificación de las llamadas al sistema para admitir acceso a dispositivos.

El entero que devuelve la función `open` sigue siendo, aunque se ejecute sobre un dispositivo, un valor de descriptor que servirá para manejar el DD; en la tabla de ficheros abiertos del proceso la nueva entrada no apunta a una posición de la tabla de ficheros abiertos por el sistema, sino a la entrada correspondiente al dispositivo en la tabla de conmutación.

[10] Necesitan `read` y `write`.
[11] Estos requerirían la función `ioctl`.

Los nombres que se usan para referirse a los dispositivos, dependen del SO. Algunos establecen unas denominaciones determinadas (`LPT`, `COM`,...) que se asocian con su correspondiente DD en su entrada de la tabla de conmutación. En otros, como UNIX, el usuario puede usar el nombre que quiera, siempre que esté asociado a un tipo especial de fichero: el inode que tienen asociado –anotado en la entrada de directorio correspondiente– contiene un campo que indica que es un DD. Normalmente dichos nombres (ficheros) residen en el directorio `/dev`, pero pueden estar en cualquiera.

4.3. Device drivers especiales

A continuación veremos más en detalle algunos device drivers especiales, que suelen tener una serie de características añadidas.

4.3.1. Device drivers de disco

Los *device drivers de disco*, como todo device driver, implementan las funciones ya vistas (`open`, `close`, `read`, `write` e `ioctl`), que además de los cometidos comentados, realizan algunas tareas más: por ejemplo, en `open`, el device driver de disco debe encargarse de cosas como encender el motor del disco y opciones de este estilo (recíprocamente, `close` deberá encargarse de apagarlo). Lo que no es necesario en ese punto es, sin embargo, *abrir* la estructura de ficheros y directorios, no se necesita conocer posiciones actuales, no se va a trabajar con ficheros a ese nivel (eso se hará cuando, explícitamente, el programador llame en concreto, solicite, de alguna manera, un fichero).

Al margen de estas tareas adicionales necesarias por el tipo de dispositivo concreto (en este caso, los discos), estos device drivers suelen añadir otras utilidades, como por ejemplo, la **partición**, que consiste en la capacidad de que un disco físico se muestre al SO como dos discos o incluso más, de una manera totalmente transparente. Esto, que es útil desde el momento en que no es posible definir dos sistemas de ficheros diferentes en el mismo disco, se lleva a cabo mediante modificaciones en las funciones anteriores.

Uno de los cometidos de los device drivers de disco (en particular, de la función `read`) es proporcionar el ordenamiento de los sectores. El hardware no entiende de *sectores* ni *bloques*, sino que se mueve por tuplas *plato-cara-pista-sector*. Para abstraer al SO de todo esto, es el device driver el que se encarga de hacer el *mapeado* al número de sector de 0 a n. El hecho de que tenga encomendada esta tarea, hace muy sencilla la definición de particiones por parte del device driver, ya que éste mantiene ya de por sí una estructura que almacena números de pista y números de sectores por pista (entre otros parámetros e información[12]), con lo que un simple `if` soluciona la cuestión. Es de este modo como

[12]Estos datos corresponden al *formateado a bajo nivel*, y se suelen almacenar en el sector 0 de la pista 0, que siempre va a estar presente, de donde los lee el device driver. Se deduce de esto además, que en realidad el mapeado del número de sector tiene "truco", pues el device driver oculta el verdadero sector 0; esta ocultación, no obstante, resulta muy útil por ejemplo para la salvaguarda contra sectores defectuosos, o para utilización de técnicas de optimización de la velocidad de acceso como puede ser el *entrelazamiento*. El *formateado a alto nivel* se corresponde sólo con la definición de la estructura del disco

también un mismo device driver de disco puede manejar varios discos iguales, incluso reales, físicos, sólo con la ayuda de un parámetro más (el número de disco).

Optimizaciones como *raid*s o *mirror*s han surgido también de esta situación.

A mayores, hay discos que aportan otras funcionalidades, que sólo podrán ser manejadas si se tiene e instala el device driver adecuado, como puede ser la presencia de una caché interna hardware para el almacenamiento de los últimos sectores leídos o cosas más interesantes como el mantenimiento de *listas de acceso* y la aplicación de algoritmos de organización de las peticiones de lectura. También en este caso es el device driver quien tiene que gestionar esa política de planificación del servicio de lecturas del disco duro.

Por último, uno de las más recientes innovaciones en este terreno ha sido la aparición del *SCSI*, que intenta trasladar la idea de los device drivers al hardware, permitiendo la conexión de todo tipo de dispositivos, regida por señales.

4.3.2. Device drivers de terminal

Los **device drivers de terminal** son los device drivers de los dispositivos a través de los que los usuarios interaccionan con el sistema (teclados, pantallas, etc.[13]), a través de los que el sistema, por tanto, va a recibir sus *órdenes*.

La gran mayoría de los dispositivos sólo funciona en base a órdenes, sólo hacen las cosas que se les ordenan, de modo que tiene que existir alguna manera de recibir dichas órdenes.

La característica diferenciadora de los device drivers de terminal es que la información que reciben, por tener el carácter especial mencionado, se va a interpretar de otra manera: no se va a transmitir limpiamente, si no que ciertos códigos y/o combinaciones de bits producirán respuestas y/o acciones en el sistema (RETURN, CTROL+C,...). Este "preprocesado" se hace mediante un módulo que se añade a las funciones `read` y `write` del device driver, módulo que se denomina **disciplina de línea**.

Este módulo maneja una tabla (que puede ser programable) donde, como decimos, ciertas secuencias de bits/códigos se asocian con una acción, una llamada a una función del sistema o similar. Además, mediante la función `ioctl` siempre se puede deshabilitar, de forma que todos los device drivers de terminal, además de realizar esta función de "traducción", pueden trabajar con un comportamiento básico, lo que se denomina en **modo raw** (frente al anteriormente comentado, que se denomina **modo canónico**).

La *orden básica* de cualquier device driver de terminal es la que encarna la filosofía: "*Y ahora haz todo lo que te acabo de decir*". Esta es la verdadera interpretación que recae sobre el RETURN, que hace al device driver salir de su función `read` y pasar la información al programa en ejecución.

–sistema de ficheros, boot, inodes...–.

[13] Aunque lo que va a decirse en esta sección sería perfectamente aplicable a cualquier dispositivo de comunicaciones.

La *disciplina de línea* aporta, por tanto, dos características diferenciadoras a los device drivers de terminal:

- Rompe la estructura de capas del sistema operativo (ya que se llama directamente a funciones suyas).
- Buffering, necesario para poder habilitar operaciones como el *borrado*, por ejemplo. Todo lo que el usuario teclea, se almacena en un buffer organizado en forma de lista (**clist**) de listas (**cblock**s). Cada cblock contiene un puntero al siguiente, y en su primera posición guarda el número de caracteres válidos que almacena (usualmente pueden ser hasta 15). Se permiten operaciones entre cblocks[14], como dividir uno en dos, reconstruir uno a partir de dos...

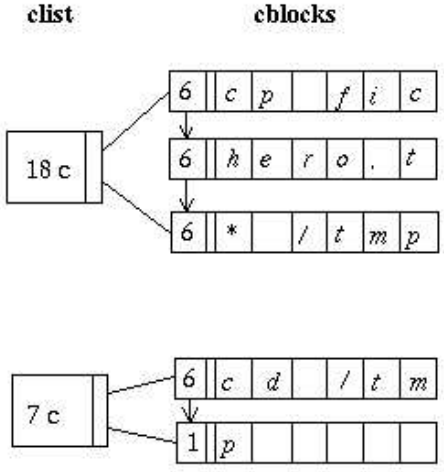

Figura 4.3: Estructura de *clist* y *cblocks*.

La lectura de dispositivos como el teclado tiene algo que la hace diferente: su *asincronismo*. En ocasiones, se necesitan leer datos, pero no los hay disponibles, de modo que hay que esperar; otras veces, sin embargo, ningún proceso necesita leer datos y llegan, de modo que hay que almacenarlos hasta que sean requeridos[15].

Cuando llegan datos, el device driver de terminal genera una *interrupción* para guardar los datos en la clist; se ejecuta entonces el algoritmo de atención a la interrupción del device driver[16], que no sólo guarda es información en la clist de entrada (la asociada al teclado), sino que también lo hace en la de salida (asociada a la pantalla), con el fin de que el usuario vea cuanto antes lo que escribe. Es, pues, en la salida donde se aplica la disciplina de línea (la que llama a `kill` cuando escribe CTROL+C en pantalla, por ejemplo).

[14]Todos estos mecanismos los implementa el device driver y pueden verse algunos de los algoritmos en el libro [Bac00].

[15]Esto es también aplicable a otros dispositivos de comunicaciones.

[16]La función `read` del device driver lee a quien lo demanda esos datos ya almacenados en la clist.

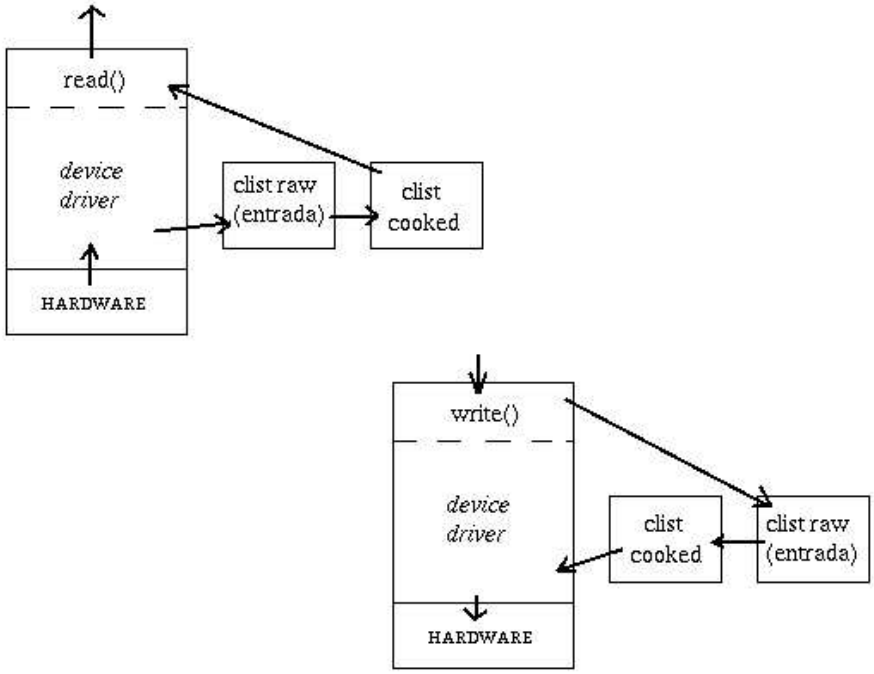

Figura 4.4: Funcionamiento de `read` y `write` sobre el device driver de terminal.

En realidad no hay una sola clist ni en la entrada ni en la salida, sino dos, siendo la segunda una clist de copia, sobre la que realmente se aplican las modificaciones que dicta la disciplina de línea y cuyos datos son los que se pasan al programa que los solicita. Cuando se trabaja en modo RAW, por el contrario, son los datos de la clist original los que se transmiten, así que se usa el buffering interno aunque no se haga tratamiento de la información por disciplina de línea.

En cuanto a la escritura, cuando llegan datos no se le pasan al device driver hasta que éste no está libre (puede estar atendiendo otra demanda), razón por la cual a veces las salidas por pantalla se demoran. Aquí, de nuevo, se copian literalmente en una clist y de esa clist a la clist de "backup", sobre la que se aplica de nuevo el "preprocesado" antes de enviarlos al hardware.

Terminal de control del proceso

Llamamos **terminal de control del proceso** a la *terminal activa*, que es la que recibe la información que es procesada por los device drivers de terminal. A cada proceso se le asigna una terminal de control, que coincide con aquélla desde la que se le invocó. Es el proceso de la terminal activa actual el que recibirá la señal que se haya definido cuando el usuario pulse, por ejemplo, CTROL+C[17].

[17]El device driver llama a la función `kill` del sistema operativo (que hace precisamente eso, enviar una señal a un proceso) cuando detecta esta secuencia.

Terminal indirecta y terminales virtuales

La forma de gestionar la terminales de control de los procesos es a través del concepto de **terminal virtual**, gracias al cual podemos entender cómo pueden funcionar los programas sin que importe desde qué terminal sean lanzados.

Lo que se hace es asignar siempre como terminal a un proceso `/dev/tty`, que es un dispositivo (un fichero en el directorio de dispositivos) que en realidad no se corresponde con uno físico concreto, sino que se refiere siempre al terminal de control activo (que será `/dev/tty1`, `/dev/tty2`,...) y se asocia a su valor en el momento en que se inicia un programa[18]. Así, se utiliza para escribir siempre en el terminal de control actual sin tener que saber explícitamente cuál es, y lo que se escriba en ella aparecerá en la terminal real del proceso que escribió (para ello el device driver de `/dev/tty` simplemente mira cuál es la terminal de control del proceso –cuyo valor se "anotó" al comenzar su ejecución– y *delega* la acción en su device driver real).

Otra forma de implementar terminales virtuales sin usar la terminal indirecta es mediante una simple línea añadida al algoritmo `write` del sistema de ficheros. Ambas formas son, en la práctica, indistinguibles, y la única manera de comprobar cuál de las dos está funcionando en un sistema operativo es mediante comandos del tipo `modstat` para comprobar si hay instalado un device driver para `/dev/tty` y el número que tiene asociado[19].

[18] Además, también se asigna y abre automáticamente el descriptor 0, como también lo hace con `stdout` (1) y `stderr` (2).

[19] Que servía, recordemos, para indexar la tabla de funciones de device drivers y que ésta no dependiese tanto del orden de carga al arrrancar el sistema.

Capítulo 5

Procesos

Haciendo una analogía con el concepto de *fichero*, que ya hemos estudiado, podríamos pensar que la operación de *crear* un proceso (`fork()`, `exec()`) ha de ser similar a la `open()` que realizamos sobre ficheros, y del mismo modo ocurrirá con la `close()` (`exit()`). Tendríamos que plantearnos, eso sí, a qué equivaldrían `read()` y `write()`.

Pero, al margen de esto, ¿qué es un **proceso**? ¿qué propiedades tienen? Igual que el resto de ficheros, tienen un *nombre* y un *contenido*. Evidentemente, en el primero no está la diferencia entre ambos, así que debemos buscarla en lo segundo. En el caso de los *procesos*, su contenido responde al conjunto de acciones que realiza, la especificación de lo que hace. Crear un proceso, por tanto, significa "abrirlo", implica que queremos que se haga lo que dice.

Ahora bien, esto ha de conllevar necesariamente que la lectura de un proceso no se realiza de igual modo a como se realiza la de un fichero normal. En efecto, un proceso no se lee, sino que se **interpreta**, y a esa interpretación es lo que denominamos **ejecución**.

Para ello, el sistema carga el contenido en memoria y *transfiere el control al primer byte* de esa zona, dando la orden a la CPU para que inicie la ejecución (interpretación). Esto es algo realmente importante, pues es el único momento en que el sistema operativo se *desentiende* del contenido de cierta parte de la memoria.

Todo lo demás referente a procesos es análogo al resto de ficheros.

El contenido de un proceso (de su fichero correspondiente) es una secuencia de bytes en un archivo normal. Distinguimos tres partes en él:

- **código** (*text*), la parte que se interpreta; con ella no se hace nada más, se lee y se ejecuta automáticamente[1]
- **datos** (*data*), espacio asociado al almacenamiento de variables
- **pila** (*stack*), parte que se necesita para poder llamar a *funciones*

[1]Su formato depende, pues, del sistema operativo –y no del compilador–. En realidad el compilador no sólo hace el ejecutable adecuado al SO, sino también a la CPU. Podría decirse que es un *device driver* para la CPU porque debe obtener un código máquina que ésta entienda.

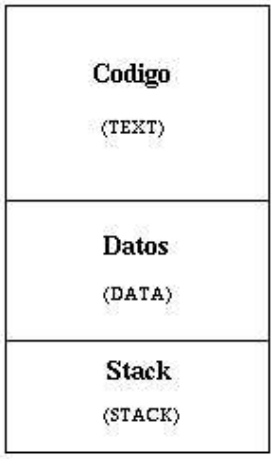

Figura 5.1: Partes en que se divide el contenido de un proceso.

De estas tres partes, la única que maneja el sistema operativo es la **pila**. Como hemos dicho, su función es habilitar la posibilidad de realizar llamadas a **funciones**, que no son más que trozos bien delimitados de código. A las funciones se les pueden pasar *argumentos*, que son en realidad variables cuya existencia está limitada al tiempo durante el que se ejecute la función correspondiente. Esas variables temporales han de almacenarse en algún sitio, y ese sitio es la pila, precisamente. Es más, no sólo se almacenan los argumentos, sino también las variables locales que se definan dentro de las propias funciones.

La estructura de la pila permite, además, llamadas sucesivas incluso desde las propias funciones, habilitando de esta manera la *recursividad*. El nivel de anidamiento de llamadas podemos saberlo en un momento dado por el nivel de profundidad de la pila, donde cada parte correspondiente a una llamada concreta a una función se denomina **frame**. En cada instante sólo se "está" en una frame, y tanto al acabar como al iniciar, la pila no contiene ningún frame, está vacía.

La única restricción con respecto a las frames es su tamaño: no pueden exceder del máximo direccionable por la CPU. Eso hace que no se puedan definir variables demasiado grandes como variables locales y que, en caso de ser necesario el pasar un argumento de dimensiones considerables a una función, se proporcione en su lugar un *puntero*.

Tras finalizar la ejecución de una subrutina invocada, debe retomarse el código en el punto siguiente a la instrucción que llamó a la función (`return`). Para poder hacerlo, en la pila se almacena además, antes incluso de empezar a ejecutar la función, la *dirección de vuelta/retorno* de la llamada.

A raíz de esto surge el mecanismo de **cambio de contexto**, que es muy similar a lo que se produce cuando se llama a una función, pero tiene lugar entre diferentes procesos: en la propia pila del proceso se guarda la información necesaria para volver a retomar su ejecución en el punto en que es interrumpido y ha de ceder la CPU. A esto se denomina

entonces **capa de contexto** de un proceso, y cada uno puede incluso tener varias.

La revolución que trajo UNIX al mundo de los sistemas operativos tiene que ver con esto: antes, el SO era un proceso especial, de suerte que llamar a una función del sistema suponía realizar una comunicación entre procesos, con toda la complejidad que conlleva. En UNIX, sin embargo, eso cambió y se pasó a considerar al SO como un proceso más, con una peculiaridad: su código se *añade* al del propio proceso que está en ejecución, de modo que la comprensión del mecanismo de llamadas a funciones del sistema se hace mucho más sencillo, simple y transparente[2].

5.1. Funciones básicas de procesos y memoria

Hay tres mecanismos básicos en lo que se refiere a procesos (y memoria), que ya hemos intuido en la introducción del tema, y que son:

- llamada al sistema: `syscall()`, para permitir la ejecución de funciones del kernel
- atención de una interrupción: `inthand()`, para ejecutar código de device driver
- cambio de contexto: `cntswt()`, para dejar de ejecutar un proceso y pasar a ejecutar otro

Pese a las diferencias que hay (el código de una llamada al sistema lo puede introducir el usuario, mientras que en los otros dos casos no,...), las tres funciones se basan en una misma idea, que veremos.

5.1.1. Mecanismo de llamada al sistema: `syscall()`

Hemos visto el mecanismo de llamada a funciones propias del usuario. `Syscall()` realiza las llamadas a funciones que no son del usuario sino del kernel. El problema más grande relacionado con esto es la localización del lugar donde residen las funciones del kernel que son llamadas por el programa del usuario.

La ubicación de las funciones definidas por el usuario se conoce gracias a la *tabla de símbolos* que el compilador construye al compilar el programa. Sería necesario, pues, contar con una tabla de función similar que permitiese saber dónde están las funciones del sistema.

El código del S.O. se encuentra siempre en memoria, tras cargarse al arrancar, y además todos los procesos, como hemos comentado, lo ven como una extensión de sí mismos y siempre en el mismo sitio, al principio de la memoria. Esto posibilita la existencia de una *tabla de símbolos del sistema*, que se genera también al iniciar el sistema y sigue la misma filosofía que la tabla de conmutación de los device drivers. Además, la inclusión o no de referencias de funciones del sistema en dicha tabla proporciona una manera

[2]Evidentemente, se mantienen las fronteras –con el objeto de distinguir y regular las zonas del kernel a las que el usuario no puede acceder–, por ejemplo, se marca en la pila el anidamiento a partir del cual un proceso hace llamada al sistema

sencilla y fácil de impedir que los usuarios llamen a funciones del kernel cuya ejecución no les está permitida (funciones como, por ejemplo, `bread()`,...). Una vez obtenida la referencia al lugar de comienzo de la rutina del kernel deseada, el funcionamiento de la llamada al sistema es exactamente igual que el de cualquier otra llamada a funciones del usuario.

En la compilación de llamadas al sistema hay un pequeño matiz: el compilador no sustituye la `syscall()` por una dirección absoluta obtenida de la tabla de símbolos del sistema, sino que deja indicado qué lugar de la tabla corresponde a la función invocada. De este modo, aunque el kernel sea recompilado y cambie, los programas pueden seguir funcionando de manera transparente.

En realidad, lo que acabamos de explicar no es exactamente así, aunque lo fue en un principio. El problema que subyace es que si, pese a no conocer explícitamente la dirección de una función del kernel, un código de usuario llamase a una dirección pertenenciente al rango del código del sistema, podría acceder igualmente, lo cual es muy peligroso (punteros fuera de control...). La solución a esta cuestión es *inhabilitar el resto de la memoria* no perteneciente a un proceso durante la ejecución del mismo, aunque con matices, puesto que si fuese estrictamente así, se conocerían las direcciones de las rutinas del sistema pero no se las podría invocar.

El matiz está en habilitar la memoria corresondiente al kernel sólo momentáneamente, para poder hacer el salto y pasar a ejecutar el código del sistema, e inmediatamente deshabilitar el espacio del usuario (y viceversa cuando se produzca la vuelta de la función). Así, en cada momento sólo la memoria correspondiente a un proceso, o bien al kernel, está accesible.

¿Cómo conseguimos implementar este "truco"? Es necesario que la CPU tenga la posibilidad de imponerse un *autolímite* de rangos de memoria por circuitería, es decir, un *módulo de manejo de memoria* (chip MMU). Entonces, simplemente hay que "jugar" con la habilitación/deshabilitación de la visibilidad de las direcciones de memoria del kernel. En caso de que, en un momento dado, una dirección de memoria enviada a la CPU se saliese del rango, se produciría una interrupción hardware (excepción por todos conocida: `segmentation fault`).

Parece todo resuelto. El sistema además guarda una tabla de procesos donde almacena los identificadores y los rangos de memoria asignados a cada proceso (entre otras cosas), cada proceso guarda sus propias características en un área especial denominada *u_area*. Pero `syscall()` es también una función del kernel[3], de modo que no hemos resuelto cómo alcanzarla. Se consigue gracias a un pequeño truco que introducen los compiladores, que añaden al código fuente dos líneas.

[3]Sus argumentos son el número correspondiente a la función del sistema que se invoca –obtenida de la tabla de funciones del sistema– y los argumentos que ésta necesite.

```
/* codigo del usuario */
...
habilitar kernel;               // <--- compilador
llamada_a_funcion_del_kernel(); //      syscall()
deshabilitar kernel;            // <--- compilador
...

/* codigo de syscall() */
syscall()
  {
    inhabilitar espacio usuario;
    ...                         //      funcion
    habilitar espacio usuario;
    return;
  }
```

En resumidas cuentas, parte del mecanismo de llamada al sistema es incluido por los compiladores en el código del usuario. De hecho, en la práctica es incluido en su totalidad, pues la habilitación e inhabilitación del espacio de usuario también se añade al código, junto con la demás información necesaria para realizar toda llamada a función:

```
/* codigo del usuario */
...
guardar direccion de vuelta en la pila;
guardar argumentos de la funcion en la pila;
habilitar kernel;
deshabilitar espacio usuario;
// saltar
...
// return
deshabilitar kernel;
habilitar espacio usuario;
...
```

Esto que puede parecer confuso es en realidad simple desde el momento en que se conoce que ambas acciones (habilitación kernel/deshabilitación usuario y deshabilitación kernel/habilitación usuario[4]) se corresponden con una única instrucción hardware: la CPU ha de incorporar una instrucción que permita el salto a una dirección de memoria de una zona de memoria B desde una zona de memoria A deshabilitando primero la zona de memoria A y saltando inmediatamente después. Esta instrucción se denomina **salto a subrutina cambiando el modo** (`trap`). Obviamente, se implementa también la vuelta

[4]En general, habilitación proceso_1/deshabilitación proceso_2 y deshabilitación proceso_1/habilitación proceso_2; estamos, pues ante la clave del mecanismo de cambio de contexto.

(`ireturn`, tipo especial de return que contendrá la función `syscall()`).

De este modo, además, sólo se permite al usuario acceder a "donde puede" (hacer llamadas permitidas), pues no salta a una dirección arbitraria, no indica a dónde quiere el `trap`, éste siempre se remite a `syscall()`, y en `syscall()` sólo se consulta la tabla de funciones del sistema. El `trap` consigue la dirección de `syscall()` de la tabla de **vectores de interrupción**, donde se guardan las direcciones de salto a funciones de atención a interrupciones hardware.

5.1.2. Mecanismo de atención de interrupciones: `inthand()`

El mecanismo de atención de interrupciones realmente es como el de una llamada al sistema, pero con el fin de ejecutar una función de un device driver. La diferencia está en que la llamada al sistema aparece explícita en el código del usuario, mientras que la interrupción no; cuando llega, no obstante, su comportamiento es como si lo hubiese estado.

Por tanto, basta con que la instrucción `trap` no sólo pueda ser ejecutada por el usuario (a instancias del compilador) sino también por una interrupción hardware, provocando el mismo efecto (aunque saltando a otra dirección).

Una de las utilidades que se ha dado en algunos sistemas a las interrupciones a raíz de la aparición del control del acceso a rangos de memoria por parte de la CPU ha sido la implementación de un régimen más estricto en el comportamiento de los procesos y el acceso a sus propios rangos de memoria. Para ello, se han separado las tres zonas que los componen (código –que además se establece de "sólo lectura"–, datos y pila). Se evitan de esta manera (mediante la aparición del famoso `segmentation fault`) accesos incontrolados a las zonas de código y pila, e incluso entre variables, proporcionando más fiabilidad y seguridad[5]. Lo ideal sería que incluso cada frame del stack estuviese en una región separada, pero la implementación de este particular ya resulta demasiado costosa en relación al beneficio reportado, razón por la cual los sistemas no la proporcionan.

Buffer overflow

Uno de los fallos de seguridad más frecuentes en sistemas y programas es el conocido como **buffer overflow**. Relacionado con lo que acabamos de ver, consiste en lo siguiente: en una variable local se lee directamente un string. Si no se comprueba la condición de finalización como la aparición de un cero, pueden introducirse en el string caracteres estratégicamente seleccionados para sobreescribir malévolamente el contenido del stack, sin alterar su estructura lógica, de forma que se cambien direcciones de retorno, por ejemplo.

[5]El caso de la pila es un tanto especial: se coloca al final de la memoria, lo más alejada de código y datos posible, y en caso de "salirse" por su extremo superior (siempre que sea por una llamada a una función y no por acceso de variables, claro está), no se da error, sino que se amplía (si es posible) dicha zona en un tamaño predeterminado. Así se implementa el crecimiento dinámico del stack.

5.1.3. Mecanismo de cambio de contexto: `cntswt()`

El último mecanismo que vamos a ver, poniendo así fin al capítulo y al temario de la asignatura, es el de cambio de contexto, que es, como ya hemos dejado entrever, muy parecido a los dos vistos. Se trata de interrumpir la ejecución de un proceso, pasar a ejecutar otra cosa (en concreto, otro proceso: `trap otro_proceso`) y más tarde volver a retomar la ejecución del proceso en el mismo punto en que se interrumpió, que puede ser cualquiera (filosofía apropiativa de Unix).

Así descrito, vemos que el mecanismo de cambio de contexto se parece al de interrupción hardware en que el `trap` que se ha de atender no está explícito en el código del usuario: como no puede ser de otro modo, pues, `cntswt()` es una llamada al sistema provocada por una interrupción hardware, la del reloj del sistema, cuya atención en lugar de la ejecución de una función de device driver provoca la llamada a la mencionada `cntswt()`. Esta función lo que hace es saltar del proceso al sistema operativo y de éste a otro proceso, de manera que se reutiliza el mecanismo de llamada al sistema (`trap`). El cambio de contexto es, como vemos, una "mezcla" de los dos mecanismos estudiados en las dos secciones previas.

Ahora bien, ¿dónde empezamos la ejecución en el otro proceso? La referencia debemos hallarla en su stack (de igual modo que haremos cuando volvamos al proceso interrumpido), en el stack del proceso al que se salta. De esta manera, todos los procesos que no están en ejecución (que están en espera) tienen en la cima de su pila una dirección de vuelta.

En realidad, hay más cosas, necesarias todas ellas para retomar correctamente una ejecución interrumpida, como son valores de los registros de la CPU, etc.[6] Esta información se denomina **capa de contexto**, para distinguirla de los frames.

El paso de la ejecución de un proceso a otro se hace pasando a través del SO porque sólo éste conoce dónde se encuentran los demás procesos ajenos al actual, sus pilas, etc. y tiene acceso a los valores en ellas almacenados.

```
cntswt ()
  {
    guardar informacion del proceso actual;
    decidir proceso a ejecutar;             // scheduler()
    recuperar informacion del proceso seleccionado;
    saltar al proceso seleccionado;
  }
```

Cuadro 5.1: Pseudocódigo de la función de cambio de contexto `cntswt()`.

[6] También en una interrupción hardware, aunque no en una llamada al sistema.

Lo que sucede cuando un proceso pasa a espera y no hay ningún otro listo para ser ejecutado es una espera en el planificador (`scheduler`).

Es particularmente importante para el correcto funcionamiento de esta función el orden de almacenamiento de la información referente al proceso que va a resultar reemplazado en ejecución. En primer lugar, ha de almacenarse en la pila el valor del contador del programa (PC), a continuación los valores de los registros de la CPU y se actualiza el valor del top de la misma. Por último, y ya justo antes de saltar, se modifica *directamente en la pila* el valor PC almacenado, sumándole el tamaño de una instrucción en la arquitectura sobre la esté montado el sistema[7].

[7]Todo lo que estamos viendo es extremadamente dependiente del hardware, y sólo puede ser implementado en ensamblador.

Apéndice A

Prácticas

A.1. Estructura de directorios de UNIX

La estructura de directorios de UNIX no es completamente estándar, pero sí lo es en su mayor parte. Veremos a continuación cómo se distribuye. Trabajaremos usualmente contra la máquina `umia` (IP 10.10.10.29), pudiendo saber la versión del sistema con la que trabajamos mediante el comando `uname -a`.

Los directorios que encontramos haciendo un listado en el raíz (`/`) son, usualmente:

Directorio	*Contenido*
`/bin`	Ejecutables, comandos del sistema operativo.
`/boot` o `/kernel`[a]	Código ejecutable del kernel que se carga en memoria al arrancar (`vmlinuz` o `genunix`, respectivamente), como el `command.com` de Windows y algunos ficheros de configuración (que hacen las mismas tareas que `msdos.sys`, `config.sys`, etc). Para que una partición sea de arranque, ha de contener obligatoriamente este directorio y estos ficheros en él.
`/dev`	Enlaces a ficheros que se corresponden con dispositivos (device drivers)[b].
`/etc`	Ficheros de datos del kernel, ficheros de configuración de aplicaciones.
`/home`	Directorios de los usuarios.
`/lib`	Librerías, fragmentos de código compilado (código objeto) enlazable (como las `dll` de Windows).
`/proc`	Un directorio por proceso que se está ejecutando.
`/sbin`	Otros comandos, en origen de ejecución reservada eminentemente al superusuario.

[a] Será `/boot` en sistemas Linux y `/kernel` en sistemas SunOS Sparc.
[b] Que se hallen normalmente aquí no quiere decir que no puedan estar en otros directorios.

Cuadro A.1: Directorios de UNIX y sus contenidos.

(Continuación)

`/tmp`	Ficheros temporales. Es un directorio que puede ser usado por cualquiera y que se borra cada vez que arranca el sistema.
`/usr`	Contiene a su vez directorios `/bin` y `/lib` con aplicaciones de usuario y sus librerías.
`/var`	Ficheros de datos de algunos programas del sistema (`/spool`, `/mail`, etc).

Cuadro A.2: Directorios de UNIX y sus contenidos (*continuación*).

A.1.1. Enunciado de la primera práctica

Implementar un programa que contenga un `fork()` *y en el que los procesos padre e hijo escriban ambos en un mismo fichero. Utilizar las funciones* `write`, `fwrite` *y* `fprintf` *(esta última con redirección a un fichero).*

Conclusiones

Ejecutado el programa, comprobamos que con `fprintf` y `printf` las salidas de padre e hijo en el fichero se mezclan menos. El motivo al que achacamos este comportamiento no es otro que el hecho de que estas dos funciones *no son llamadas al sistema*, mientras que `write` sí lo es. La llamada al sistema, al contrario que las otras, no utiliza buffers intermedios (proporcionados por el compilador y que no se vacían hasta que se llenan o se fuerza el volcado).

Usar llamadas al sistema nos previene de errores en el compilador, pero también es más lento.

A.1.2. Enunciado de la segunda práctica

Implementar un programa que haga acceso a disco **asíncrono**, *esto es, que haga una lectura y no se quede esperando a que ésta termine, sino que se devuelva el control al hilo principal de ejecución. Idear alguna manera que comprobar el estado de la lectura y notificar el final de ésta, una vez los datos estén listos.*

Conclusiones

Se encontraron dos maneras de codificar la propuesta realizada:

- Utilizar el flag `O_NDELAY` en la apertura (con `open`) del fichero, realizar la lectura con `read` normalmente y comprobar mediante la función `fcntl` el estado de la misma.

  Por alguna razón que desconocemos, este método no dio resultado.

- Abrir el fichero normalmente con `open` pero utilizar la función `aioread` para realizar la lectura asíncrona, comprobando el estado del campo `aio_return` de una variable de tipo `aio_result_t` para chequear la finalización de la misma (`!=AIO_INPROGRESS`).

  Este fue el método empleado, con éxito. Se pudo comprobar cómo, tras una primera lectura que produce espera (por tener que acudir a disco), la repetición de la misma resulta en una obtención inmediata de los datos, obviamente debida a la residencia de los mismos en buffer caché.

A.2. Threads

Los **threads** surgen para evitar la copia del espacio de contexto, inevitable cuando se utiliza `fork()`.

A.2.1. Creación

En `SOLARIS`, la llamada es de la forma:

```
thr_create( ... , void *(* start_routine)(void *) , ...);
```

A.2.2. Características

Los *threads* tienen una serie de características propias, como por ejemplo:

√ La prioridad es relativa entre ellos, puesto que pertenecen a un mismo proceso.

√ Se pueden sincronizar, mediante el uso de `lock()` y `unlock()`.

√ Se puede hacer que cada thread se ejecute en un procesador diferente –siempre que estemos en el entorno adecuado– (`set_concurrency()`).

√ Implementar threads es igual de problemático que implementar las funciones `sleep` o `wake_up` (saltos en el código entre funciones).

A.2.3. Enunciado de la tercera práctica

Implementar un **simulador** *de buffer caché con el que se pueda usar el "verdadero" código de las funciones del kernel* `bread` *y* `getblk`*, es decir, implementar un planificador de procesos que gestione el caché y las funciones* `sleep` *y* `wake_up` *para el control de procesos.*

Realizar una simulación creando al efecto varios procesos que pretendan leer diferentes bloques utilizando las llamadas mencionadas (cuyo código será proporcionado).

A.3. Curiosidades

A.3.1. Sistemas Operativos basados en disco ¿cómo funciona el arranque configurable?

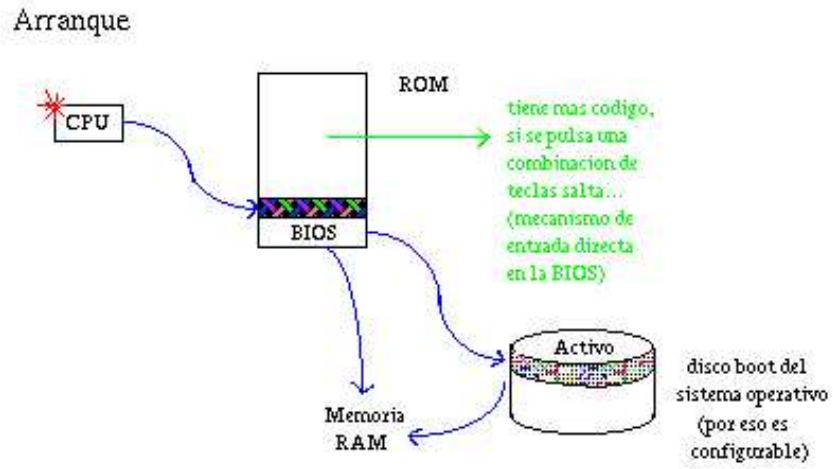

Figura A.1: Sistemas operativos basados en disco.

Índice alfabético

Índice de figuras

Índice de cuadros

Bibliografía

[Bac00] Bach, Maurice J.
The Design of the Unix Operating System.
Prentice Hall Software Series. Prentice Hall, 2000.

[Lefts] Leffler.
The Design and Implementation of the 4.3 BSD Operating System.
. . . , ...