3.5.1. El problema de inversión de prioridades

Un efecto colateral de que las estructuras del núcleo estén protegidas por mecanismos de sincronización es el problema de inversión de prioridades. Un caso simple de inversión de prioridades es el siguiente

- Un proceso A de baja prioridad hace una llamada al sistema y es interrumpido a la mitad de la llamada
- Un proceso B de prioridad tiempo real hace una llamada al sistema que requiere de la misma estructura que tiene bloqueada el proceso A

Un proceso de alta prioridad no puede avanzar hasta que uno de baja prioridad libere el recurso. Una solución a este problema se resuelve con herencia de prioridades: todos los procesos que estén bloqueando recursos requeridos por procesos de mayor prioridad son tratados como procesos de la prioridad más alta hasta que liberen el recurso.

3.6. Linux scheduler - CFS

Ver del apunte, está bastante resumido :).

4. Administración de memoria

El procesador puede utilizar directamente sus registros, su memoria caché y la memoria principal. Como todo programa debe ser cargado en memoria para ser utilizado, el sistema operativo tiene que administrarla de forma de que varios programas puedan compartirla.

Espacio de direccionamiento La memoria se estructura como un arreglo de bytes. Un procesador que soporta un espacio de direccionamiento de n bits puede referirse directamente a 2^n bytes. Sin embargo a través de un mecanismo llamado PAE (*physical address extension*) es posible referirse indirectamente a más.

Unidad de manejo de memoria - MMU En los sistemas multitarea se debe resolver cómo ubicar los programas en la memoria física disponible (principal y secundaria, ver 4.4) y esto no se puede hacer sin soporte de hardware (MMU). Por otro lado, la MMU también se encarga de verificar que un proceso pueda acceder o modificar datos de otro.

Memoria caché Memoria de alta velocidad situada entre la memoria principal y el procesador que guarda copia de las páginas (ver 4.3) a las que se acceden siguiendo los principios de localidad temporal (un recurso empleado recientemente, pronto se volverá a acceder) y localidad espacial (un recurso muy probablemente se accederá si recursos cercanos se accedieron).

Espacio en memoria de un proceso Un proceso se vuelca en memoria dividido en partes

- Sección de texto: imagen en memoria de las instrucciones a ejecutarse
- Sección de datos: espacio para variables globales y datos inicializados. Se fija en tiempo de compilación y no cambia
- Espacio de libres: espacio para asignación dinámica de memoria durante la ejecución del proceso (*Heap*)
- Pila de llamadas: espacio para la secuencia de funciones llamadas dentro del proceso e información asociada (*Stack*)

Resolución de direcciones En la compilación se sustituyen los nombres de funciones o variables por sus direcciones en memoria. En la copia en memoria de la sección de texto se deben resolver o traducir las direcciones a una forma que sea relativa al inicio del proceso en memoria.

- En tiempo de compilación: el texto del programa tiene la dirección absoluta de las variables y funciones
- En tiempo de carga: previo a la ejecución se actualizan las referencias de memoria dentro del texto para que apunten al lugar correcto
- En tiempo de ejecución: el programa hace referencia a una base y un desplazamiento lo cual permite que el proceso pueda ser reubicado en memoria mientras se ejecuta sin sufrir cambios. Requiere hardware específico

4.1. Asignación de memoria contigua

Partición de la memoria Se asigna a cada programa un bloque contiguo de memoria de tamaño fijo. El sistema operativo utiliza la región baja de la memoria y luego se puede asignar memoria a cada proceso con un registro base y uno límite. Desde el punto de vista desde el SO cada espacio asignado a un proceso es una partición.

4.1.1. Fragmentación

Cuando un proceso termina de ejecutarse y se libera su memoria se produce fragmentación: comienzan a aparecer regiones de memoria disponible interrumpidas por procesos activos.

Para asignarle memoria a los nuevos procesos y cubrir las áreas fragmentadas existen 3 estrategias:

- Primer ajuste: elegir el primer bloque en el que el proceso quepa. Es el mecanismo más simple y de más rápida ejecución pero no es óptimo.
- Mejor ajuste: elegir el espacio que mejor se ajuste al requerido. Los bloques que quedan después de esta elección están mejor ajustados (son lo más chico que se pueden hacer). Implica una revisión completa de todos los bloques lo cual es más costoso.
- Peor ajuste: elegir el bloque más grande disponible. Se busca que los bloques que queden después de esta elección sean tan grandes como sea posible. Usando un heap esta operación puede ser más rápida que mejor ajuste.

La **fragmentación externa** se produce cuando hay muchos bloques libres entre bloques asignados a procesos; la **fragmentación interna** se refiere a la cantidad de memoria dentro de un bloque que no se va a usar.

4.1.2. Compactación

El espacio total libre puede ser mucho más que lo que requiere un proceso pero al estar fragmentado no existe partición continua en la que quepa. Los procesos que emplean resolución de direcciones en tiempo de ejecución pueden lanzar una operación de compresión o compactación cuando noten un alto índice de fragmentación.

La compactación consiste en mover el contenido de los bloques para que ocupen espacios contiguos. Es una operación costosa ya que implica mover prácticamente la totalidad de la memoria y probablemente más de una vez cada bloque.

4.1.3. Intercambio con el almacenamiento secundario - *swap*

El sistema operativo puede comprometer más memoria de la físicamente disponible. Cuando la memoria se termina, el sistema suspende un proceso (usualmente uno bloqueado) y almacena una copia de su imagen en almacenamiento secundario que será luego restaurado.

Si un proceso que se quiere llevar al área de intercambio tiene operaciones de E/S pendientes, los resultados de éstas se pueden guardar en buffers (almacenamiento temporal) para luego trasladarlos al espacio correspondiente en el área de intercambio.

Esta técnica cayó en desuso por lo costoso de copiar un proceso completo en almacenamiento entero para luego restaurarlo en memoria principal. Ver 4.2.1 para una mejora

4.2. Segmentación

Los segmentos que conforman un programa se organizan en secciones (código compilado, tabla de símbolos, etc) y cuando el sistema operativo crea un proceso a partir del programa, carga algunas de estas secciones en memoria (como mínimo la sección de texto y variables globales). Para garantizar la protección de estas secciones el sistema puede asignar cada sección a segmentos diferentes los cuales pueden tener distintos juegos de permisos (texto: lectura y ejecución, datos libres y pila: lectura y escritura).

Otra ventaja de la segmentación es que incrementa la modularidad de un programa: las bibliotecas ligadas dinámicamente están representadas en segmentos independientes.

Un código compilado para procesadores que implementa segmentación contiene *direciones lógicas* que luego el sistema operativo podrá asociar a *direcciones físicas*. La traducción de direcciones lógicas a físicas puede fallar porque

- (*)El segmento no cuenta con los permisos: violación de seguridad
- (*)El tamano del segmento es menor al desplazamiento pedido: acceso de lectura arroja desplazamiento fuera de rango mientras que acceso de escritura arroja violación de segmento

- El segmento está marcado como no presente (ver 4.2.1 - segmentos de texto): segmento faltante
- (*)El segmento no existe: segmento inválido
- Una combinación de las anteriores. El sistema reacciona a la más severa

Las fallas indicadas con (*) suelen implicar la terminación de proceso.

4.2.1. Intercambio parcial

Permitir que solo ciertas regiones de un programa sean llevadas al area de intercambio. Si un programa tiene porciones de código que nunca se ejecutarán al mismo tiempo entonces tiene sentido separar su texto y datos en diferentes segmentos. En esta división se generarán segmentos que no se emplearán por un largo período de tiempo y que podrán ser llevados al swap por solicitud del proceso o por iniciativa del sistema operativo.

Segmentos de texto Una agregado que mejora el desempeno del intercambio parcial tiene que ver con copiar una única vez al disco segmentos de texto de solo lectura. Como no será modificado durante la ejecución, basta con marcarlo como no presente en las tablas de segmentos en memoria. Cualquier acceso (falla de segmento faltante) hará que se suspenda el proceso y que el SO traiga el segmento del área de intercambio a la memoria principal.

Bibliotecas dinámicas (código objeto independiente de su ubicación) Si la biblioteca reside en disco como una imágen de su representación en memoria, el sistema solamente tiene que saber su locación en el disco (no es necesario cargarla en memoria para luego llevarla al área de intercambio).

4.3. Paginación

Cada proceso está divido en bloques de tamano fijo (menor que el tamano de un segmento) llamados páginas. Los programas ya no requieren asignación de espacio contiguo de memoria (resuelve la paginación externa). La paginación requiere mayor soporte de hardware y aumenta la cantidad de información asociada a cada proceso (mapeo entre la ubicación real y lógica del programa).

La memoria física se divide en marcos (*frames*) del mismo tamano y el espacio para los procesos se divide en páginas que coinciden en tamano con los marcos. La MMU se encarga de hacer el mapeo entre marcos y páginas mediante tablas de páginas. Ahora cada dirección en memoria (potencias de 2) se puede dividir en un identificador de página (m bits más significativos) y un desplazamiento (n bits menos significativos).

Tamaño de la página Para evitar la fragmentación interna se puede emplear un tamano de página tan chico como sea posible, pero esto implica una carga administrativa alta:

- Las transferencias entre unidades de disco y memoria con más eficientes si pueden mantenerse recorridos continuos. El controlador de disco puede responder a solicitudes de acceso directo a memoria (DMA) si los fragmentos son continuos.
- El bloque de control de proceso (PCB) se agranda a medida que aumenta la cantidad de páginas de un proceso y esto incide en la rapidez con la que se puede hacer un cambio de contexto

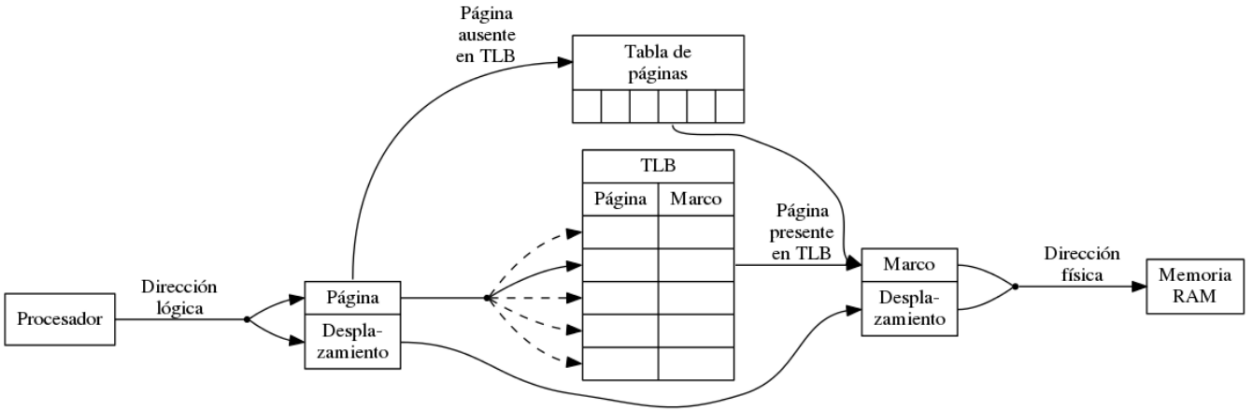
Por estas razones se trata de mantener el tamano de página tan grande como sea posible. El tamano habitual de una página suele ser 4 u 8 KB

4.3.1. Almacenamiento de la tabla de páginas

La tabla de páginas se guarda en memoria y se hace referencia a su inicio y longitud con registros especiales PTBR (*page table base register*) y PTLR (*page table length register*) respectivamente. En cada cambio de contexto, estos dos registros se modifican.

Con este mecanismo cada acceso a memoria se penaliza con un acceso adicional (el acceso a memoria se duplica): para traducir una dirección lógica a una física es necesario consultar la tabla de páginas en memoria.

Buffer de traducción adelantada (TLB - *translation lookaside buffer*) Tabla asociativa (*hash*) en memoria de alta velocidad dentro de la MMU que funciona como caché donde las claves son páginas y los valores son los marcos correspondientes en memoria física. Cuando el procesador solicita un acceso a memoria, si la traducción está en la TLB, la MMU tiene la dirección física inmediatamente, en otro caso, la MMU lanza un fallo de página (*page fault*) y se hace una consulta ordinaria en la memoria principal. Esta consulta luego se agrega a la TLB.



Como la TLB es limitada, se tiene que explicitar una política (ver 4.4.1) que elija que página/s víctima remover.

Subdividir la tabla de páginas El espacio empleado por las páginas es muy grande incluso empleando TLB. Aprovechando que la mayor parte del espacio de direccionamiento está típicamente vacío (stack y heap) se puede subdividir el identificador de página en 2 o más niveles (tablas externas y tablas intermedias). Este esquema funciona para computadoras con direccionamiento de hasta 32 bits: la paginación jerárquica implica que un fallo de página triplica el tiempo de acceso a memoria. Para un sistema de 64 bits, para que la tabla de páginas sea manejable se necesita septuplicar el tiempo de acceso.

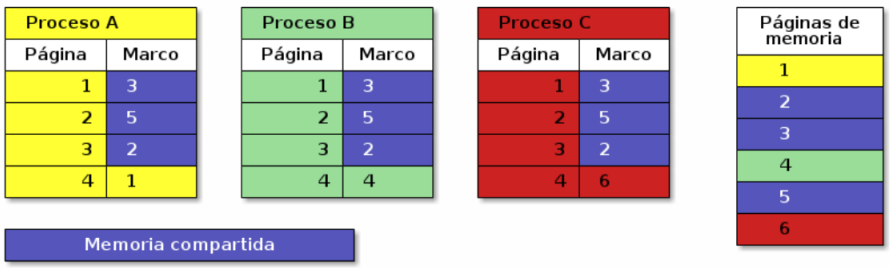
Otra alternativa es emplear funciones digestoras (*hash functions*) para mapear cada página a un espacio muestral mucho más chico (lista de direcciones físicas). Lo importante de estas funciones es que deben ser más ágiles para evitar que el tiempo que toma calcular la posición en la tabla no sea significativo frente a otras alternativas.

4.3.2. Memoria compartida

En la comunicación entre procesos (IPC) o cuando se ejecuta más de una vez un programa (sin código automodificable) no tiene sentido que las estructuras compartidas en el caso de los procesos o las páginas asociadas a cada instancia en el caso de muchas instancias de un mismo programa ocupen un marco independiente.

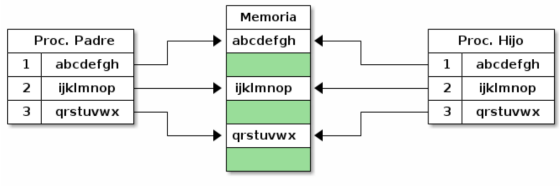
La idea también aplica a las bibliotecas de sistema (libc, openssl, zlib, libpng): las bilbiotecas suelen ser empleadas por una gran cantidad de programas.

En general, cualquier programa que esté desarrollado y compilado de forma de que todo su código sea de solo lectura permite que otros procesos entren en su espacio de memoria sin tener que sincronizarse con otros procesos que lo estén empleando.

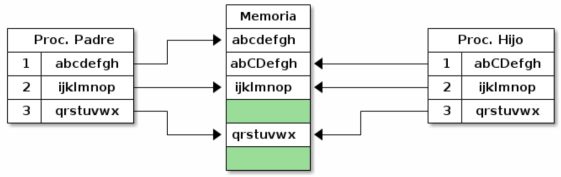


CoW - *Copy on write* En los sistemas Unix, cuando un proceso llama a fork() se crea un proceso hijo idéntico al padre con el cual se comparten todas las páginas en memoria. Para garantizar que la memoria no se modifique si no es por los canales explícitos de comunicación entre procesos, se usa el mecanismo copiar al escribir (CoW): mientras las páginas sean de solo lectura, las páginas de padre e hijo son las mismas pero ni bien cualquiera de los procesos modifica alguna de

las páginas, las modificaciones se copian a un nuevo marco y este marco deja de ser compartido. El sistema operativo, en vez de terminar el proceso debido al fallo de página que se produce al querer escribir en un bloque de solo lectura, copia la página en la que se encuentra la dirección de memoria que causó el fallo y marca esta nueva página como de lectura / escritura.



(a) Memoria luego del fork



(b) Memoria luego de la modificación de la primer página

4.4. Memoria virtual

En un sistema que emplea paginación, un proceso no conoce su dirección en memoria relativa a otros procesos, sino que trabajan con una idealización de la memoria, en la cual ocupan el espacio completo de direccionamiento, desde el cero hasta el límite lógico de la arquitectura, independientemente del tamaño físico de la memoria disponible.

Para ofrecer a los procesos mayor espacio en memoria del que se cuenta físicamente, el sistema emplea espacio en almacenamiento secundario (típicamente, disco duro), mediante un esquema de intercambio (*swap*) guardando y trayendo páginas enteras. La memoria es gestionada de forma **automática** y **transparente** por el sistema operativo (los procesos no saben de páginas). El encargado de esto es el *paginador*, no es intercambiador (*swapper*).

Paginación sobre demanda - *demand loading* Existe una gran cantidad de código durmiente o inalcanzable: código asociado a excepciones, a exportar archivos en ciertos formatos o a verificar que no haya tareas pendientes al cerrar un programa. Y un programa puede empezar a ejecutarse sin que esté completamente en memoria: basta cargar en memoria las instucciones que permiten continuar con la ejecución actual.

La paginación por demanda significa que al comienzo de la ejecución de un proceso se carga en memoria **solamente** la porción necesaria de código para comenzar y a lo largo de la ejecución solo se cargan en memoria las páginas que van a ser utilizadas (las páginas que no se necesiten no se cargarán).

La estructura empleada por la MMU para implementar el paginador flojo (*lazy*) es parecida a la que utiliza la TLB: la tabla de páginas incluye un bit de validez que indica si cada página del proceso está presente en memoria o no. Si el proceso trata de emplear un página marcada como no válida, se genera un fallo de página, el proceso se suspende, se trae a memoria la página solicitada y se actualiza el PCB del proceso y la TLB (si las páginas son válidas).

Un caso extremo de demand loading que se conoce como *pure demand loading* consiste en que todas las páginas que llegan a un proceso, llegan a través de un fallo de página. El proceso comienza sin ninguna página cargada y avanza con fallos de página sucesivos.

Una ventaja de demand loading es que al no requerir que se tengan en memoria todas las páginas de un proceso, permite que haya más procesos activos en el mismo espacio de memoria aumentando el grado de multiprogramación del equipo. Una desventaja de demand loading es que el tiempo efectivo de acceso a memoria se ve afectado. Este mecanismo no puede satisfacer las necesidades de procesos con requerimiento de tiempo.

Acomodo de las páginas en disco El acomodo de las páginas en disco no es óptimo. Además, si el espacio asignado a la memoria virtual es compartido con los archivos en disco, el rendimiento de la memoria virtual sufrirá. En los sistemas de la familia Windows se asigna espacio de almacenamientro en el espacio libre del sistema de archivos. Los de la familia Unix, en contraste, reservan una partición de disco exclusiva para paginación.

4.4.1. Reemplazo de páginas

Al sobre-comprometer memoria los procesos que están en ejecución eventualmente van a requerir cargar en memoria física más páginas de las que caben. Si todos los marcos están ocupados, el sistema deberá encontrar una página que pueda liberar (una página víctima) y llevarla al espacio de intercambio en el disco. Luego, se puede emplear el espacio recién liberado para traer de vuelta la página requerida, y continuar con la ejecución del proceso. Este mecanismo implica una doble transferencia.

Con apoyo de la MMU se puede incluir un bit de modificación o bit de página sucia (dirty bit) a la tabla de páginas. que se marca como apagado cuando se carga una página en memoria y se enciende cuando se escribe la página. Si el bit de página sucia de la página víctima elegida está apagado, la información es idéntica a la copia en memoria y se ahorra la

mitad de tiempo en transferencia. Este mecanismo disminuye la probabilidad de tener que realizar la doble transferencia del párrafo anterior.

Anomalía de Belady En general, a mayor número de marcos menor cantidad de fallos de página. Pero en algunas cadenas particulares (1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 - 3 y 4 marcos) se produce una degradación por aumentar la cantidad de marcos.

Algoritmos de reemplazo de páginas

- FIFO: se elige de víctima a la página que haya sido cargada hace más tiempo

Pagina	1	2	3	4	1	2	5	1	2	3	4	5
Marcos	1	1	1	1	1	1	5	5	5	5	4	4
		2	2	2	2	2	2	1	1	1	1	5
			3	3	3	3	3	3	2	2	2	2
				4	4	4	4	4	4	3	3	3
M/H	M	M	M	M	H	H	M	M	M	M	M	M

Tabla 2: Política FIFO en un sistema de 4 páginas físicas

Ventaja: fácil de implementar y comprender

Desventajas: 1) no tiene en cuenta el historial de solicitudes, todas las páginas tienen la misma probabilidad de ser reemplazadas, 2) es vulnerable a la anomalía de Belady

- OPT / MIN: se elige de víctima la página que no vaya a ser utilizada por el máximo tiempo. Este algoritmo es de interés teórico ya que no es posible predecir el orden en el que se van a requerir las páginas pero da una cota mínima para los otros algoritmos

Traza	1	2	3	4	1	2	5	1	2	3	4	5
P1	1	1	1	1	1	1	1	1	1	1	4	4
P2	-	2	2	2	2	2	2	2	2	2	2	2
P3	-	-	3	3	3	3	3	3	3	3	3	3
P4	-	-	-	4	4	4	5	5	5	5	5	5
M/H	M	M	M	M	H	H	M	H	H	H	M	H

Tabla 3: Política FIFO en un sistema de 4 páginas físicas

- LRU: se elige de víctima la página menos usada recientemente (la que no se usó hace más tiempo). El resultado de evaluar una cadena S empleando LRU es equivalente a emplear OPT sobre su inversa.

Traza	1	2	3	4	1	2	5	1	2	3	4	5
P1	1	1	1	1	1	1	1	1	1	1	1	5
P2	-	2	2	2	2	2	2	2	2	2	2	2
P3	-	-	3	3	3	3	5	5	5	5	4	4
P4	-	-	-	4	4	4	4	4	4	3	3	3
M/H	M	M	M	M	H	H	M	H	H	M	M	M

Tabla 4: Política FIFO en un sistema de 4 páginas físicas

Ventajas: 1) es una buena aproximación a OPT, 2) está libre de la anomalía de Belady

Desventajas: requiere apoyo de hardware (es mucho más complejo que FIFO[2])

[2]Tiene que recorrer todas las páginas si se usa un contador por página y encontrar la máxima; o actualizar reiteradas veces una lista doblemente enlazada para acceder a la víctima en tiempo constante

Capítulo 38

Concepto de archivo

En primer término, un archivo es un *tipo de datos abstracto* —esto es, podría verse como una estructura que exclusivamente permite la manipulación por medio de una interfaz *orientada a objetos*: los procesos en el sistema sólo pueden tener acceso a los archivos por medio de la interfaz ofrecida por el sistema operativo.[1] La siguiente sección describe las principales operaciones provistas por esta interfaz.

Para el usuario, los archivos son la *unidad lógica mínima* al hablar de almacenamiento: todo el almacenamiento *persistente* (que sobrevive en el tiempo, sea a reinicios del sistema, a pérdida de corriente o a otras circunstancias en el transcurso normal de ejecución) en el sistema al que tiene acceso, se efectúa dentro de archivos; el espacio libre en los diferentes dispositivos no tiene mayor presencia fuera de saber que está *potencialmente* disponible.

Dentro de cada *volumen* (cada medio de almacenamiento), los archivos disponibles conforman un *directorio*, y son típicamente identificados por un *nombre* o una *ruta*. Más adelante se presentarán de las diferentes construcciones semánticas que pueden conformar a los directorios.

[1] Como se verá en la sección 43.3.1, esto no es *necesariamente* así, sin embargo, el uso de los dispositivos *en crudo* es muy bajo. Este capítulo está enfocado exclusivamente al uso estructurado en sistemas de archivos.

38.1. Operaciones con archivos

Cada sistema operativo definirá la interfaz de archivos acorde con su semántica, pero en líneas generales, las operaciones que siempre estarán disponibles con un archivo son:

Borrar Elimina al archivo del directorio y, de ser procedente, libera el espacio del dispositivo

Abrir Solicita al sistema operativo verificar si el archivo existe o puede ser creado (dependiendo del modo requerido) y se cuenta con el acceso para el *modo de acceso* al archivo indicado y si el medio lo soporta (por ejemplo, a pesar de contar con todos los permisos necesarios, el sistema operativo no debe permitir abrir para escritura un archivo en un CD-ROM u otro medio de sólo lectura). En C, esto se hace con la función `fopen()`.

Al abrir un archivo, el sistema operativo asigna un *descriptor de archivo* que identifica la relación entre el proceso y el archivo en cuestión; éstos serán definidos propiamente en la sección 38.2.

Todas las operaciones descritas a continuación operan sobre el descriptor de archivo, no con su nombre o ruta.

Cerrar Indica al sistema que *el proceso en cuestión* terminó de trabajar con el archivo; el sistema entonces debe escribir los buffers a disco y eliminar la entrada que representa a esta combinación archivo-proceso de las tablas activas, invalidando al *descriptor de archivo*. En C, para cerrar un descriptor de archivo se usa `fclose()`.

Dado que todas las operaciones se realizan por medio del descriptor de archivo, si un proceso cierra un archivo y requiere seguir utilizándolo, tendrá que abrirlo de nuevo para obtener un nuevo descriptor.

Leer Si se solicita al sistema la lectura de un archivo hacia determinado buffer, éste copia el siguiente *pedazo* de información. Este *pedazo* podría ser una línea o un bloque de longitud definida, dependiendo del modo en que se solicite la lectura. El sistema mantiene un apuntador a la última posición leída, para poder *continuar* con la lectura de forma secuencial.

La función que implementa la lectura en C es `fread()`. Cabe mencionar que `fread()` entrega el *número de caracteres* especificado; para

trabajar con líneas de texto hace falta hacerlo mediante bibliotecas que implementen esta funcionalidad, como `readline`.

Escribir Teniendo un archivo abierto, guarda información en él. Puede ser que escriba desde su primer posición (*truncando* al archivo, esto es, borrando toda la información que pudiera ya tener), o *agregando* al archivo, esto es, iniciando con el apuntador de escritura al final del mismo. La función C para escribir a un descriptor de archivo es `fwrite()`.

Reposicionar Tanto la lectura como la escritura se hacen siguiendo un *apuntador*, que indica cuál fue la última posición del archivo a la que accesó el proceso actual. Al reposicionar el apuntador, se puede *saltar* a otro punto del archivo. La función que reposiciona el apuntador dentro de un descriptor de archivos es `fseek()`.

Hay varias otras operaciones comunes que pueden implementarse con llamadas compuestas a estas operaciones (por ejemplo, *copiar* un archivo puede verse como *crear* un archivo nuevo en modo de escritura, abrir en modo de lectura al archivo fuente, e ir *leyendo* de éste y *escribiendo* al nuevo hasta llegar al fin de archivo fuente).

Las operaciones aquí presentadas no son todas las operaciones existentes; dependiendo del sistema operativo, habrá algunas adicionales; estas se presentan como una base general común a los principales sistemas operativos.

Vale la pena mencionar que esta semántica para el manejo de archivos presenta a cada archivo como si fuera una *unidad de cinta*, dentro de la cual la cabeza lectora/escritora simulada puede avanzar o retroceder.

38.2. Tablas de archivos abiertos

Tanto el sistema operativo como cada uno de los procesos mantienen normalmente *tablas de archivos abiertos*. Éstas mantienen información acerca de todos los archivos actualmente abiertos, presentándolos hacia el proceso por medio de un *descriptor de archivo*; una vez que un archivo fue abierto, las operaciones que se realizan dentro de éste no son empleando su nombre, sino su descriptor de archivo.

En un sistema operativo multitareas, más de un proceso podría abrir el mismo archivo a la vez; lo que cada uno de ellos pueda hacer, y cómo esto

repercute en lo que vean los demás procesos, depende de la semántica que implemente el sistema; un ejemplo de las diferentes semánticas posibles es el descrito en la sección 38.3.

Ahora, ¿por qué estas tablas se mantienen tanto por el sistema operativo como por cada uno de los procesos, no lleva esto a una situación de información redundante?

La respuesta es que la información que cada uno debe manejar es distinta. El sistema operativo necesita:

Conteo de usuarios del archivo Requiere saberse cuántos procesos están empleando en todo momento a determinado archivo. Esto permite, por ejemplo, que cuando el usuario solicite *desmontar* una partición (puede ser para expulsar una unidad removible) o eliminar un archivo, el sistema debe poder determinar cuándo es momento de declarar la solicitud como *efectuada*. Si algún proceso tiene abierto un archivo, y particularmente si tiene cambios pendientes de guardar, el sistema debe hacer lo posible por evitar que el archivo *desaparezca* de su visión.

Modos de acceso Aunque un usuario tenga permisos de acceso a determinado recurso, el sistema puede determinar negarlo si llevaría a una inconsistencia. Por ejemplo, si dos procesos abren un mismo archivo en modo de escritura, es probable que los cambios que realice uno sobreescriban a los que haga el otro.

Ubicación en disco El sistema mantiene esta información para evitar que cada proceso tenga que consultar las tablas en disco para encontrar al archivo, o cada uno de sus fragmentos.

Información de bloqueo En caso de que los modos de acceso del archivo requieran protección mutua, puede hacerlo por medio de un bloqueo.

Por otro lado, el proceso necesita:

Descriptor de archivo Relación entre el nombre del archivo abierto y el identificador numérico que maneja internamente el proceso. Un archivo abierto por varios procesos tendrá descriptores de archivo distintos en cada uno de ellos.

Al detallar la implementación, el descriptor de archivo otorgado por el sistema a un proceso es simplemente un número entero, que podría entenderse como *el n-ésimo archivo empleado por el proceso*.[2]

Permisos Los modos válidos de acceso para un archivo. Esto no necesariamente es igual a los permisos que tiene el archivo en cuestión en disco, sino que el *subconjunto* de dichos permisos bajo los cuales está operando para este proceso en particular —si un archivo fue abierto en modo de sólo lectura, por ejemplo, este campo únicamente permitirá la lectura.

38.3. Acceso concurrente: bloqueo de archivos

Dado que los archivos pueden emplearse como mecanismo de comunicación entre procesos que no guarden relación entre sí, incluso a lo largo del tiempo, y para emplear un archivo basta indicar su nombre o ruta, los sistemas operativos multitarea implementan mecanismos de bloqueo para evitar que varios procesos intentando emplear de forma concurrente a un archivo se corrompan mutuamente.

Algunos sistemas operativos permiten establecer bloqueos sobre determinadas regiones de los archivos, aunque la semántica más común es operar sobre el archivo entero.

En general, la nomenclatura que se sigue para los bloqueos es:

Compartido (*Shared lock*) Podría verse como equivalente a un bloqueo (o *candado*) para realizar lectura — varios procesos pueden adquirir al mismo tiempo un bloqueo de lectura, e indica que todos los que posean dicho *candado* tienen la expectativa de que el archivo no sufrirá modificaciones.

Exclusivo (*Exclusive lock*) Un bloqueo o *candado* exclusivo puede ser adquirido por un sólo proceso, e indica que realizará operaciones que modifiquen al archivo (o, si la semántica del sistema operativo permite expresarlo, a la *porción* del archivo que indica).

[2] No sólo los archivos reciben descriptores de archivo. Por ejemplo, en todos los principales sistemas operativos, los descriptores 0, 1 y 2 están relacionados a *flujos de datos*: respectivamente, la entrada estándar (`STDIN`), la salida estándar (`STDOUT`) y el error estándar (`STDERR`); si el usuario no lo indica de otro modo, la terminal desde donde fue ejecutado el proceso.

Respecto al *mecanismo* de bloqueo, hay también dos tipos, dependiendo de qué tan explícito tiene que ser su manejo:

Mandatorio u obligatorio (*Mandatory locking*) Una vez que un proceso adquiere un candado obligatorio, el sistema operativo se encargará de imponer las restricciones corresponidentes de acceso a todos los demás procesos, independientemente de si éstos fueron programados para considerar la existencia de dicho bloqueo o no.

Consultivo o asesor (*Advisory locking*) Este tipo de bloqueos es manejado cooperativamente entre los procesos involucrados, y depende del programador de *cada uno* de los programas en cuestión el solicitar y respetar dicho bloqueo.

Haciendo un paralelo con los mecanismos presentados en el capítulo III, los mecanismos que emplean mutexes, semáforos o variables de condición serían *consultivos*, y únicamente los que emplean monitores (en que la única manera de llegar a la información es por medio del mecanismo que la protege) serían *mandatorios*.

No todos los sistemas operativos implementan las cuatro posibles combinaciones (compartido mandatorio, o compartido consultivo, exclusivo mandatorio y exclusivo consultivo). Como regla general, en los sistemas Windows se maneja un esquema de bloqueo obligatorio, y en sistemas Unix es de bloqueo consultivo.[3]

Cabe mencionar que el manejo de bloqueos con archivos requiere del mismo cuidado que el de bloqueo por recursos cubierto en la sección 22: dos procesos intentando adquirir un candado exclusivo sobre dos archivos pueden caer en un bloqueo mutuo tal como ocurre con cualquier otro recurso.

38.4. Tipos de archivo

Si los archivos son la *unidad lógica mínima* con la que se puede guardar información en almacenamiento secundario, naturalmente sigue que hay

[3]Esto explica el que en Windows sea tan común que el sistema mismo rechace hacer determinada operación porque *el archivo está abierto por otro programa* (bloqueo mandatorio compartido), mientras que en Unix esta responsabilidad recae en cada uno de los programas de aplicación.

archivos de diferentes tipos: cada uno podría ser un documento de texto, un binario ejecutable, un archivo de audio o video, o un larguísimo etcetera, e intentar emplear un archivo como uno de un tipo distinto puede resultar desde una frustración al usuario porque el programa no responde como éste quiere, hasta en pérdidas económicas.[4]

Hay tres estrategias principales para que el sistema operativo reconozca el tipo de un archivo:

Extensión En los sistemas CP/M de los setenta, el nombre de cada archivo se dividía en dos porciones, empleando como elemento separador al punto: el nombre del archivo y su extensión. El sistema mantenía una lista de extensiones conocidas, para las cuales sabría cómo actuar, y este diseño se propagaría a las aplicaciones, que sólo abrirían a aquellos archivos cuyas extensiones supieran manejar.

Esta estrategia fue heredada por VMS y MS-DOS, de donde la adoptó Windows; ya en el contexto de un entorno gráfico, Windows agrega, más allá de las extensiones directamente ejecutables, la relación de extensiones con los programas capaces de trabajar con ellas, permitiendo invocar a un programa con sólo dar "doble clic" en un archivo.

Como nota, este esquema de asociación de tipo de archivo permite ocultar las extensiones toda vez que ya no requieren ser del conocimiento del usuario, sino que son gestionadas por el sistema operativo, abre una vía de ataque automatizado que se popularizó en su momento: el envío de correos con extensiones engañosas duplicadas, esto es, el programa maligno (un *programa troyano*) se envía a todos los contactos del usuario infectado, presentándose por ejemplo como una imágen, con el nombre `inocente.png.exe`. Por el esquema de ocultamiento mencionado, éste se presenta al usuario como `inocente.png`, pero al abrirlo, el sistema operativo lo reconoce como un ejecutable, y lo ejecuta en vez de abrirlo en un visor de imágenes.

[4]Por ejemplo, imprimir un archivo binario resulta en una gran cantidad de hojas inútiles, particularmente tomando en cuenta que hay caracteres de control como el ASCII 12 (avance de forma, *form feed*), que llevan a las impresoras que operan en modo texto a iniciar una nueva página; llevar a un usuario a ejecutar un archivo ejecutable *disfrazado* de un documento inocuo, como se verá a continuación, fue un importante vector de infección de muchos virus.

Números mágicos La alternativa que emplean los sistemas Unix es, como siempre, simple y *elegante*, aunque indudablemente presenta eventuales lagunas: el sistema mantiene una lista compilada de las *huellas digitales* de los principales formatos que debe manejar,[5] para reconocer el contenido de un archivo basado en sus primeros bytes.

Casi todos los formatos de archivo incluyen lo necesario para que se lleve a cabo este reconocimiento, y cuando no es posible hacerlo, se intenta por medio de ciertas reglas *heurísticas*. Por ejemplo, todos los archivos de imágen en *formato de intercambio gráfico* (GIF) inician con la cadena `GIF87a` o `GIF89a`, dependiendo de la versión; los archivos del lenguaje de descripción de páginas PostScript inician con la cadena`%!`, el *Formato de Documentos Portátiles* (PDF) con`%PDF`, un documento en formatos definidos alrededor de XML inicia con `<!DOCTYPE`, etcétera. Algunos de estos formatos no están *anclados* al inicio, sino en un punto específico del primer bloque.

Un caso especial de números mágicos es el llamado *hashbang* (`#!`). Esto indica a un sistema Unix que el archivo en cuestión (típicamente un archivo de texto, incluyendo código fuente en algún lenguaje de *script*) debe tratarse como un ejecutable, y empleando como *intérprete* al comando indicado inmediatamente después del *hashbang*. Es por esto que se pueden ejecutar directamente, por ejemplo, los archivos que inician con `#!/usr/bin/bash`: el sistema operativo invoca al programa `/usr/bin/bash`, y le especifica como argumento al archivo en cuestión.

Metadatos externos Los sistemas de archivos empleado por las Apple Macintosh desde 1984 separan en dos *divisiones* (*forks*) la información de un archivo: los datos que propiamente constituyen al archivo en cuestión son la *división de datos* (*data fork*), y la información *acerca del archivo* se guardan en una estructura independiente llamada *división de recursos* (*resource fork*).

Esta idea resultó fundamental para varias de las características *amigables al usuario* que presentó Macintosh desde su introducción, particularmente, para presentar un entorno gráfico que respondiera ágilmente, sin tener que buscar los datos base de una aplicación dentro

[5] Una de las ventajas de este esquema es que cada administrador de sistema puede ampliar la lista con las huellas digitales que requiera localmente.

de un archivo de mucho mayor tamaño. La *división de recursos* cabe en pocos sectores de disco, y si se toma en cuenta que las primeras Macintosh funcionaban únicamente con discos flexibles, el tiempo invertido en leer una lista de iconos podría ser demasiada.

La división de recursos puede contener todo tipo de información; los programas ejecutables son los que le dan un mayor uso, dado que incluyen desde los aspectos gráficos (icono a mostrar para el archivo, ubicación de la ventana a ser abierta, etc.) hasta aspectos funcionales, como la traducción de sus cadenas al lenguaje particular del sistema en que está instalado. Esta división permite una gran flexibilidad, dado que no es necesario tener acceso al fuente del programa para crear traducciones y temas.

En el tema particular que concierne a esta sección, la división de recursos incluye un campo llamado *creador*, que indica cuál programa fue el que generó al archivo. Si el usuario solicita ejecutar un archivo de datos, el sistema operativo lanzaría al programa *creador*, indicándole que abra al archivo en cuestión.

Las versiones actuales de MacOS ya no emplean esta técnica, sino que una llamada *appDirectory*, para propósitos de esta discusión, la técnica base es la misma.

38.5. Estructura de los archivos y métodos de acceso

La razón principal del uso de sistemas de archivos son, naturalmente, *los archivos*. En estos se almacena información de *algún tipo*, con o sin una estructura predeterminada.

La mayor parte de los sistemas operativos maneja únicamente archivos *sin estructura* —cada aplicación es responsable de preparar la información de forma congruente, y la responsabilidad del sistema operativo es únicamente entregarlo como un conjunto de bytes. Históricamente, hubo sistemas operativos, como IBM CICS (1968), IBM MVS (1974) o DEC VMS (1977), que administraban ciertos tipos de datos en un formato básico de *base de datos*.

El hecho de que el sistema operativo no imponga estructura a un archivo no significa, claro está, que la aplicación que lo genera no lo haga. La razón por la que los sistemas creados en los últimos 30 años no han

implementado este esquema de base de datos es que le *resta* flexibilidad al sistema: el que una aplicación tuviera que ceñirse a los tipos de datos y alineación de campos del sistema operativo impedía su adecuación, y el que la funcionalidad de un archivo tipo base de datos dependiera de la versión del sistema operativo creaba un *acoplamiento* demasiado rígido entre el sistema operativo y las aplicaciones.

Esta práctica ha ido cediendo terreno para dejar esta responsabilidad en manos de procesos independientes en espacio de usuario (como sería un gestor de bases de datos tradicional), o de bibliotecas que ofrezcan la funcionalidad de manejo de archivos estructurados (como en el caso de SQL*ite*, empleado tanto por herramientas de adquisición de datos de bajo nivel como *systemtap* como por herramientas tan de escritorio como el gestor de fotografías *shotwell* o el navegador *Firefox*).

En los sistemas derivados de MS-DOS puede verse aún un remanente de los archivos estructurados: en estos sistemas, un archivo puede ser *de texto* o *binario*. Un archivo de texto está compuesto por una serie de caracteres que forman *líneas*, y la separación entre una línea y otra constituye de un *retorno de carro* (caracter ASCII 13, `CR`) seguido de un *salto de línea* (caracter ASCII 10, `LF`).[6]

El acceso a los archivos puede realizarse de diferentes maneras:

Acceso secuencial Mantiene la semántica por medio de la cual permite leer de los archivos, de forma equivalente a como lo harían las unidades de cinta mencionadas en la sección 38.1, y como lo ilustra la figura 38.1: el mecanismo principal para leer o escribir es ir avanzando consecutivamente por los bytes que conforman al archivo hasta llegar a su final.

Típicamente se emplea este mecanismo de lectura para leer a memoria código (programas o bibliotecas) o documentos, sean enteros o fracciones de los mismos. Para un contenido estructurado, como una base de datos, resultaría absolutamente ineficiente, dado que no se conoce el punto de inicio o finalización de cada uno de los registros, y

[6] Esta lógica es herencia de las máquinas de escribir manuales, en que el *salto de línea* (avanzar el rodillo a la línea siguiente) era una operación distinta a la del *retorno de carro* (devolver la cabeza de escritura al inicio de la línea). En la época de los teletipos, como medida para evitar que se perdieran caracteres mientras la cabeza volvía hasta la izquierda, se decidió separar el inicio de nueva línea en los dos pasos que tienen las máquinas de escribir, para inducir una demora que evitara la pérdida de información.

probablemente sería necesario hacer *barridos secuenciales* del archivo completo para cada una de las búsquedas.

Figura 38.1: Archivo de acceso secuencial.

Acceso aleatorio El empleo de gestores como *SQLite* u otros muchos motores de base de datos más robustos no exime al usuario de pensar en el archivo como una tabla estructurada, como lo ilustra la figura 38.2. Si la única semántica por medio de la cual el sistema operativo permitiera trabajar con los archivos fuera la equivalente a una unidad de cinta, implementar el acceso a un punto determinado del archivo podría resultar demasiado costoso.

Afortunadamente, que el sistema operativo no imponga registros de longitud fija no impide que *el programa gestor* lo haga. Si en el archivo al cual apunta el descriptor de archivo `FD` hay 2 000 registros de 75 bytes cada uno y el usuario requiere recuperar el registro número 65 hacia el buffer `registro`, puede *reposicionar* el apuntador de lectura al byte $65 \times 75 = 4\,875$ (`seek(FD, 4875)`) y leer los siguientes 75 bytes en `registro` (`read(FD, *registro, 75)`).

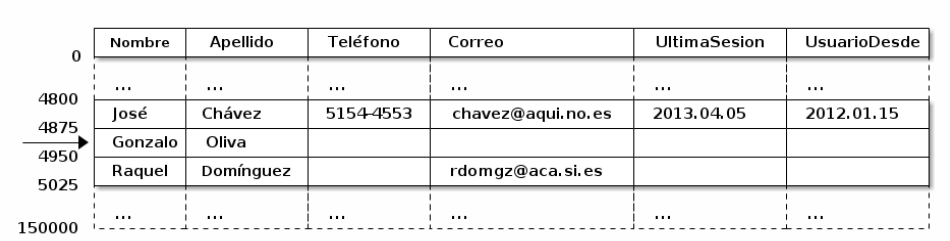


Figura 38.2: Archivo de acceso aleatorio.

Acceso relativo a índice En los últimos años se han popularizado los gestores de base de datos *débilmente estructurados* u *orientados a documentos*, llamados genéricamente *No*SQL. Estos gestores pueden guardar registros de tamaño variable en disco, por lo que, como lo ilustra la figura 38.3, no pueden encontrar la ubicación correcta por medio de los mecanismos de acceso aleatorio.

Para implementar este acceso, se divide al conjunto de datos en dos secciones (incluso, posiblemente, en dos archivos independientes): la primer sección es una lista corta de identificadores, cada uno con el punto de inicio y término de los datos a los que apunta. Para leer un registro, se emplea acceso aleatorio sobre el índice, y el apuntador se avanza a la ubicación específica que se solicita.

En el transcurso de un uso intensivo de esta estructura, dado que la porción de índice es muy frecuentemente consultada y relativamente muy pequeña, muy probablemente se mantenga completa en memoria, y el acceso a cada uno de los registros puede resolverse en tiempo muy bajo.

La principal desventaja de este modelo de indexación sobre registros de longitud variable es que sólo resulta eficiente para contenido *mayormente de lectura*: cada vez que se produce una escritura y cambia la longitud de los datos almacenados, se va generando fragmentación en el archivo, y para resolverla frecuentemente se hace necesario suspender un tiempo la ejecución de todos los procesos que lo estén empleando (e invalidar, claro, todas las copias en caché de los índices). Ahora bien, para los casos de uso en que el comportamiento predominante sea de lectura, este formato tendrá la ventaja de no desperdiciar espacio en los campos nulos o de valor irrelevante para algunos de los registros, y de permitir la flexibilidad de registrar datos originalmente no contemplados sin tener que modificar la estructura.

Es importante recalcar que la escritura en ambas partes de la base de datos (índice y datos) debe mantenerse con garantías de atomicidad — si se pierde la sincronía entre ellas, el resultado será una muy probable corrupción de datos.

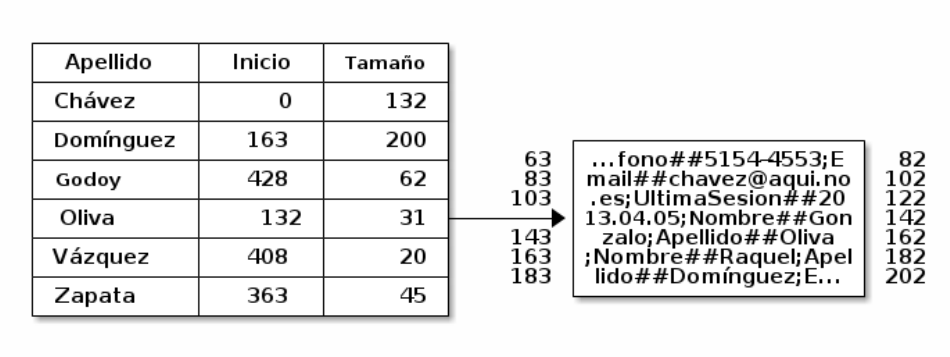


Figura 38.3: Acceso relativo a índice: tabla con apuntadores al lugar justo en un archivo sin estructura.

38.6. Archivos especiales

Los sistemas de archivos son estructuras de tan fácil manejo y comprensión que resulta natural que los diseñadores de sistemas operativos buscaran aprovecharlos para presentar todo tipo de estructuras, no sólo archivos. En este sentido, la máxima Unix, *todo es un archivo*, resulta el ejemplo natural.

Este concepto fue brevemente presentado en la sección 16; complementando dicha sección, en un sistema Unix estándar, un archivo puede pertenecer a las siguientes categorías:

Archivo estándar Aunque suene obvio, el tipo básico de archivo es, precisamente, el archivo. Representa directamente la información que lo conforma. Su semántica de uso es la presentada en el transcurso de este capítulo.

Objetos del sistema de archivos La información acerca del sistema de archivos se almacena también dentro de archivos, aunque su tratamiento es especial (con una reminiscencia de los sistemas históricos mencionados en la sección 38.5): El sistema operativo oculta al archivo *real*, y gestiona su información procesada para que lo emplee el usuario. Este tipo de archivos puede referirse a directorios o a ligas simbólicas. Ambos se detallan en la sección 39.1.

Dispositivos Permiten el acceso directo a un dispositivo externo. Como fue presentado en la sección 16, pueden ser *de bloques* o *de caracteres*.

El contenido del archivo son dos números, el *mayor* y el *menor*; tradicionalmente, el mayor indica de qué clase de dispositivo se trata, y el menor, la instancia de que se trata.

Por ejemplo, tanto los discos duros como sus particiones llevan por número mayor el ocho; los números menores para los duros son 0, 16, 32, 48. Cada uno de los discos puede tener hasta 15 particiones: 1 a 15, 17 a 31, etcétera.

Comunicación entre procesos Los archivos pueden representar *tuberías nombradas* y *sockets*. Ambos son mecanismos que permiten intercambiar información entre distintos procesos; la tubería nombrada es más sencilla de emplear, pero permite únicamente la comunicación unidireccional, en tanto que el socket permite comunicación bidireccional como si se tratara de una conexión por red.

38.7. Transferencias orientadas a bloques

Un sistema de archivos es la representación que se da a un conjunto de archivos y directorios sobre un *dispositivo de bloques*, esto es, un dispositivo que, para cualquier transferencia solicitada desde o hacia él, responderá con un bloque de tamaño predefinido.

Esto es, si bien el sistema operativo presenta una abstracción por medio de la cual la lectura (`read()`) puede ser de un tamaño arbitrario, todas las transferencias de datos desde cualquiera de los discos serán de un múltiplo del tamaño de bloques, definido por el hardware (típicamente 512 bytes).

Al leer, como en el ejemplo anterior, sólamente un registro de 75 bytes, el sistema operativo lee el bloque completo y probablemente lo mantiene en un caché en la memoria principal; si en vez de una lectura, la operación efectuada fue una de escritura (`write()`), y el sector a modificar no ha sido leído aún a memoria (o fue leído hace mucho, y puede haber sido expirado del caché), el sistema tendrá que leerlo nuevamente, modificarlo en memoria, y volver a guardarlo a disco.

Diversos sistemas han manejado otros caracteres (por ejemplo, el MacOS histórico empleaba los dos puntos, `:`), y aunque muchas veces los mantenían ocultos del usuario mediante una interfaz gráfica rica, los programadores siempre tuvieron que manejarlos explícitamente.

A lo largo del presente texto se empleará la diagonal (`/`) como separador de directorios.

39.1.2. Sistema de archivos *plano*

Los primeros sistemas de archivos limitaban el concepto de directorio a una representación plana de los archivos que lo conformaban, sin ningún concepto de *jerarquía de directorios* como el que hoy resulta natural a los usuarios. Esto se debía, en primer término, a lo limitado del espacio de almacenamiento de las primeras computadoras en implementar esta metáfora (por lo limitado del espacio de almacenamiento, los usuarios no dejaban sus archivos a largo plazo en el disco, sino que los tenían ahí meramente mientras los requerían), y en segundo término, a que no se había aún desarrollado un concepto de separación, permisos y privilegios como el que poco después aparecería.

En las computadoras personales los sistemas de archivos eran también planos en un primer momento, pero por otra razón: en los sistemas *profesionales* ya se había desarrollado el concepto; al aparecer la primer computadora personal en 1975, ya existían incluso las primeras versiones de Unix diseñadas para trabajo en red. La prioridad en los sistemas personales era mantener el código del sistema operativo simple, mínimo. Con unidades de disco capaces de manejar entre 80 y 160 KB, no tenía mucho sentido implementar directorios — si un usuario quisiera llevar a cabo una división temática de su trabajo, lo colocaría en distintos *discos flexibles*. El sistema operativo CP/M nunca soportó jerarquías de directorios, como tampoco lo hizo la primer versión de MS-DOS.[1]

El sistema de archivos original de la Apple Macintosh, MFS, estaba construido sobre un modelo plano, pero presentando la *ilusión* de directorios de una forma comparable a las etiquetas: eran representados bajo *ciertas* vistas (aunque notoriamente no en los diálogos de abrir y grabar archivos), pe-

[1] El soporte de jerarquías de directorios fue introducido apenas en la versión 2.0 de dicho sistema operativo, junto con el soporte a discos duros de 10 MB, acompañando al lanzamiento de la IBM PC modelo XT.

ro el nombre de cada uno de los archivos tenía que ser único, dado que el directorio al que pertenecía era básicamente sólo un atributo del archivo.

Y contrario a lo que dicta la intuición, el modelo de directorio plano no ha desaparecido: el sistema de *almacenamiento en la nube* ofrecido por el servicio *Amazon S3* (*simple storage service*, *servicio simple de almacenamiento*) maneja únicamente *objetos* (comparable con la definición de *archivos* que se ha venido manejando) y *cubetas* (de cierto modo comparables con las *unidades* o *volúmenes*), y permite referirse a un objeto o un conjunto de éstos basado en *filtros* sobre el total que conforman una cubeta.

Conforme se desarrollen nuevas interfaces al programador o al usuario, probablemente se popularicen más ofertas como la que hoy hace Amazon S3. Al día de hoy, sin embargo, el esquema jerárquico sigue, con mucho, siendo el dominante.

39.1.3. Directorios de profundidad fija

Las primeras implementaciones de directorios eran *de un sólo nivel*: el total de archivos en un sistema podía estar dividido en directorios, fuera por tipo de archivo (separando, por ejemplo, programas de sistema, programas de usuario y textos del correo), por usuario (facilitando una separación lógica de los archivos de un usuario de pertenecientes a los demás usuarios del sistema)

El directorio *raíz* (base) se llama en este esquema MFD (*master file directory*, *directorio maestro de archivos*), y cada uno de los directorios derivados es un UFD (*user file directory*, *directorio de archivos de usuario*).

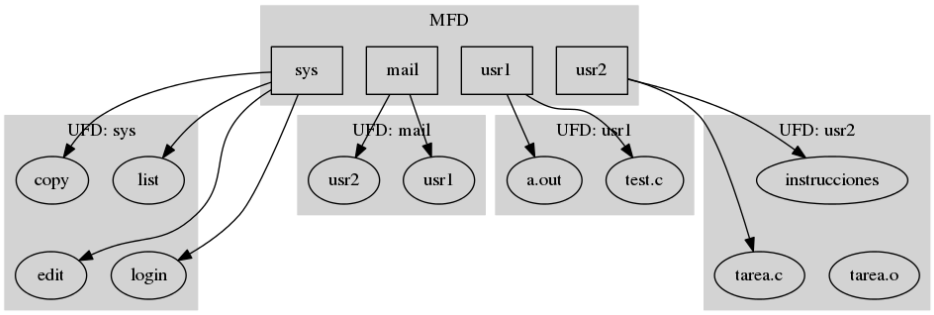


Figura 39.1: Directorio simple, limitado a un solo nivel de profundidad.

Este esquema resuelve el problema principal del nombre global único:

antes de los directorios, cada usuario tenía que cuidar que los nombres de sus archivos fueran únicos en el sistema, y ya teniendo cada uno su propio espacio, se volvió una tarea mucho más simple. La desventaja es que, si el sistema restringe a cada usuario a escribir en su UFD, se vuelve fundamentalmente imposible trabajar en algún proyecto conjunto: no puede haber un directorio que esté tanto dentro de `usr1` como de `usr2`, y los usuarios encontrarán más dificil llevar un proyecto conjunto.

39.1.4. Directorios estructurados en árbol

El siguiente paso natural para este esquema es permitir una *jerarquía ilimitada*: en vez de exigir que haya una capa de directorios, se le puede *dar la vuelta* al argumento, y permitir que cada directorio pueda contener a otros archivos o directorios anidados arbitrariamente. Esto permite que cada usuario (y que el administrador del sistema) estructure su información siguiendo criterios lógicos y piense en el espacio de almacenamiento como un espacio a largo plazo.

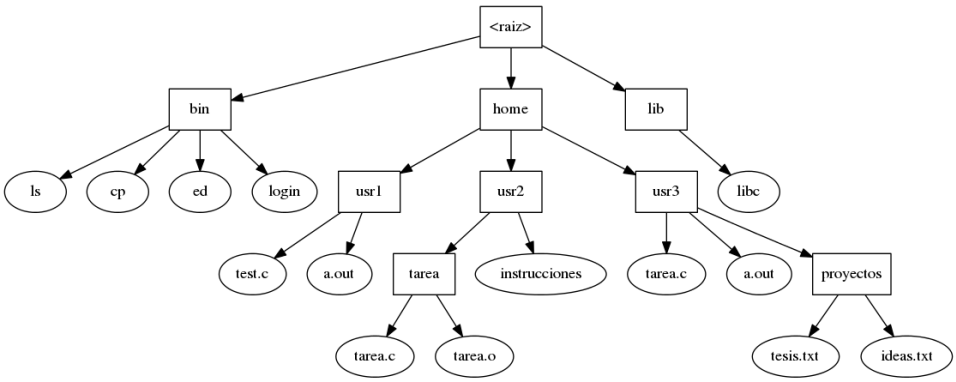


Figura 39.2: Directorio estucturado en árbol.

Junto con esta estructura nacen las *rutas de búsqueda* (*search path*): tanto los programas como las bibliotecas de sistema ahora pueden estar en cualquier lugar del sistema de archivos. Al definirle al sistema una *ruta de búsqueda*, el usuario operador puede desentenderse del lugar exacto en el que está determinado programa —el sistema se encargará de buscar en todos los directorios mencionados los programas o bibliotecas que éste requiera.[2]

[2]La *ruta de búsqueda* refleja la organización del sistema de archivos en el contexto

39.1.5. El directorio como un *grafo dirigido*

Si bien parecería que muchos de los sistemas de archivos empleados hoy en día pueden modelarse suficientemente con un árbol, donde hay un sólo nodo raíz, y donde cada uno de los nodos tiene un sólo nodo padre, la semántica que ofrecen es en realidad un *superconjunto estricto* de ésta: la de un grafo dirigido.

En un grafo dirigido como el presentado en la figura 39.3, un mismo nodo puede tener varios directorios *padre*, permitiendo, por ejemplo, que un directorio de trabajo común sea parte del directorio personal de dos usuarios. Esto es, *el mismo objeto* está presente en más de un punto del árbol.

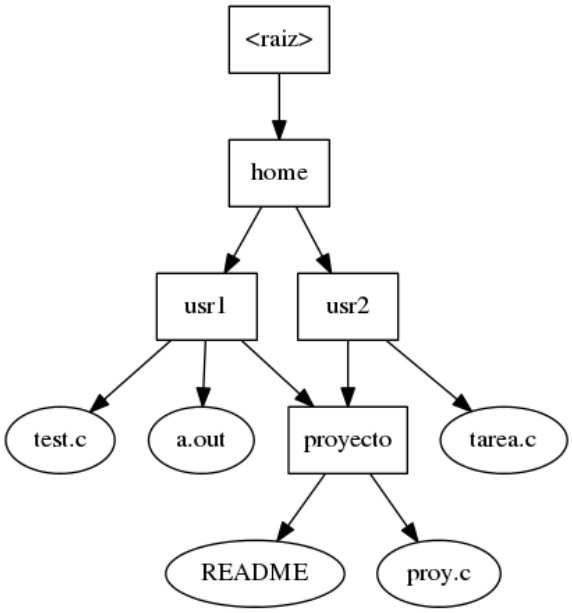


Figura 39.3: Directorio como un *grafo dirigido acíclico*: el directorio `proyecto` está tanto en el directorio `/home/usr1` como en el directorio `/home/usr2`.

Un sistema de archivos puede permitir la organización como un *grafo*

de la instalación específica. Es común que la ruta de búsqueda de un usuario estándar en Unix sea similar a `/usr/local/bin:/usr/bin:/bin:~/bin` — esto significa que cualquier comando que sea presentado es buscado, en el orden indicado, en los cuatro directorios presentados (separados por el caracter `:`, la notación `~` indica el directorio personal del usuario activo). En Windows, es común encontrar la ruta de búsqueda `c:\WINDOWS\system32;c:\WINDOWS`

dirigido, aunque es común que la interfaz que presenta al usuario[3] se restrinja a un *grafo dirigido acíclico*: las ligas múltiples son permitidas, siempre y cuando no generen un ciclo.

La semántica de los sistemas Unix implementa directorios como grafos dirigidos por medio de dos mecanismos:

Liga o enlace duro La entrada de un archivo en un directorio Unix es la relación entre la ruta del archivo y el *número de i-nodo* en el sistema de archivos.[4] Si a partir de un archivo en cualquier punto del árbol se crea una *liga dura* a él, ésta es sencillamente otra entrada en el directorio apuntando al mismo *i-nodo*. Ambas entradas, pues, son el mismo archivo —no hay uno *maestro* y uno *dependiente*.

En un sistema Unix, este mecanismo tiene sólo dos restricciones:

1. Sólo se pueden hacer ligas duras dentro del mismo volumen.
2. No pueden hacerse ligas duras a directorios, sólo a archivos.[5]

Liga o enlace simbólico Es un archivo *especial*, que meramente indica a dónde apunta. El encargado de seguir este archivo a su destino (esto es, de *resolver* la liga simbólica) es el sistema operativo mismo; un proceso no tiene que hacer nada especial para seguir la liga.

Una liga simbólica puede *apuntar* a directorios, incluso creando ciclos, o a archivos en otros volúmenes.

Cuando se crea una liga simbólica, la liga y el archivo son dos entidades distintas. Si bien cualquier proceso que abra al archivo destino estará trabajando con la misma entidad, en caso de que éste sea renombrado o eliminado, la liga quedará *rota* (esto es, apuntará a una ubicación inexistente).

Si bien estos dos tipos de liga son válidos también en los sistemas Windows,[6] en dichos sistemas sigue siendo más común emplear los *accesos directos*. Se denomina así a un archivo (identificado por su extensión, `.lnk`),

[3] Esta simplificación es simplemente una abstracción, y contiene una pequeña mentira, que será desmentida en breve.

[4] El significado y la estructura de un i-nodo se abordan en el capítulo VII.

[5] Formalmente, puede haberlas, pero sólo el administrador puede crearlas; en la sección 39.2.1 se cubre la razón de esta restricción al hablar de recorrer los directorios.

[6] Únicamente en aquellos que emplean el sistema de archivos que introdujo Windows NT, el NTFS, no en los que utilizan alguna de las variantes de FAT.

principalmente creado para poder *apuntar* a los archivos desde el escritorio y los menúes; si un proceso solicita al sistema abrir el *acceso directo*, no obtendrá al archivo destino, sino al acceso directo mismo.

Si bien el API de Win32 ofrece las funciones necesarias para emplear las ligas, tanto duras como simbólicas, éstas no están reflejadas desde la interfaz usuario del sistema — y son sistemas donde el usuario promedio no emplea una interfaz programador, sino que una interfaz gráfica. Las ligas, pues, no son más empleadas por *cuestión cultural*: en sus comunidades de usuarios, nunca fueron frecuentes, por lo cual se mantienen como conceptos empleados sólo por los *usuarios avanzados*.

Ya con el conocimiento de las ligas, y reelaborando la figura 39.3 con mayor apego a la realidad: en los sistemas operativos (tanto Unix como Windows), todo directorio tiene dos entradas especiales: los directorios `.` y `..`, que aparecen tan pronto como el directorio es creado, y resultan fundamentales para mantener la *navegabilidad* del árbol.

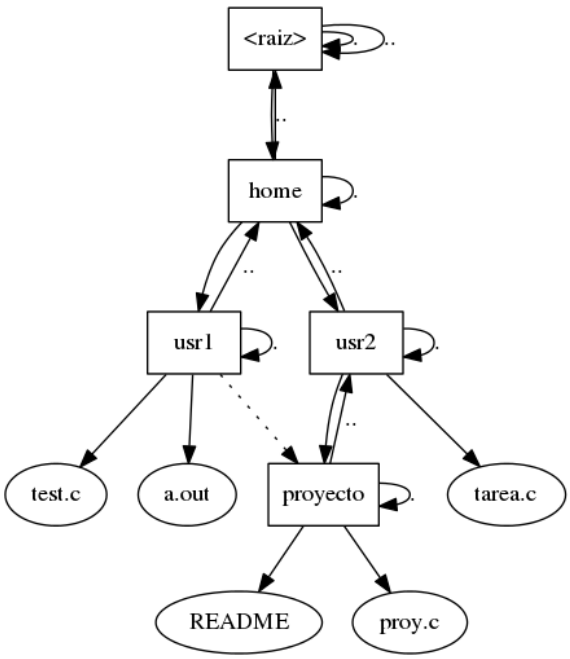


Figura 39.4: Directorio como un *grafo dirigido*, mostrando los *enlaces ocultos* al directorio actual `.` y al directorio *padre* `..`

Como se puede ver en la figura 39.4, en todos los directorios, `.` es una

liga dura al mismo directorio, y `..` es una liga al directorio *padre* (de nivel jerárquico inmediatamente superior). Claro está, como sólo puede haber una liga `..`, un directorio enlazado desde dos lugares distintos sólo apunta hacia uno de ellos con su enlace `..`; en este caso, el directorio común `proyecto` está dentro del directorio `/home/usr2`. La figura representa la *liga simbólica* desde `/home/usr1` como una línea punteada.

Hay una excepción a esta regla: el directorio raíz. En este caso, tanto `.` como `..` apuntan al mismo directorio.

Esta es la razón por la cual no se puede tomar rigurosamente un árbol de archivos como un *grafo dirigido acíclico*, ni en Windows ni en Unix: tanto las entradas `.` (al apuntar al mismo directorio donde están contenidas) como las entradas `..` (al apuntar al directorio padre) crean ciclos.

39.2. Operaciones con directorios

Al igual que los archivos, los directorios tienen una semántica básica de acceso. Éstos resultan también tipos de datos abstractos con algunas operaciones definidas. Muchas de las operaciones que pueden realizarse con los directorios son análogas a las empleadas para los archivos.[7] Las operaciones básicas a presentar son:

Abrir y cerrar Como en el caso de los archivos un programa que requiera trabajar con un directorio deberá *abrirlo* para hacerlo, y *cerrarlo* cuando ya no lo requiera. Para esto, en C, se emplean las funciones `opendir()` y `closedir()`. Estas funciones trabajan asociadas a un *flujo de directorio* (*directory stream*), que funciona de forma análoga a un descriptor de archivo.

Listado de archivos Para mostrar los archivos que conforman a un directorio, una vez que se abrió el directorio se *abre*, el programa *lee* (con `readdir()`) cada una de sus entradas. Cada uno de los resultados es una estructura `dirent` (*directory entry*, esto es, *entrada de directorio*), que contiene su nombre en `d_name`, un apuntador a su *i-nodo* en `d_ino`, y algunos datos adicionales del archivo en cuestión.

[7] De hecho, en muchos sistemas de archivos los directorios son meramente archivos de tipo especial, que son presentados al usuario de forma distinta. En la sección 43.4 se presenta un ejemplo.

Para presentar al usuario la lista de archivos que conforman un directorio, podría hacerse:

```
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
  struct dirent *archivo;
  DIR *dir;
  if (argc != 2) {
    printf("Indique el directorio a mostrar\n");
    return 1;
  }
  dir = opendir(argv[1]);
  while ((archivo = readdir(dir)) != 0) {
    printf("%s\t", archivo->d_name);
  }
  printf("\n");
  closedir(dir);
}
```

Al igual que en al hablar de archivos se puede *reposicionar* (`seek()`) al descriptor de archivo, para *rebobinar* el descriptor del directorio al principio del listado se emplea `rewinddir()`.

Buscar un elemento Si bien en el transcurso del uso del sistema operativo resulta una operación frecuente que el usuario solicite el listado de archivos dentro de un directorio, es mucho más frecuente buscar a un archivo en particular. La llamada `fopen()` antes descrita efectúa una búsqueda similar a la presentada en el ejemplo de código anterior, claro está, deteniéndose cuando encuentra al archivo en cuestión.

Crear, eliminar o renombrar un elemento Si bien estas operaciones se llevan a cabo sobre el directorio, son invocadas por medio de la semántica orientada a archivos: un archivo es creado con `fopen()`, eliminado con `remove()`, y renombrado con `rename()`.

39.2.1. Recorriendo los directorios

Es frecuente tener que aplicar una operación a todos los archivos dentro de cierto directorio, por ejemplo, para agrupar un directorio completo en un archivo comprimido, o para copiar todos sus contenidos a otro medio; procesar todas las entradas de un directorio, incluyendo las de sus subdirectorios, se denomina *recorrer el directorio* (en inglés, *directory traversal*).

Si se trabaja sobre un sistema de archivos plano, la operación de recorrido completo puede realizarse con un programa tan simple como el presentado en la sección anterior.

Al hablar de un sistema de profundidad fija, e incluso de un directorio estructurado en árbol, la lógica se complica levemente, dado que para recorrer el directorio es necesario revisar, entrada por entrada, si ésta es a su vez un directorio (y en caso de que así sea, entrar y procesar a cada uno de sus elementos). Hasta aquí, sin embargo, se puede recorrer el directorio sin mantener estructuras adicionales en memoria representando el estado.

Sin embargo, al considerar a los grafos dirigidos, se vuelve indispensable mantener en memoria la información de todos los nodos que ya han sido tocados —en caso contrario, al encontrar ciclo (incluso si éste es creado por mecanismos como las *ligas simbólicas*), se corre el peligro de entrar en un bucle infinito.

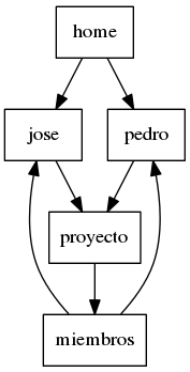


Figura 39.5: Directorio basado en grafo dirigido que incluye ciclos.

Para recorrer al directorio ilustrado como ejemplo en la figura 39.5, no bastaría tomar nota de las rutas de los archivos conforme son recorridas —cada vez que los sean procesados, su ruta será distinta. Al intentar respaldar el directorio `/home/jose/proyecto`, por ejemplo, el recorrido re-

sultante podría ser:

- `/home/jose/proyecto`
- `/home/jose/proyecto/miembros`
- `/home/jose/proyecto/miembros/jose`
- `/home/jose/proyecto/miembros/jose/proyectos`
- `/home/jose/proyecto/miembros/jose/proyectos/miembros`
- ... Y un etcétera infinito.

Para resolver esta situación, los programas que recorren directorios en los sistemas de archivos *reales* deben emplear un indexado basado en *i-nodo*,[8] que identifica sin ambigüedad a cada uno de los archivos. En el caso presentado, si el i-nodo de `jose` fuera 10 543, al consultar a los miembros de `miembros` el sistema encontrará que su primer entrada apunta al i-nodo 10 543, por lo cual la registraría sólo como un apuntador a datos *ya archivados*, y continuaría con la segunda entrada del directorio, que apunta a `pedro`.

39.2.2. Otros esquemas de organización

Por más que el uso de sistemas de archivos basados en directorios jerárquicos parece universal y es muy ampliamente aceptado, hay cada vez más casos de uso que apuntan a que se pueda estar por dar la bienvenida a una nueva metáfora de organización de archivos.

Hay distintas propuestas, y claro está, es imposible aún saber cuál dirección obtendrá el favor del mercado —o, dado que no necesariamente siga habiendo un modelo apto para todos los usos, de *qué* segmento del mercado.

39.3. *Montaje* de directorios

Para trabajar con el contenido de un sistema de archivos, el sistema operativo tiene que *montarlo*: ubicarlo en algún punto del árbol de archivos visible al sistema y al usuario.

[8] Que si bien no ha sido definido aún formalmente, para esta discusión bastará saber que es un número único por volumen.

Es muy común, especialmente en los entornos derivados de Unix, que un sistema operativo trabaje con distintos sistemas de archivos al mismo tiempo. Esto puede obedecer a varias causas, entre las cuales se encuentran:

Distintos medios físicos Si la computadora tiene más de una unidad de almacenamiento, el espacio dentro de cada uno de los discos se maneja como un sistema de archivos indepentiente. Esto es especialmente cierto en la presencia de unidades removibles (`CD`, unidades `USB`, discos duros externos, etc.)

Diferentes usos esperados Como se verá más adelante, distintos *esquemas de organización* (esto es, distintos sistemas de archivos) presentan ventajas para diversos patrones de uso. Por ejemplo, tiene sentido que una base de datos resida sobre una organización distinta a la de los programas ejecutables (binarios) del sistema.

Abstracciones de sistemas no-físicos El sistema operativo puede presentar diversas estructuras *con una estructura* de sistema de archivos. El ejemplo más claro de esto es el sistema de archivos virtual `/proc`, presente en los sistemas Unix, que permite ver diversos aspectos de los procesos en ejecución (y, en Linux, del sistema en general). Los archivos bajo `/proc` no están en ningún disco, pero se presentan como si fueran archivos estándar.

Razones administrativas El administrador del sistema puede emplear sistemas de archivos distintos para aislar espacios de usuarios entre sí: por ejemplo, para evitar que un exceso de mensajes enviados en la bitácora (típicamente bajo `/var/log`) saturen al sistema de archivos principal, o para determinar patrones de uso máximo por grupos de usuarios.

En los sistemas tipo Unix, el mecanismo para montar los archivos es el de un árbol con *puntos de montaje*. Esto es, *todos los archivos y directorios* del sistema operativo están estructurados en *un único árbol*. Cuando se solicita al sistema operativo *montar* un sistema de archivos en determinado lugar, éste se integra al árbol, ocultando todo lo que el directorio en cuestión previamente tuviera.[9]

[9]Hay implementaciones que exigen que el montaje se realice exclusivamente en directorios vacíos; se tienen otras, como UnionFS, que buscan seguir presentando una interfaz *de*

Capítulo 40

Control de acceso

Parte importante de la información que se almacena respecto a cada uno de los archivos, directorios u otros objetos son sus permisos de acceso. Éstos indican al sistema qué puede hacerse y qué no con cada uno de los objetos, posiblemente diferenciando el acceso dependiendo del usuario o clase de usuarios de que se trate.

Hay muchos esquemas de control de acceso; a continuación se presentan tres esquemas, comenzando por el más simple, y avanzando hacia los más complejos.

40.1. Sistemas FAT

El sistema de archivos FAT[1] fue diseñado a fines de los setenta, y al día de hoy es muy probablemente el sistema de archivos más ampliamente utilizado del mundo. Su diseño es suficientemente simple para ser incluido en dispositivos muy limitados; esta simplicidad se demostrará al analizar sus principales estructuras son analizadas en el capítulo VII. Y por otro lado, con sólo modificaciones relativamente menores a su diseño original, ha logrado extenderse de un sistema pensado para volúmenes de 150 KB hasta llegar a las decenas de gigabytes.

En cada una de las entradas del directorio en un sistema FAT, el byte número 12 almacena los siguientes *atributos*:

[1] Recibe su nombre de las siglas de su estructura fundamental, la *Tabla de Asignación de Archivos*, o *File Allocation Table*.

Oculto Si está encendido, el archivo no se deberá mostrar al usuario en listados de directorio.

Sólo lectura El sistema operativo debe evitar toda modificación al archivo, esto es, rechazar las llamadas al sistema que soliciten esta modificación.

Sistema Como se explicará en la sección 43.4.2, un sistema de archivos FAT tiende conforme se va utilizando a *fragmentar* los archivos que almacena. La carga del sistema operativo tiene que realizarse empleando código lo más compacto y sencillo posible: sin siquiera tener conocimiento del sistema de archivos. Este atributo indica que el sistema operativo debe cuidar no mover ni fragmentar al archivo en cuestión.

Archivado MS-DOS incluía herramientas bastante sencillas para realizar respaldos; estas basaban su operación en este atributo: Conforme se realizaba un respaldo, todos los archivos iban siendo marcados como *archivados*. Y cuando un archivo era modificado, el sistema operativo retiraba este atributo. De esta manera, el siguiente respaldo contendría únicamente los archivos que habían sido modificados desde el respaldo anterior.

Estos atributos eran suficientes para las necesidades de un sistema de uso personal de hace más de tres décadas. Sin embargo, y dado que está reservado todo un byte para los atributos, algunos sistemas extendieron los atributos cubriendo distintas necesidades. Estos cuatro atributos son, sin embargo, la base del sistema.

Puede verse que, bajo MS-DOS, no se hace diferencia entre *lectura* y *ejecución*. Como se presentó en la sección 38.4, el sistema MS-DOS basa la identificación de archivos en su *extensión*. Y, claro, siendo un sistema concebido como *monousuario*, no tiene sentido condicionar la *lectura* de un archivo: el sistema operativo siempre permitirá la lectura de cualquier archivo.

40.2. Modelo tradicional Unix

El sistema Unix precede a MS-DOS por una década. Sin embargo, su concepción es de origen para un esquema multiusuario, con las provisiones necesarias para que cada uno de ellos indique qué usuarios pueden o no tener diferentes tipos de acceso a los archivos.

En Unix, cada uno de los usuarios pertenece a uno o más *grupos*. Estos grupos son gestionados por el administrador del sistema.

Cada objeto en el sistema de archivos describe sus permisos de acceso por nueve bits, y con el identificador de su usuario y grupo propietarios.

Estos bits están dispuestos en tres grupos de tres; uno para el *usuario*, uno para el *grupo*, y uno para los *otros*. Por último, los tres grupos de tres bits indican si se otorga permiso de *lectura*, *escritura* y *ejecución*. Estos permisos son los mismos que fueron presentados en la sección 32.1, relativos al manejo de memoria.

Estos tres grupos de tres permiten suficiente *granularidad* en el control de acceso para expresar casi cualquier combinación requerida.

Por poner un ejemplo, si los usuarios `jose`, `pedro` y `teresa` están colaborando en el desarrollo de un sistema de facturación, pueden solicitar al administrador del sistema la creación de un grupo, `factura`. Una vez creado el sistema, y suponiendo que el desarrollo se efectúa en el directorio `/usr/local/factura/`, podrían tener los siguientes archivos:

```
$ ls -la /usr/local/factura/
drwxrwxr-x root   factura   4096 .
drwxr-xr-x root   root      4096 ..
-rw-r----- root   factura  12055 compilacion.log
-rwxrwxr-x pedro  factura 114500 factura
-rw-rw-r-- teresa factura  23505 factura.c
-rw-rw-r-- jose   factura   1855 factura.h
-rw-rw---- teresa factura  36504 factura.o
-rw-rw---- teresa factura  40420 lineamientos.txt
-rw-rw-r-- pedro  factura   3505 Makefile
```

Este ejemplo muestra que cada archivo puede pertenecer a otro de los miembros del grupo.[2] Para el ejemplo aquí mostrado, el usuario propietario es menos importante que el grupo al que pertenece.

La ejecución sólo está permitida, para los miembros del grupo `factura`, en el archivo `factura`: los demás archivos son ya sea el código fuente que se emplea para *generar* dicho archivo ejecutable, o la información de desarrollo y compilación.

[2]Los archivos pertenecen al usuario que los creó, a menos que su propiedad sea *cedida* por el administrador a otro usuario.

de archivos —a continuación se abordará la estrategia que sigue FAT. Cabe recordar que FAT fue diseñado cuando el medio principal de almacenamiento era el disco flexible, decenas de veces más lento que el disco duro, y con mucha menor confiabilidad.

Cuando se le solicita a un sistema de archivos FAT eliminar un archivo, éste no se borra del directorio, ni su información se libera de la tabla de asignación de archivos, sino que se *marca* para ser ignorado, reemplazando el primer caracter de su nombre por `0xE5`. Ni la entrada de directorio, ni la *cadena* de *clusters* correspondiente en las tablas de asignación,[11] son eliminadas —sólo son marcadas como *disponibles*. El espacio de almacenamiento que el archivo eliminado ocupa debe, entonces, ser *sumado* al espacio libre que tiene el volumen. Es sólo cuando se crea un nuevo archivo empleando esa misma entrada, o cuando otro archivo ocupa el espacio físico que ocupaba el que fue eliminado, que el sistema operativo marca *realmente* como desocupados los *clusters* en la tabla de asignación.

Es por esto por lo que desde los primeros días de las PC hay tantas herramientas de recuperación (o *des-borramiento*, *undeletion*) de archivos: siempre que no haya sido creado uno nuevo que ocupe la entrada de directorio en cuestión, recuperar un archivo es tan simple como volver a ponerle el primer caracter a su nombre.

Este es un ejemplo de un *mecanismo flojo* (en contraposición de los *mecanismos ansiosos*, como los vistos en la sección 34.1). Eliminar un archivo requiere de un trabajo mínimo, mismo que es *diferido* al momento de reutilización de la entrada de directorio.

43.5. Compresión y *desduplicación*

Los archivos almacenados en un área dada del disco tienden a presentar patrones comunes. Algunas situaciones ejemplo que llevarían a estos patrones comunes son:

- Dentro del directorio de trabajo de cada uno de los usuarios hay típicamente archivos creados con los mismos programas, compartiendo encabezado, estructura, y ocasionalmente incluso parte importante de los datos.

[11] Este tema será abordado en breve, en la sección 44.4, al hablar de las tablas de asignación de archivos.

- Dentro de los directorios de binarios del sistema, habrá muchos archivos ejecutables compartiendo el mismo *formato binario*.
- Es muy común también que un usuario almacene versiones distintas del mismo documento.
- Dentro de un mismo documento, es frecuente que el autor repita en numerosas ocasiones las palabras que describen sus conceptos principales.

Conforme las computadoras aumentaron su poder de cómputo, desde finales de los ochenta se presentaron varios mecanismos que permitían aprovechar las regularidades en los datos almacenados en el disco para comprimir el espacio utilizable en un mismo medio. La compresión típicamente se hace por medio de mecanismos estimativos derivados del análisis del contenido,[12] que tienen como resultado un nivel variable de compresión: con tipos de contenido altamente regulares (como podría ser texto, código fuente, o audio e imágenes *en crudo*), un volumen puede almacenar frecuentemente mucho más de 200% de su espacio real.

Con contenido poco predecible o con muy baja redundancia (como la mayor parte de formatos de imágenes y audio, que incluyen ya una fase de compresión, o empleando cualquier esquema de cifrado) la compresión no ayuda, y sí reduce el rendimiento global del sistema en que es empleada.

43.5.1. Compresión de volumen completo

El primer sistema de archivos que se popularizó fue *Stacker*, comercializado a partir de 1990 por *Stac Electronics*. *Stacker* operaba bajo MS-DOS, creando un dispositivo de bloques virtual alojado en un disco estándar.[13] Varias implementaciones posteriores de esta misma época se basaron en este mismo principio.

Ahora, sumando la variabilidad derivada del enfoque probabilístico al uso del espacio con el ubicarse como una compresión orientada al volu-

[12] Uno de los algoritmos más frecuentemente utilizados y fáciles de entender es la *Codificación Huffman*; éste y la familia de algoritmos *Lempel-Ziv* sirven de base para prácticamente la totalidad de las implementaciones.

[13] Esto significa que, al solicitarle la creación de una unidad comprimida de 30 MB dentro del volumen C (disco duro primario), ésta aparecería disponible como un volumen adicional. El nuevo volumen requería de la carga de un *controlador* especial para ser *montado* por el sistema operativo.

men entero, resulta natural encontrar una de las dificultades resultantes del uso de estas herramientas: dado que el sistema operativo estructura las operaciones de lectura y escritura por bloques de dimensiones regulares (por ejemplo, el tamaño típico de sector hardware de 512 bytes), al poder éstos traducirse a más o menos bloques reales tras pasar por una capa de compresión, es posible que el sistema de archivos tenga que reacomodar constantemente la información al *crecer* alguno de los bloques previos. Conforme mayor era el tiempo de uso de una unidad comprimida por *Stacker*, se notaba más degradación en su rendimiento.

Además, dado que bajo este esquema se empleaba el sistema de archivos estándar, las tablas de directorio y asignación de espacio resultaban también comprimidas. Estas tablas, como ya se ha expuesto, contienen la información fundamental del sistema de archivos; al comprimirlas y reescribirlas constantemente, la probabilidad de que resulten dañadas en alguna falla (eléctrica o lógica) aumenta. Y si bien los discos comprimidos por *Stacker* y otras herramientas fueron populares, principalmente durante la primera mitad de los noventa, conforme aumentó la capacidad de los discos duros fue declinando su utilización.

43.5.2. Compresión archivo por archivo

Dado el éxito del que gozó *Stacker* en sus primeros años, Microsoft anunció como parte de las características de la versión 6.0 de MS-DOS (publicada en 1993) que incluiría *DoubleSpace*, una tecnología muy similar a la de *Stacker*. Microsoft incluyó en sus sistemas operativos el soporte para DoubleSpace por siete años, cubriendo las últimas versiones de MS-DOS y las de Windows 95, 98 y Millenium, pero como ya se vio, la compresión de volumen completo presentaba importantes desventajas.

Para el entonces nuevo sistemas de archivos NTFS, Microsoft decidió incluir una característica distinta, más segura y más modular: mantener el sistema de archivos funcionando de forma normal, sin compresión, y habilitar la compresión *archivo por archivo* de forma transparente al usuario.

Este esquema permite al administrador del sistema elegir, por archivos o carpetas, qué areas del sistema de archivos desea almacenar comprimidas; esta característica viene como parte de todos los sistemas operativos Windows a partir de la versión XP, liberada en el año 2003.

Si bien la compresión transparente por archivo se muestra mucho más atractiva que la de volumen completo, no es una panacea y es frecuente

Capítulo 45

Fallos y recuperación

El sistema de archivos FAT es *relativamente frágil*: no es difícil que se presente una situación de *corrupción de metadatos*, y muy particularmente, que ésta afecte la estructura de las tablas de asignación. Los usuarios de sistemas basados en FAT en Windows sin duda conocen a `CHKDSK` y `SCANDISK`, dos programas que implementan la misma funcionalidad base, y difieren principalmente en su interfaz al usuario: `CHKDSK` existe desde los primeros años de MS-DOS, y está pensado para su uso interactivo en línea de comando; `SCANDISK` se ejecuta desde el entorno gráfico, y presenta la particularidad de que no requiere (aunque sí recomienda fuertemente) *acceso exclusivo* al sistema de archivos mientras se ejecuta.

¿Cómo es el funcionamiento de estos programas?

A lo largo de la vida de un sistema de archivos, conforme los archivos se van asignando y liberando, cambian su tamaño, y conforme el sistema de archivos se monta y desmonta, pueden ir apareciendo *inconsistencias* en su estructura. En los sistemas tipo FAT, las principales inconsistencias[1] son:

Archivos cruzados (*cross-linked file*) Recuérdese que la entrada en el directorio de un archivo incluye un apuntador al primer *cluster* de una *cadena*. Cada cadena debe ser única, esto es, ningún *cluster* debe pertenecer a más de un archivo. Si dos archivos incluyen al mismo *cluster*, esto indica una inconsistencia, y la única forma de resolverla es *truncar* uno de los archivos en el punto inmediato anterior a este cruce.

[1] Que no las únicas. Éstas y otras más están brevemente descritas en la página de manual de `dosfsck` (véase la sección 46.3).

Cadenas perdidas o *huérfanas* (*lost clusters*) Cuando hay espacio marcado como ocupado en la tabla de asignación, pero no hay ninguna entrada de directorio haciendo referencia a ella, el espacio está efectivamente bloqueado y, desde la perspectiva del usuario, inutilizado; además, estas cadenas pueden ser un archivo que el usuario aún requiera.

Este problema resultó tan frecuente en versiones históricas de Unix que incluso hoy es muy común tener un directorio llamado `lost+found` en la raíz de todos los sistemas de archivos, donde `fsck` (el equivalente en Unix de `CHKDSK`) creaba ligas a los archivos perdidos por corrupción de metadatos.

Cada sistema de archivos podrá presentar un distinto conjunto de inconsistencias, dependiendo de sus estructuras básicas y de la manera en que cada sistema operativo las maneja.

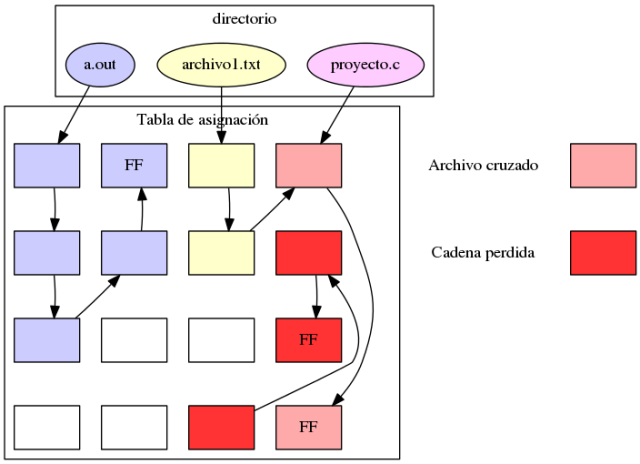


Figura 45.1: Inconsistencias en un sistema de archivos tipo FAT.

En la década de los 1980 comenzaron a venderse los *controladores de disco inteligentes*, y en menos de 10 años dominaban ya el mercado. Estos controladores, con interfaces físicas tan disímiles como SCSI, IDE, o los más modernos, SAS y SATA, introdujeron muchos cambios que fueron disociando cada vez más al sistema operativo de la gestión física directa de los dispositivos; en el apéndice X se presenta con mayor profundidad lo que esto ha significado para el desarrollo de sistemas de archivos y algoritmos relacionados. Sin embargo, para el tema en discusión, los *controladores inteligentes*

resultan relevantes porque, si bien antes el sistema operativo podía determinar con toda certeza si una operación se había realizado o no, hoy en día los controladores dan un *acuse de recibo* a la información en el momento en que la colocan en el caché incorporado del dispositivo —en caso de un fallo de corriente, esta información puede no haber sido escrita por completo al disco.

Es importante recordar que las operaciones con los metadatos que conforman el sistema de archivos no son atómicas. Por poner un ejemplo, crear un archivo en un volumen FAT requiere:

1. Encontrar una lista de *clusters* disponibles suficiente para almacenar la información que conformará al archivo.
2. Encontrar el siguiente espacio disponible en el directorio.
3. Marcar en la tabla de asignación la secuencia de *clusters* que ocupará el archivo.
4. Crear en el espacio encontrado una entrada con el nombre del archivo, apuntando al primero de sus *clusters*.
5. Almacenar los datos del archivo en cuestión en los *clusters* determinados en el paso 1.

Cualquier fallo que se presente después del tercer paso (tras efectuarse la primera modificación) tendrá como consecuencia que el archivo resulte corrupto, y muy probablemente todo que el sistema de archivos *presente inconsistencias* o *esté en un estado inconsistente*.

45.1. Datos y metadatos

En el ejemplo recién presentado, el sistema de archivos estará en un estado consistente siempre que se terminen los pasos 3 y 4 —la consistencia del sistema de archivos es independiente de la validez de los datos del archivo. Lo que busca el sistema de archivos, más que garantizar la integridad de los *datos* de uno de los archivos, es asegurar la de los *metadatos*: los datos que describen la estructura.

En caso de que un usuario desconecte una unidad a media operación, es muy probable que se presente pérdida de información, pero el sistema de

archivos debe buscar no presentar ningún problema que ponga en riesgo *operaciones posteriores* o *archivos no relacionados*. La corrupción y recuperación de datos en archivos corruptos y truncados, si bien es también de gran importancia, cae más bien en el terreno de las aplicaciones del usuario.

45.2. Verificación de la integridad

Cada sistema operativo incluye programas para realizar verificación (y, en su caso, corrección) de la integridad de sus sistemas de archivo. En el caso de MS-DOS y Windows, como ya se vio, estos programas son `CHKDSK` y `SCANDISK`; en los sistemas Unix, el programa general se llama `fsck`, y típicamente emplea a asistentes según el tipo de sistema que haya que revisar — `fsck.vfat`, `fsck.ext2`, etcétera.

Estos programas hacen un *barrido* del sistema de archivos, buscando evidencias de inconsistencia. Esto lo hacen, en líneas generales:

- Siguiendo todas las cadenas de *clusters* de archivos o tablas de i-nodos, y verificando que no haya archivos cruzados (compartiendo erróneamente bloques).
- Verificando que todas las cadenas de *clusters*, así como todos los directorios, sean alcanzables y sigan una estructura válida.
- Recalculando la correspondencia entre las estructuras encontradas y los diferentes bitmaps y totales de espacio vacío.

Estas operaciones son siempre procesos intensivos y complejos. Como requieren una revisión profunda del volumen entero, es frecuente que duren entre decenas de minutos y horas. Además, para poder llevar a cabo su tarea deben ejecutarse teniendo acceso exclusivo al volumen a revisar, lo cual típicamente significa colocar al sistema completo en modo de mantenimiento.

Dado el elevado costo que tiene verificar el volumen entero, en la década de los noventa surgieron varios esquemas orientados a evitar la necesidad de invocar a estos programas de verificación: las *actualizaciones suaves*, los *sistemas de archivos con bitácora*, y los *sistemas de archivos estructurados en bitácora*.

45.3. Actualizaciones suaves

El esquema de *actualizaciones suaves* (*soft updates*) aparentemente es el más simple de los que se presentan, pero su implementación resultó mucho más compleja de lo inicialmente estimado y, en buena medida, por esta causa no ha sido empleado más ampliamente. La idea básica detrás de este esquema es estructurar el sistema de archivos de una forma más simple y organizar las escrituras del mismo de modo que el estado resultante *no pueda* ser inconsistente, ni siquiera en caso de fallo, y de exigir que todas las operaciones de actualización de metadatos se realicen de forma *síncrona*.[2]

Ante la imposibilidad de tener un sistema *siempre consistente*, esta exigencia se relajó para permitir inconsistencias *no destructivas*: pueden presentarse *cadenas perdidas*, dado que esto no pone en riesgo a ningún archivo, sólo disminuye el espacio total disponible.

Esto, aunado a una reestructuración del programa de verificación (`fsck`) como una tarea *ejecutable en el fondo*[3] y en una tarea de *recolector de basura*, que no requiere intervención humana (dado que no pueden presentarse inconsistencias destructivas), permite que un sistema de archivos que no fue *limpiamente desmontado* pueda ser montado y utilizado de inmediato, sin peligro de pérdida de información o de corrupción.

Al requerir que todas las operaciones sean síncronas, parecería que el rendimiento global del sistema de archivos tendría que verse afectado, pero por ciertos patrones de acceso muy frecuentes, resulta incluso beneficioso. Al mantenerse un ordenamiento lógico entre las dependencias de todas las operaciones pendientes, el sistema operativo puede *combinar* muchas de estas y reducir de forma global las escrituras a disco.

A modo de ejemplos: si varios archivos son creados en el mismo directorio de forma consecutiva, cada uno de ellos mediante una llamada `open()` independiente, el sistema operativo combinará todos estos accesos en uno solo, reduciendo el número de llamadas. Por otro lado, un patrón frecuente en sistemas Unix es que, para crear un archivo de uso temporal reforzando la confidencialidad de su contenido, el proceso solicite al sistema la creación de un archivo, abra el archivo recién creado, y ya teniendo al

[2] Esto es, no se le reporta éxito en alguna operación de archivos al usuario sino hasta que ésta es completada y grabada a disco.

[3] Una tarea que no requiere de intervención manual por parte del operador, y se efectúa de forma automática como parte de las tareas de mantenimiento del sistema.

descriptor de archivo, lo elimine —en este caso, con estas tres operaciones seguidas, *soft updates* podría ahorrarse por completo la escritura a disco.

Esta estrategia se vio afectada por los controladores inteligentes: si un disco está sometido a carga intensa, no hay garantía para el sistema operativo del orden que seguirán *en verdad* sus solicitudes, que se *forman* en el caché propio del disco. Dado que las actualizaciones suaves dependen tan profundamente de confiar en el ordenamiento, esto rompe por completo la confiabilidad del proceso.

Las actualizaciones suaves fueron implementadas hacia 2002 en el sistema operativo FreeBSD, y fueron adoptadas por los principales sistemas de la familia BSD, aunque NetBSD lo retiró en 2012, prefiriendo el empleo de sistemas con bitácora, tema que será tratado a continuación. Muy probablemente, la lógica detrás de esta decisión sea la cantidad de sistemas que emplean esta segunda estrategia, y lo complejo que resulta dar mantenimiento dentro del núcleo a dos estrategias tan distintas.

45.4. Sistemas de archivo con bitácora

Este esquema tiene su origen en el ámbito de las bases de datos distribuidas. Consiste en separar un área del volumen y dedicarla a llevar una bitácora (*journal*) con todas las *transacciones* de metadatos.[4] Una *transacción* es un conjunto de operaciones que deben aparecer como atómicas.

La bitácora se implementa generalmente como una *lista ligada circular*, con un apuntador que indica cuál fue la última operación realizada exitosamente. Periódicamente, o cuando la carga de transferencia de datos disminuye, el sistema verifica qué operaciones quedaron pendientes, y *avanza* sobre la bitácora, marcando cada una de las transacciones conforme las realiza.

En caso de tener que recuperarse de una condición de fallo, el sistema operativo sólo tiene que leer la bitácora, encontrar cuál fue la última operación efectuada, y aplicar las restantes.

Una restricción de este esquema es que las transacciones guardadas en la bitácora deben ser *idempotentes*— esto es, si una de ellas es efectuada dos

[4] Hay implementaciones que registran también los datos en la bitácora, pero tanto por el tamaño que ésta requiere como por el efecto en velocidad que significa, son poco utilizadas. La sección 45.5 presenta una idea que elabora sobre una bitácora que almacena tanto datos como metadatos.

veces, el efecto debe ser exactamente el mismo que si hubiera sido efectuada una sola vez. Por poner un ejemplo, no sería válido que una transacción indicara "agregar al directorio x un archivo llamado y", dado que si la falla se produce después de procesar esta transacción pero antes de avanzar al apuntador de la bitácora, el directorio x quedaría con dos archivos y —una situación inconsistente. En todo caso, tendríamos que indicar "registrar al archivo y en la posición z del directorio x"; de esta manera, incluso si el archivo ya había sido registrado, puede volverse a hacerlo sin peligro.

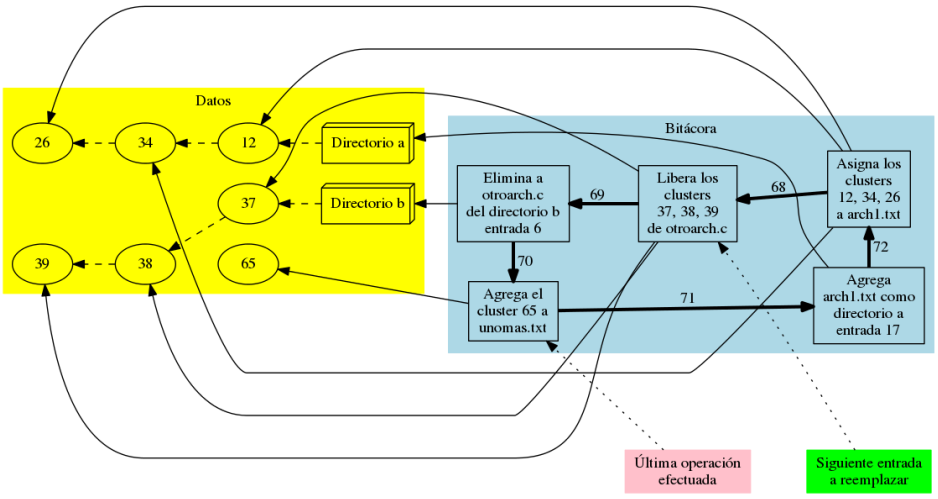


Figura 45.2: Sistema de archivos con bitácora.

Este esquema es el más utilizado hoy en día, y está presente en casi todos los sistemas de archivos modernos. Si bien con un sistema con bitácora no hace falta verificar el sistema de archivos completo tras una detención abrupta, esta no exime de que, de tiempo en tiempo, el sistema de archivos sea verificado: es altamente recomendado hacer una verificación periódica en caso de presentarse alguna corrupción, sea por algún bug en la implementación, fallos en el medio físico, o factores similarmente poco frecuentes.

La mayor parte de los sistemas de archivos incluyen contadores de *cantidad de montajes* y de *fecha del último montaje*, que permiten que el sistema operativo determine, automáticamente, si corresponde hacer una verificación preventiva.

45.5. Sistemas de archivos estructurados en bitácora

Si se lleva el concepto del sistema de archivos con bitácora a su límite, y se designa a *la totalidad* del espacio en el volumen como la bitácora, el resultado es un sistema de archivos *estructurado en bitácora* (*log-structured file systems*). Obviamente, este tipo de sistemas de archivos presenta una organización completa radicalmente diferente de los sistemas de archivo tradicionales.

Las ideas básicas detrás de la primer implementación de un sistema de archivos de esta naturaleza (Ousterhut y Rosenblum, 1992) apuntan al empleo agresivo de caché de gran capacidad, y con un fuerte mecanismo de *recolección de basura*, reacomodando la información que esté más cerca de la *cola* de la bitácora (y liberando toda aquella que resulte redundante).

Este tipo de sistemas de archivos facilita las escrituras, haciéndolas siempre secuenciales, y buscan –por medio del empleo del caché ya mencionado– evitar que las cabezas tengan que desplazarse con demasiada frecuencia para recuperar fragmentos de archivos.

Una consecuencia directa de esto es que los sistemas de archivos estructurados en bitácora fueron los primeros en ofrecer *fotografías* (*snapshots*) del sistema de archivos: es posible apuntar a un momento en el tiempo, y –con el sistema de archivos aún en operación– montar una copia de sólo lectura con la información del sistema de archivos *completa* (incluyendo los datos de los archivos).

Los sistemas de archivo estructurados en bitácora, sin embargo, no están optimizados para cualquier carga de trabajo. Por ejemplo, una base de datos relacional, en que cada uno de los registros es típicamente actualizado de forma independiente de los demás, y ocupan apenas fracciones de un bloque, resultaría tremendamente ineficiente. La implementación referencia de Ousterhut y Rosenblum fue parte de los sistemas BSD, pero dada su tendencia a la *extrema fragmentación*, fue eliminado de ellos.

Este tipo de sistemas de archivo ofrece características muy interesantes, aunque es un campo que aún requiere de mucha investigación e implementaciones ejemplo. Muchas de las implementaciones en sistemas libres han llegado a niveles de funcionalidad aceptables, pero por diversas causas han ido perdiendo el interés o el empuje de sus desarrolladores, y su ritmo de desarrollo ha decrecido. Sin embargo, varios conceptos muy importantes han nacido bajo este tipo de sistemas de archivos, algunos de los cuales (como el de las *fotografías*) se han ido aplicando a sistemas de archivo

1. Organización de la memoria física

1.1. Zonas de memoria

El sistema operativo debe decidir qué porción de la memoria física asigna a memoria interna del núcleo (en donde habrá buffers, datos y el código mismo del núcleo) y qué porción será asignada dinámicamente (memoria dinámica *del sistema*) para procesos y cachés. Además hay porciones de memoria reservadas por el hardware o para uso específico del hardware y que el núcleo no puede utilizar libremente (p.ej.: dispositivos mapeados en memoria, vectores de interrupciones, etc...). La configuración exacta depende de la arquitectura de hardware, pero en los x86 GNU/Linux utiliza la siguiente disposición (fig.: 1):

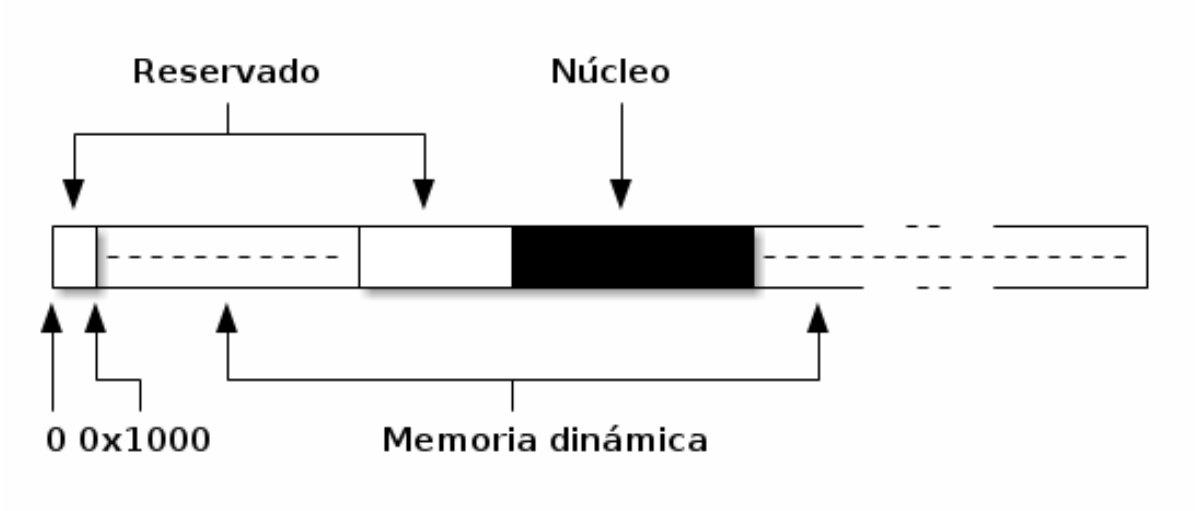


Figura 1: Configuración de memoria en un x86

- Bloque [0 .. 0x0fff] (primeros 4 kb -1 página-): reservado para el hardware (configuración de hardware detectada durante el POST[1]).
- Bloque [0x1000 .. 0x9ffff]: disponible para memoria dinámica.
- Bloque [0xa0000 .. 0xfffff]: reservado para el hardware (dispositivos mapeados en memoria y rutinas de bios), este el concoido "agujero" desde los 640 Kb hasta 1Mb..
- Bloque [0x100000 .. 0x100000+sizeof(kernel)]: núcleo.
- Bloque [0x100000+sizeof(kernel) .. DIRMAX]: disponible para memoria dinámica.

Se decide cargar el núcleo después de la memoria reservada para el hardware para poder cargarlo en un espacio de memoria contiguo (es demasiado grande para cargarlo antes de ls 640 Kb). De esta forma, el código del núcleo se mantiene sencillo (sin huecos) y además es más fácil cargarlo en memoria.

[1] Power On Self Test - Pruebas realizadas por la BIOS durante la inicialización de la máquina

1.2. Algunas optimizaciones y características a tener en cuenta

GNU/Linux implementa memoria compartida entre procesos (e hilos). Así, una página que está en memoria puede estar siendo referenciada desde una, dos o más tablas de paginación. Además se implementa Copy On Write para evitar duplicar todas las páginas de un proceso en `fork()`. Por último, también se soporta carga por demanda: las páginas se van leyendo o creando a partir de la informaciòn del programa binario a medida que es necesario.

Otra característica a tener en cuenta es el mapeo de archivos en memoria (memory mapped files): un proceso puede solicitar que una porción de su memoria virtual se refiera a un archivo. De esta forma, escribir/leer en un archivo (desde el punto de vista del proceso) consiste en escribir/leer directamente en una dirección de memoria. Eventualmente, para liberar esa página no es conveniente escribirla en swap, sino que conviene escribirla directamente en el espacio reservado para el archivo.

Además, para optimizar el rendimiento de los dispositivos de almacenamiento secundario, se implementa la *lectura por adelantado* (readahead): cuando se lee un bloque es muy alta la probabilidad de que sea necesario leer en poco tiempo más el bloque siguiente. Por las velocidades de posicionamiento de los dispositivos tal vez no convenga esperar a que el proceso haga la solicitud: unos pocos milisegundos seguramente harán que el disco pase sobre el siguiente bloque a leer y haya que esperar una nueva rotación completa para volver a leer el bloque (y la cosa empeora mucho más si también se desplazó el brazo lector). Por esto se utilizan cachés de disco que permiten leer secuencialmente varios bloques contiguos cuando en principio sólo se necesita el primero.

1.3. NUMA

Además GNU/Linux soporta *Acceso No Uniforme a Memoria* (NUMA - Not Uniform Memory Access). NUMA se utiliza típicamente en equipos multiprocesadores, en donde determinadas porciones de memoria (nodos) están asociadas a un procesador brindando una velocidad de acceso mayor a ese procesador (con respecto al resto de la memoria del cluster o del equipo). Así, cuando se utiliza NUMA la memoria es particionada en nodos y a su vez cada nodo puede subdividirse en zonas. Muchas de las estructuras de datos relacionadas con la gestión de memoria deben entonces replicarse a cada nodo (y si es necesario en cada zona).

1.4. Páginas compartidas, paginación y reverse mapping

Para la paginación en i386 se decide utilizar páginas de 4Kb, utilizando tablas de paginación de 4 niveles.

El sistema permite que un marco de página sea utilizado por varios procesos (p. ej.: para archivos mapeados en la memoria de dos procesos o para páginas marcadas para Copy On Write).

1.4.1. Reverse mapping

Además de las tablas de paginación de cada proceso, para el funcionamiento del algoritmo de reemplazo de páginas se debe almacenar cierta información sobre cada marco "candidato", por ejemplo: si la página está bloqueada en memoria (temporalmente o por pedido del proceso), qué entradas en tablas de traducciones están apuntando a esta página

(o sea: qué procesos están utilizando esta página), si fue utilizada hace poco, etc... Los punteros a las entradas en las tablas de traducciones son necesarios para poder actualizar las tablas correspondientes cuando se elige la página como víctima.

El proceso de obtener las entradas que apuntan a determinado marco se llama *mapeo inverso* (reverse mapping). Para almacenar los punteros a las entradas de traducción (translation entries) que apuntan a cada marco se podría utilizar una lista. Sin embargo, hacerlo así incrementaría significativamente la sobrecarga del núcleo y en su lugar se utilizan punteros a regiones de memoria que permiten obtener (a través de otras tablas relacionadas) la misma información.

1.5. Otras reservas

Por otro lado, hay otras porciones de memoria que requieren tratamiento particular. Veamos: si se necesita un nuevo marco de memoria como parte del procesamiento de una interrupción no se puede esperar a que el gestor de memoria envíe una página al disco o actualice muchas estructuras de datos (pues los tiempos de procesamiento de muchas interrupciones deben ser cortos para poder sincronizarse con la velocidad de los dispositivos). Otro caso similar es durante el procesamiento de código en una sección crítica del núcleo.

Para estos casos, cuando se requiere memoria, el núcleo no hace un pedido normal de memoria, sino que existe un mecanismo llamado *pedido de ubicación atómica* (atomic allocation request) que debe resolver instantáneamente la demanda (o retornar un error). Para esto se necesita una reserva (pool) de páginas destinadas sólo a esos casos.

2. Ubicación de memoria y gestión de memoria libre

Se ha visto que una solución al problema de la fragmentación externa es utilizar el mecanismo de paginación para que no queden huecos en memoria que luego obliguen a desfragmentar la memoria. Si bien GNU/Linux utiliza paginación en los espacios de direcciones de los procesos, hacerlo también para la memoria gestionada por el núcleo y para rastrear la memoria libre/usada resulta casi imposible:

- Los controladores de DMA requieren de memoria (física) contigua para poder leer/escribir masivamente datos desde/hacia los dispositivos de almacenamiento. Si se utilizara paginación, tal vez se necesite desfragmentar la memoria para construir huecos lo suficientemente grandes (la gestión se complica).

- Cada vez que se hace una modificación en la tabla de paginación activa, la CPU vacía automáticamente la TLB. Hacer eso con la memoria del sistema o del núcleo penalizaría mucho el rendimiento.

- Utilizar un tamaño de página más grande (4Mb) para el núcleo y áreas reservadas aumenta mucho la eficiencia de la TLB, pero si se usa paginación se deberían utilizar páginas de 4Kb.

Por esta razón, se emplea un método muy conocido para hacer el seguimiento de la memoria libre: el *buddy system* (sistema de colegas). Básicamente la información sobre los bloques de memoria libre se guarda en una estructura de datos que puede verse como un arreglo de listas. Los bloques de memoria libre se agrupan en potencias de dos. Esto

es: la lista correspondiente al elemento 0 del arreglo contiene la información (básicamente ubicación) sobre huecos del tamaño de una página (es decir, es una lista enlazada de huecos del tamaño de una página); la lista en el elemento 1 describe todos los huecos de 2 páginas, la del elemento 2 describe los huecos de 4 páginas y así sucesivamente. GNU/Linux utiliza 11 potencias de dos, así los huecos más grandes tienen 2^{11} páginas.

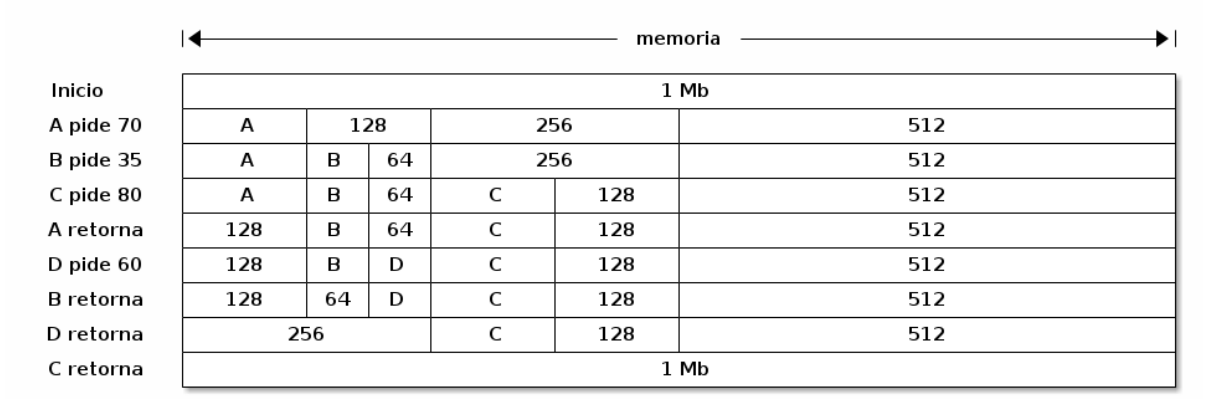


Figura 2: Comportamiento del buddy system ante distintos pedidos

La figura 2 muestra cómo evoluciona la reserva de memoria ante diversos pedidos: cuando se necesita asignar un bloque contiguo de memoria se busca el primer hueco lo suficientemente grande como para satisfacer el pedido (mejor ajuste). Si se necesita partir un hueco grande, se asigna lo necesario (removiendo ese hueco del arreglo) y el espacio restante se añade a las listas correspondientes.

Lo opuesto ocurre cuando se libera una porción de memoria: en este caso se añaden los huecos a las listas correspondientes. Pero esta vez hay que tener en cuenta una cosa: si en una lista quedan dos bloques "colegas" (vecinos en memoria y alineados al siguiente tamaño más grande), esos dos bloques pueden fundirse para formar un nuevo bloque más grande. De esta manera se soluciona el problema de la fragmentación externa.

2.1. Regiones más pequeñas

Dado que el buddy system permite administrar huecos de por lo menos una página, no puede utilizarse para reservar memoria en porciones más chicas. Sin embargo, esto es una tarea también bastante habitual. Por esto existe otra porción del manejo de memoria que se encarga de gestionar pedidos más chicos de memoria: el *slab* allocator. Esta gestión permite reducir la fragmentación interna y es también una porción muy complicada de software.

2.2. Asignación de memoria

La memoria RAM se asigna indistintamente a procesos y a caché. Todos puede crecer indiscriminadamente (si hay pocos procesos posiblemente haya mucha memoria utilizada para caché y si hay muchos procesos, buena parte de la memoria se utilizará para guardar información de los procesos y poca para los cachés).

3. Reclamo de páginas

¿Qué ocurre cuando la memoria se llena? En realidad un poco antes, el *algoritmo de reclamo de marcos del núcleo* (page frame reclaiming algorithm - PFRA -) se encarga de rellenar la lista de bloques libres "robando" marcos de procesos (en modo usuario) y de los cachés del núcleo. Esto se debe hacer antes de que la memoria se agote completamente pues para escribir los datos en disco (si hace falta) se necesitan también algunas páginas de memoria (p. ej.: para cabecera de los buffers a utilizar en E/S). O sea: el algoritmo de reclamo de marcos conserva un pool de marcos libres mínimo.

3.1. Tipos de marcos

El algoritmo maneja los marcos de diferente manera según su contenido:

Tipo	Descripción	Acción
No reclamable	Libres ó reservadas Memoria dinámica del núcleo Stack del núcleo de los procesos Bloqueadas temporalmente Bloqueadas en memoria	Ignorar
Swapeable	Anónimas en espacios de direcciones en modo usuario Páginas del *tmpfs* (p. ej.: páginas de memoria compartida para IPC)	Guardar el contenido en swap
Sincronizable	Mapeadas en espacios de direcciones en modo usuario Caché y contiene datos de archivos en disco Buffer de dispositivos de bloques Algunos cachés de disco (p. ej.: el caché de inodos)	Sincronizar con la imágen en disco (si es necesario)
Descartable	Páginas no usadas del caché de memoria Páginas no usadas del caché de direntries	No hace falta hacer nada

Veamos algunas distinciones:

- Hay páginas que pueden estar *bloqueadas en memoria*. Este bloqueo puede ser temporal (p. ej. si se está usando la página para realizar entradas/salida o en una sección crítica del núcleo) o permanente (p. ej. por pedido explícito del proceso).
- La páginas *anónimas* son aquellas que forman parte del espacio de memoria virtual de un proceso (por ejemplo, el segmento de código, datos, etc...) y no contienen datos provenientes de archivos (como ser archivos mapeados en memoria, librerías compartidas, etc...).
- Algunas páginas pueden contener archivos *mapeados en memoria*: estas páginas aparecen dentro del espacio de memoria virtual de un proceso pero sus datos provienen de un archivo.
- El sistema tiene cachés para distintos tipos de datos: direntries, inodos, datos de archivos, etc... Algunos tipos de datos se usan más frecuentemente que otros. Los

datos que están en caché podrían estar modificados en memoria y en ese caso, para liberar la memoria ocupada se deben escribir los datos en el lugar correspondiente en el disco (y puede involucrar actualizar otros datos).

3.2. Principios del PFRA

El algoritmo de reemplazo de marcos es una pieza de software compleja y no responde (al menos directamente) a ningún marco teórico: es más bien una serie de criterios que se han ido afinando en el tiempo, según parámetros y algoritmos más específicos para casos particulares.

No obstante, hay una serie de reglas que se aplican en general:

- Liberar primero las páginas que provoquen menos daño (p. ej.: la páginas de cachés).
- Todas las páginas de un proceso (salvo las bloqueadas en memoria) son reclamables.
- Antes de liberar una página compartida, se deben ajustar las entradas en las tablas de paginación de los procesos involucrados (veremos luego que esto se hace con *mapeo inverso*).
- Reclamar sólo páginas no utilizadas recientemente (usa una versión simplificada de LRU), utilizando el bit Accessed.
- En lo posible elegir las páginas limpias (no-dirty). Esto ahorra la escritura en el disco.

3.3. Cuando la memoria escasea

El reclamo de páginas se hace en forma periódica a través de un proceso llamado *kswapd*. También hay un mecanismo similar para recuperar memoria de los bufferes utilizados por el slab allocator.

Sin embargo, si falla algún pedido de memoria (low on memory) se disparan mecanismos más agresivos para recuperar la memoria que consisten principalmente en achicar el tamaño de los cachés.

Incluso, si la situación se torna muy crítica, el núcleo puede decidir eliminar un proceso (sin siquiera enviarlo a swap). De tomar la decisión se encarga una pieza especial del rompecabezas llamado el Out of Memory (OOM) Killer. Esta decisión no es nada sencilla: se intenta elegir un proceso bastante grande (para poder liberar bastante memoria), que no haya hecho demasiado progreso (para no desperdiciar trabajo) y que no esté involucrado en secciones críticas, entrada/salida o tareas del núcleo que puedan dejar el sistema inconsistente.

3.4. Algunas consideraciones sobre swap

Para realizar el intercambio de página de swap con el disco se utilizá un caché de swap: de esta forma, si se necesita una página que fue escrita en disco pero que todavía está en el caché no es necesario traerla nuevamente del disco. Sin embargo, también hay que tener en cuenta un par de problemas de concurrencia:

- *Multiple swap-in*: el núcleo debe tomar las medidas para que una página compartida no sea cargada dos veces en memoria (convirtiéndose en una página no compartida). Esto podría darse si el proceso A intenta acceder a una página compartida con el proceso B y que está en swap. En ese caso, lanzará el pedido para obtener tal página. Mientras se efectúa la lectura, puede ocurrir que el proceso B también intente acceder a esa página y el sistema operativo al notar que no está en memoria lance un nuevo pedido. Así, cuando el disco termine las dos lecturas cada proceso tendrá una copia distinta de la misma página que por lo tanto yo no estará compartida.

- *swap-in y swap-out concurrentes*: otra situación nefasta puede darse cuando el sistema decide enviar una página a swap. Si antes de que se efectivice la escritura el proceso intenta utilizar esa página y detecta que no está en memoria podría traer del disco una versión anterior de la página.

A. Apéndice: Demostración con programa que usa mucha memoria

A continuación se muestra una demostración de un programa grande que usa mucha memoria virtual:

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

int m[1000*1000*1000];

int main(){
    long i;
    printf("Arranqué, durmiendo, pid: %d\n", getpid());
    sleep(20);
    printf("llenando matriz\n");
    for (i=0;i<1000*1000*1000;i++){
      m[i]=i;
    }
    printf("durmiendo\n");
    sleep(20);
    printf("Pidiendo memoria\n");
    char *a = (char *) malloc(1000*1000*1000);
    printf("malloc retorno: %p, durmiendo\n", a);
    sleep(20);
    printf("Usando memoria\n");
    for (i=0;i<1000*1000*1000;i++){
      a[i]=i;
    }
    printf("Fin, durmiendo\n");
    sleep(20);
    return 0;
}
```

8 Sistemas de archivos

Un sistema de archivos (File System) sirve para gestionar informacion persistente. Deber permitir:

- Guardar informacion
- Consultarla
- Actualizarla
- Eliminarla

Tipicamente se usan medios magnéticos. Pueden ser cintas, discos, etc.
Todo sistema de archivos contara con dos cosas centrales:

- La información que tiene que guardar (data)
- La información extra para la gestión (metadata)

Veamos algunos sistemas de archivos

8.1 TAR (UNIX)

Usa un sistema de archivos con esta estructura:

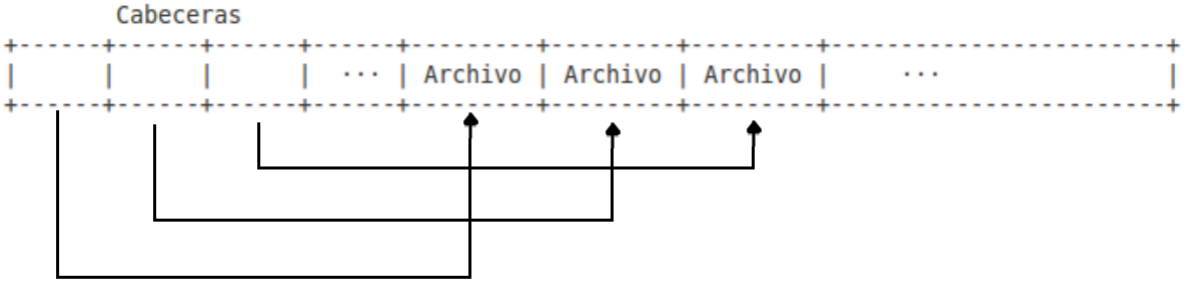
```
          cabecera                  cabecera
          ------------------------------------------------------------------------------------
          |512 B  |   archivo   |*| 512 B   |   archivo   |*|            ...               |
          ------------------------------------------------------------------------------------
*= Relleno hasta un multiplo de 512
la cabecera contiene: Nombre, permisos, tamaño, etc
```

- Ventajas
  - Simple
- Desventajas
  - Acceso secuencial
  - Actualizacion compleja
  - Los archivos están en un sub bloque

Sirve para ocasinoes en que actualizacion, etc. no sea necesaria.

8.2 Concentrar la metadata (Commodore 64)

Podemos saber qué cabeceras están libres con un valor en particular.



La 1° cabecera apunta al 1° archivo, la 2° al segundo, etc

- Ventajas
  - Simple
  - Es más rápido saber si un archivo está o no
- Desventajas
  - Actualizacion compleja
  - Tamaño fijo del sector de cabeceras.
  - Los archivos están en un sub bloque

8.3 Questar/MFS

Trató de permitir la fragmentación de archivos


```
*= 1 bit por sector de 512 e indicaba si estaba ocupado
**= cada Dir Entry tenia
+------+-----------------+-------+------------+----------------+------------+------------+---+
|nombre|     en ASCI     | largo | 1er sector | nro de sectores| 2er sector | 3er sector |...|
+------+-----------------+-------+------------+----------------+------------+------------+---+
|  7+3 | clave de acceso |       |                      Fragmentos                            |
```

Problema: Fragmentacion acotada.

NOTA: Un subdirectorio es simplemente un archivo solo que sus fragmentos son dir entries.

8.4 MSDOS FS

A principio de los 80 MICROSOFT comienza a usar este File System. Se basa en unificar dos aspectos a manejar:

- Gestion del espacio libre
- Permitir y gestionar la fragmentacion arbitraria.

Hicieron esto con una estructura llamada FAT (File Allocation Table). El medio magnético aparece asi:
(MSDOS en lugar de sectores de 512 bytes usa clusters de 4 sectores fisicos contiguos)

```
arranque        1era FAT        2da FAT         directorio raiz (catalogo raiz)         resto del disco
=============================================================================================================
|       |                 |           |      |       |      | ... | |         |                              |
=============================================================================================================
                                                dir. entries

cada dir. entry tiene
=============================================================================================
| nombre (2 B) | ext (3 B) | atributos (oculto, etc) | creacion | modificacion | nro de cluster |
=============================================================================================
                                                |          fechas         |     inicio     |
```

La FAT es un arreglo de enteros de 24 bits para los disquettes; luego se estira a 32 para los primeros rígidos. A cada cluster le corresponde un entero, en el mismo orden.

`cluster 0 <---> entero 0`

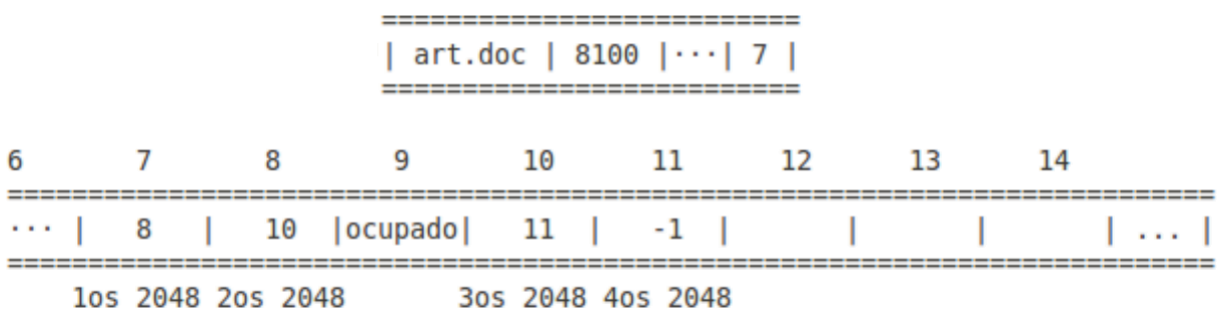
`clsuter 1 <---> entero 1`

...

El valor del entero determina el estado del cluster correspondiente:

- 0: cluster libre
- fffffe: cluster dañado (seteado durante el formateo)
- El valor de un cluster usado por un archivo es ffffff si es el último; caso contrario el número de cluster siguiente.

8.4.1 Ejemplo - Un archivo con 8100 btes de largo

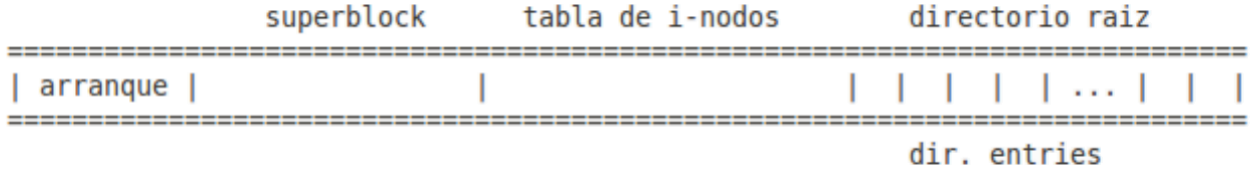


La FAT permite representar la ubicación de los clusters de un archivo como una lista enlazada.

- Ventajas
  - Es un buen FS
- Desventajas
  - Es un buen FS para discos chicos. Si la FAT no cabe en memoria hay varias lecturas extras para acceder a un archivo
  - La politica de actualizacion sincrona de la FAT lentificaba la escritura
  - The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 20-GB disk and a 1-KB block size, the table needs 20 million entries, one for each of the 20 million disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 60 MB or 80 MB of main memory all the time, depending on whether the system is optimized for space or time. Conceivably the table could be put in pageable memory, but it would still occupy a great deal of virtual memory and disk space as well as generating extra paging traffic.

8.5 UNIX FS

Hecho a principios de los 70. Es resumen del System7. El disco se estructura asi:

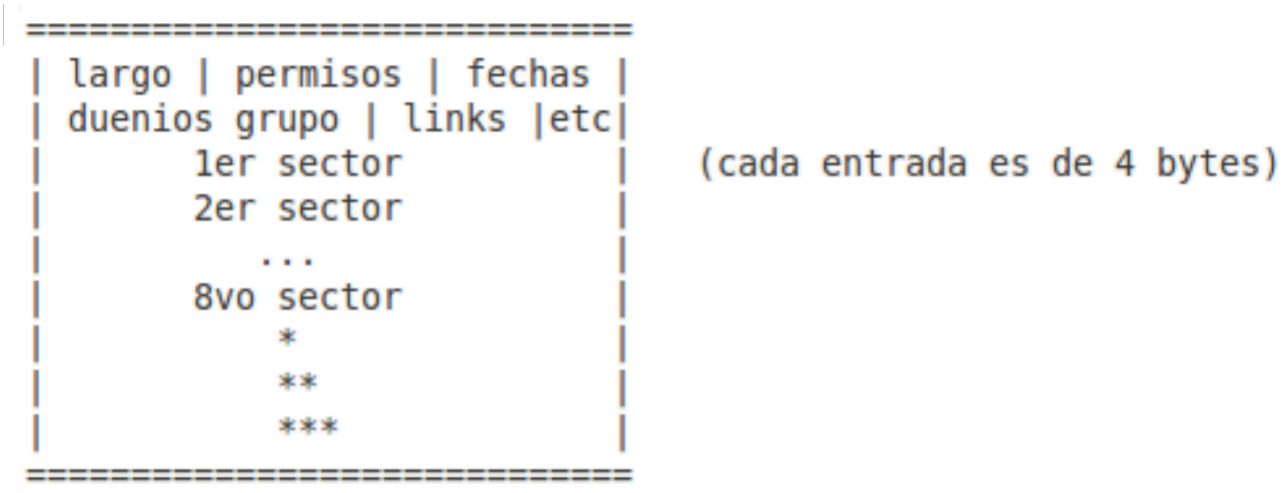


El superblock tiene:

- Un mapa de bits para sectores libres y ocupados
- Una lista enlazada con los i-nodes libres.

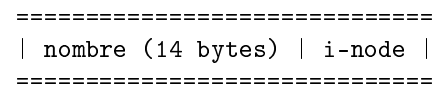
- The first one is the superblock. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file system type, the number of blocks in the file system, and other key administrative information.

Un i-nodo (information node)
Sector logico (mínimo tamaño que toma el sistema operativo): 512 bytes (coincide con el físico) tiene



en * habia una dirección a una tabla con 128 entradas de 4 bytes que indicaban dónde estaban los sectores 9, 10...etc.
** se conocia como 2 indirecciones y basicamente apuntaba a una tabla como a la que apuntaba * pero en vez de que sus entradas apuntaran a sectores, sus entradas apuntaban a una tabla del tipo de *. Es decir que tenia 128^2 sectores de informacion.
Te animas a buscar como seria el *** (3 indirecciones)

Cada dir. entry era



Un archivo puede tener varios nombres (links). Solo necesitan estar apuntando al mismo i-node. De este modo para borrar un archivo es borrar un link y decrementar el numero de links. Cuando este llega a 0 puedo borrar el archivo

- Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an i-node (index-node), which lists the attributes and disk addresses of the files blocks. A simple example is depicted in Fig. 6-15. Given the i-node, it is then possible to find all the blocks of the file. The big advantage of this scheme over linked files using an in-memory table is that the i-node need only be in memory when the corresponding file is open. If each i-node occupies n bytes and a maximum of k files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only kn bytes. Only this much space need be reserved in advance.

- One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address of a block containing more disk block addresses, as shown in Fig. 6-15. Even more advanced would be two or more such blocks containing disk addresses or even disk blocks pointing to other disk blocks full of addresses. We will come back to i-nodes when studying UNIX later.

8.6 Mejorando el FS

Queremos mejorar el Unix FS. Con mejorar nos referminos a varias posibles lineas: seguridad, rapidez, etc.
Probemos con rapidez.

Para eso iremos al estado de California, más exactamente a la universidad de Berkeley, Oakland. BSD modifica el UNIX File System consiguiendo incrementos de velocidad de aproximadamente 80%

Como lo lograron? Hicieron dos innovaciones: aumentaron el tamaño del sector lógico, desarrollaron el algoritmo del ascensor

8.6.1 Hacer crecer el sector lógico

Un sector lógico distinto del físico. El sector logico de BSD era de 4096 bytes en lugar de 512 bytes. Se leían 4096 bytes (8 sectores) porque aunque los programas leían de a 512, el sistema operativo pedía de a 8 de estos sectores. Entonces cuando el programa pedía los proximos 512, el sistema operativo ya los tenía en cache. Esto acelero un 20%

8.6.2 Algoritmo del ascensor

Teniendo en cuenta que un disco tiene dos tipos de movimientos

- Rotación de platos (angular)
- Cabezal (radial)

Puedo optimizarlos. Se puede acelerar el acceso a un disco no poniendo dos sectores seguidos. Esto mejora el rendimiento angular [interleaping de sectores fisicos para minimizar la cantidad de revoluciones para leer una pista]

Si quiero leer 1 y 2 y están pegados quizás termino de leer 1 y quiero leer 2 pero me pase entonces tengo que esperar que el disco de toda la vuelta. En cambio si 2 no está pegado a 1 no tengo que esperar eso.

Para mejorar el rendimiento del movimiento radial se usa el ALGORITMO DEL ASCENSOR: colectar pedidos de lectura/escritura y reordenarlos de modo de no hacer mas de una o dos pasadas. Es óptimo excepto que requiere que los pedidos de lectura/escritura sean independientes.

Es decir, al inicio el cabezal está detenido. Al llegar la primer petición va una dirección. Cada pedido que llega se va cumpliendo en el orden en que aparecen en el recorrido del cabezal hasta el fnal del recorrido, luego se empieza el recorrido inverso y se responden los de esa dirección y así sucesivamente.

8.6.3 Distribución del Superblock

Qué ocurre si el sistema se cae en el medio de la creación de un directorio?

Queda un FS inconsistente que hay que devolver a un estado consistente.

Esto lo hace el fsck (File System Checker). Para esto es que las operaciones de creación deben ordenarse. Lo siguiente a mejorar consiste en distribuir el superblock. En vez de tenerlo al principio del disco lo distribuyo en partes en el disco.

Entonces una operación en vez de tener que ir al superblock al principio del disco solo tiene que ir al más cercano. Es además más seguro porque si se me caga en el que tiene el superblock todo al principio cago todo, pero en el otro si cago alguno se me caga un sector pero no todo el disco. Este sistema dio un 60% más de coso.

```
================================= ================================
|SB|             | |SB0|   |SB1|   |SB2|   | ... | ================================
==============================
Unix File System           FFS (Fast File System)
```

Esto aún se puede mejorar! En Paris ubicamos a Gelinas que hizo unos FS llamados ext y ext2. Saco las secuencias dependites (no las saca...las IGNORA!). Ahroa el fsck de los ext* es mucho más complicado.

8.7 Más mejoras

Tomemos la apertura de un archivo. El nucleo debe:

- buscar en el directorio correspondiente /* un directorio es un arreglo de dir entries O(n) */
- sacar su i-nodo de informacion.
- chequear los accesos.
- construir el descriptor en el System Segment

Podemos acelerar esto cambiando el arreglo por otra estructura. Por ejemplo, un árbol! Si está balanceado sale con O(ln N). Casos de estos son Red-Black, AVL y BTREE.

Los BTREE (Balanced Trees) fueron elaborados en 1971. "En vez de hacer crecer un arbol desde la raíz, lo hacemos crecer desde las hojas".

Podemos usar un BTREE para los directorios, idem para los sectores libes. (ordenado: por tamaño y ubicación)

El pionero en esto fue IRIX (el UNIX de SGI). Tener dos BTREES para los bloques libres es complicado y eso llevó a que IRIX tuviera que incluir journaling.

Qué es un journaling? Son acciones que se toman para, frente a un cambio de metadata permitir volver atrás cambios parciales y alcanzar un estado consistente. Es muy usado, por ejemplo en bases de datos. En esencia se guardan los valores intermedios de la matadata con un timestamp. Por supuesto si se hace 'commit' (se completa la transacción) se marca también.

Journaling baja la performance pero aumenta la seguridad.

FS con journaling: IRIX (xfs), IBM (jfs), ext3, ext4, ResierFS, etc.

Qué podemos mejorar? Arreglo-O(n) BTREE—O(log N) Hash—-O(1)

Para buscar un ejecutable tener una base de datos con las ubicaciones de los ejecutables y traerlos con una busqueda con hash. El que comenzo a usar esto con atributos de archivos fue MacOS (en Linux lo hace el shell). Esto tambien fue usado por BeFS (practical implementation of file system...me la corto si lo consigo) de BeOS usado por BeBox.

Podemos mejorar esto!? (campo experimental).

Podemos pedir:

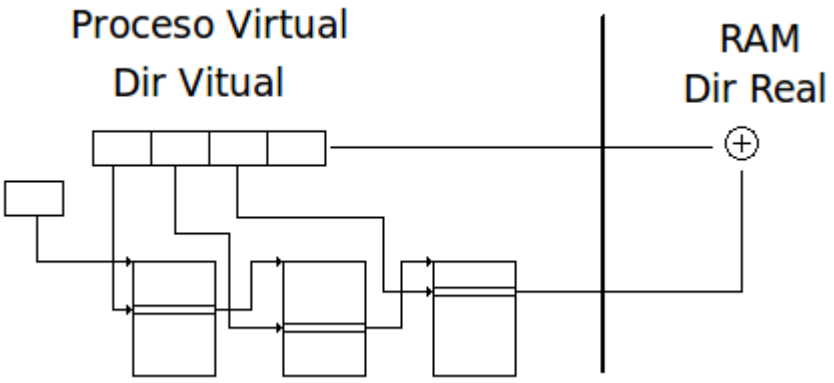
- Máxima velocidad
- Máxima seguridad

Son antagónicos pués mantener el log de transacciones para el journaling consume mucho tiempo. Para mejorar esto ponemos el FS en el LOG ahora cada vez que escribimos en el FS estamos escribiendo en el LOG también. Cada cambio se agrega en lugar libre con un timestamp.

Este FS se conoce como LogFS y fue una tesis de Ph.D. de Margo Seltzer en Berkeley. Es muy rápido pero tiene el inconveniente de derrochar mucho espacio (nunca pisa nada sino que va agregando) así que periódicamente hay que correr una aspiradora

9 Manejo de memoria

Usaremos sistemas con paginacion. (Sistemas Operativos Modernos Tanenbaum o Silberchautz)



- Todo proceso tiene sus tablas y su conjunto de páginas.

  Cuando referenciamos una página tenemos dos posibilidades

  - La página está en RAM (es un hit)
  - La página no está en RAM (es un miss)

- Para poder usar una página, esta debe estar en RAM.