File Systems

(Parte 1)

Índice

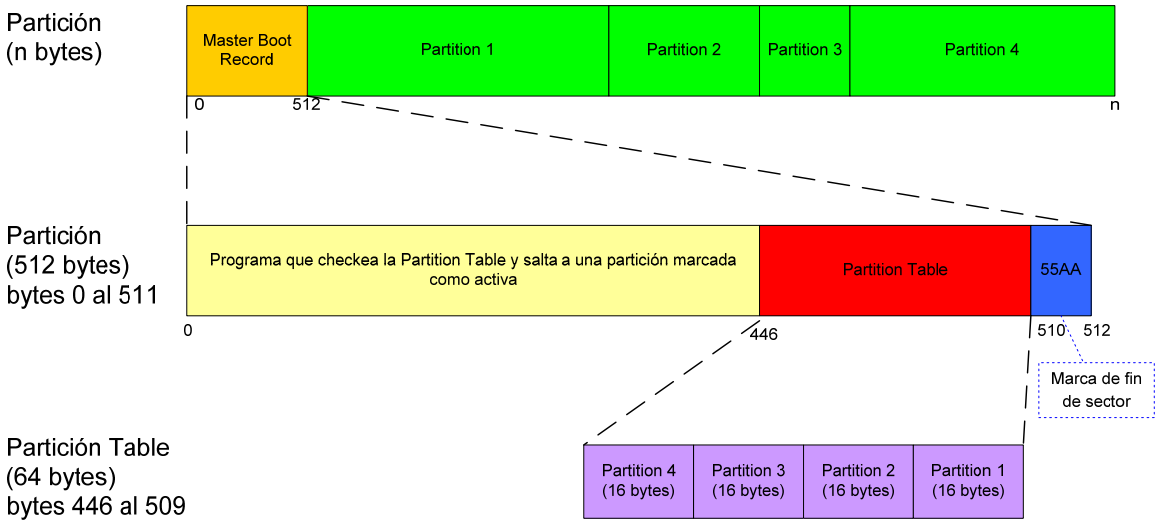
Generalidades

Antes de comenzar tenemos que definir 3 elementos que estarán presentes en todos los filesystem de los que vamos hablar:

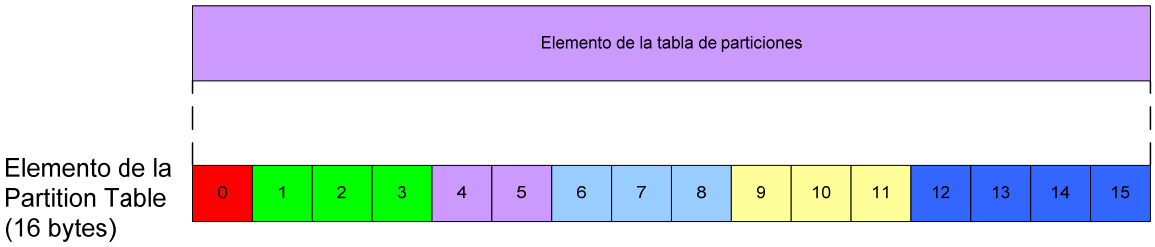
Archivo: Conjunto de información almacenada bajo un nombre.
Mínima unidad de información a la que puedo acceder de la memoria secundaria.
Partición: División lógica en un disco rígido, formada por una porción continua de un disco rígido.
Filesystem: Forma de organizar una partición.

Particiones y arranque

Cuando arranca la computadora automáticamente se busca el cylinder 0, head 0, sector 1 (primer sector del disco) del dispositivo que haya sido elegido para bootear.
En ese sector lo que vamos a encontrar es el master boot record (MBR) que contiene la descripción de las particiones que tiene el disco, así como también un pequeño programa de 446 bytes que verifica la integridad de la Partition Table y hace que el sistema "salte" a la partición correspondiente para poder iniciar el sistema operativo.
Si quisiéramos instalar algún boot manager como lilo, grub o el administrador de arranque de Windows NT 4.0/5.x, este es el lugar indicado.
Veamos un poco como se organiza este primer sector:



Para poder saber a dónde debe saltar el programa, es que se consulta la tabla de particiones. En cada elemento de la tabla vamos a encontrar la siguiente información:



byte	Descripción
0	Boot indicator. 80h = Bootable 00h = No booteable
1-3	Dirección de inicio de la partición (CHS)*
4-5	Indicador de tipo de sistema de Archivos***
6-9	Dirección de Fin de la Partición (CHS)*
10-12	Sector de comienzo (relativo a la partición)
13-15	Tamaño de la partición (Expresado en # de Sectores)

*CHS = 3 bytes que indican la cabeza(H), el sector(S) y el cilindro(C). Esta es la forma en la que se representa si bien hoy en día este tipo de direccionamiento no se utiliza más. Actualmente se utiliza un direccionamiento llamado LBA (Logical Block Addressing) que supera la limitación de 8,4GB de direccionamiento.

**Son valores válidos los siguientes 01h FAT12; 04h FAT16; 83h Ext2; 84h Linux Swap; etc. Una lista completa se encuentra en: http://www.win.tue.nl/~aeb/partitions/partition_types-1.html

FileSystems

Un filesystem es una forma de organizar los datos a guardar, hay muchos tipos de filesystem distintos y cada uno de ellos propone una solución a una necesidad de almacenamiento. Los filesystem que vamos a tratar en la materia son de propósito general y corresponden a los más ampliamente difundidos. Igualmente no debe perderse de vista que hay muchísimos filesystem que fueron diseñados para usos específicos como por ejemplo para sistemas de procesamiento y almacenamiento distribuido o para multimedia y juegos, los cuales resultan imprescindibles para que el sistema operativo pueda cumplir con su propósito. Por ejemplo imposible imaginar a google sin el gfs que permite que las miles de computadoras de procesamiento distribuido de google.com realicen su tarea buscando la información en alguno de los discos que cada na de ellas tiene o en un NAS (network area storage).

*Sistemas de Archivos en *nix*

Consideraciones generales

Cuado hablamos de los sistemas de archivos en un entorno *nix, estamos hablando de los sistemas de archivos que se implementaron para sistemas Unix y sus derivados: Linux, Aix, Minix, Solaris etc.
Si bien los filesystem de *nix pueden tranquilamente utilizarse en otros Sistemas operativos, es con ellos que se introducen conceptos que difieren mucho de otros filesystem como NTFS, o FAT y es por eso que antes de comenzar a explicar la estructura física se detallarán las estructuras lógicas y de administración de los filesystems en un entorno *nix.

El Buffer Cache

El Kernel (núcleo del sistema operativo) podría leer y escribir directamente en el disco rígido cada vez que se hace un acceso al sistema de archivos, pero el tiempo de respuesta sería muy pobre debido a que los accesos a disco son muy lentos, así como también la velocidad de transferencia de la información almacenada en ellos. Tengamos en mente que la velocidad de respuesta de una memoria está en el orden de los nanosegundos (10^{-9}) mientras su equivalente en un disco rígido es del orden de los milisegundos (10^{-3}).

Es por este motivo que el Kernel utiliza una serie de buffers para minimizar la cantidad de accesos a disco, llamado buffer cache, que contiene los bloques de datos más recientemente utilizados, para de esta forma minimizar las lecturas y escritura en disco.

Cuando el Kernel deba acceder a un bloque de datos, lo que va a hacer es fijarse si el bloque se encuentra ya en el buffer cache[1]. Si está, se ahorro una lectura, y si no está, entonces accede a disco y leer la información. Esta información queda luego en memoria como parte del buffer cache para futuras lecturas.
De esta forma, se logra reducir la cantidad de accesos a Disco, ya que buena parte de las operaciones se van a hacer directamente en memoria.

Por otra parte, la memoria RAM no es infinita, con lo cual el buffer cache, deberá también disponer de un algoritmo que decida qué datos deben mantenerse en memoria y cuales ya pueden ser escritos en disco. Esto último algoritmo también es importante en el eventual caso en el cual el sistema fallara por un corte de luz u otro problema, con lo cual, regularmente se deberían escribir los cambios en disco.

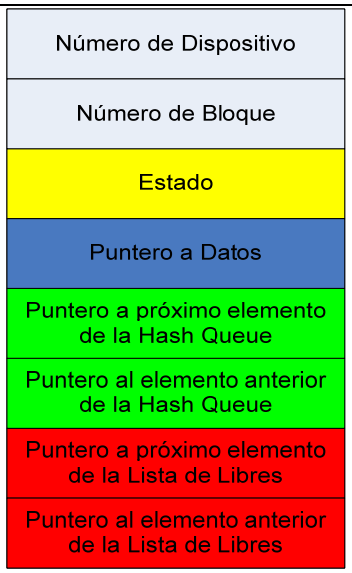
Estructuras del Buffer Cache

Headers

Un buffer consta de dos partes: los datos (una copia de la información tal cual y como está grabada en el disco) y el header (que permite identificar al buffer).

[1] El buffer Cache es una estructura de datos que no debe ser confundida con los cache de Hardware de por ejemplo una grabadora de CD o un Microprocesador.

Un header tiene la siguiente estructura:

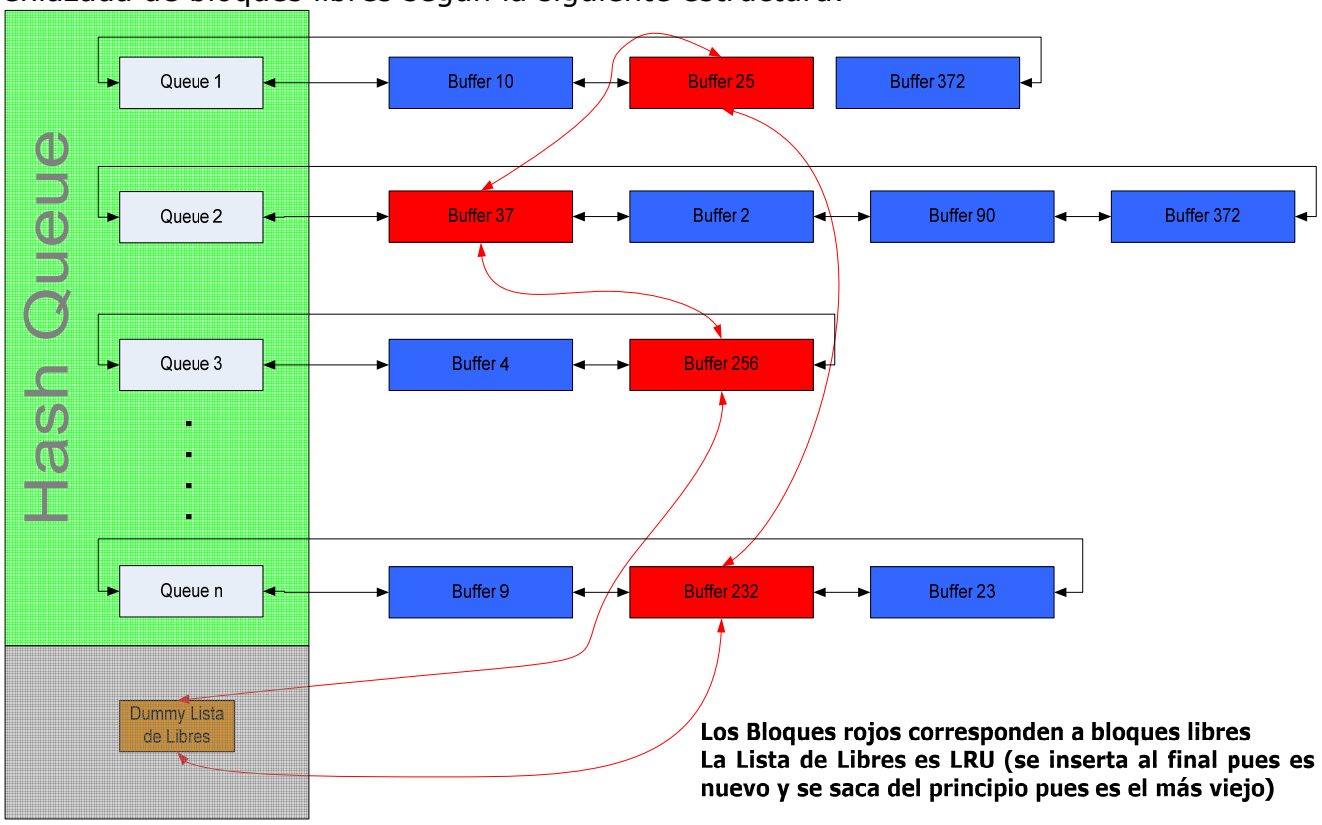
Número de dispositivo lógico Número de bloque de datos Estado del buffer Puntero al área de datos Puntero al próximo en la hash queue Puntero al anterior en la hash queue Puntero al próximo en la lista de libres Puntero al anterior en la lista de libres	

El estado del buffer es una combinación de los siguientes condiciones:
(Flags)

Locked/Busy (Bloqueado/ocupado)
Valid (Valid)
Delayed Write (Escritura demorada)
On Demand (a Demanda)
Bloqued by Kernel (Bloquedado por el Kernel)

Listas

El Buffer Cache está formado por N Hash Queues y una lista doblemente enlazada de bloques libres según la siguiente estructura:



Los Bloques rojos corresponden a bloques libres
La Lista de Libres es LRU (se inserta al final pues es nuevo y se saca del principio pues es el más viejo)

Cuando arranca el Sistema, se crea una cierta cantidad de buffers para el buffer cache, según parámetros de diseño del sistema operativo y todos los buffers del Buffer Cache son puestos en la lista de libres. Los Bloques de disco que se mantienen en el Buffer Cache responden a la política LRU (Least Recently Used), lo que quiere decir que se mantendrán en memoria los N últimos bloques que hayan sido utilizados. Cada vez que el Kernel necesita un buffer libre, consulta esta lista y toma el primer buffer libre de la misma. Esta lista es una lista doblemente enlazada que contiene un buffer dummy para marcar el comienzo y el fin de la misma.

También tiene organizados los buffers en una hash queue. Es decir, cuando el Kernel necesita acceder aun buffer en particular lo identifica por número de device lógico y número de bloque. En vez de buscar en todo el buffer cache si el bloque se encuentra en memoria, hashea estos números accediendo luego a la hash queue donde se espera encontrar al buffer buscado. Existen varias posibilidades de aquí en mas, dependiendo si encuentra o no el buffer en la cola a la que accede.

Cada buffer siempre existe en una hash queue (no existen los buffers "sueltos"), sin tener importancia del orden que tengan dentro de cada una. Dos buffers no pueden contener al mismo tiempo los datos del mismo bloque de disco. Sin embargo un mismo buffer puede estar en una hash queue y en la lista de libres.

Utilización del Buffer Cache

getblk

Es el algoritmo utilizado para hacer una búsqueda dentro del Buffer Cache.
Cuando se realiza una búsqueda dentro del buffer cache, pueden presentarse los siguientes casos.

1. El buffer está en su Hash Queue y el estado es libre.
El Kernel intenta buscar un bloque en el buffer cache, para eso hashea el device number y el block number para poder determinar en qué hash queue debería encontrarse el bloque. Una vez hecho esto, comienza a buscar en la Queue correspondiente hasta que encuentra un buffer con el número de device y bloque buscados, el cual está marcado como libre (pertenece también a la lista de libres). En este caso se marca el buffer como ocupado y se lo quita de la lista de libres. Finalmente se devuelve el bloque al proceso que lo solicitó.

2. El buffer no está en su Hash Queue.
El Kernel intenta buscar un bloque en el buffer cache, para eso hashea el device number y el block number para poder determinar en qué hash queue debería encontrarse el bloque. Una vez hecho esto, comienza a buscar en la Queue correspondiente sin suerte, ya que no encuentra un buffer con el número de device y bloque buscados. Dado que no lo encontró, toma un buffer de la lista de libres (el último, pues el criterio del buffer cache es least reacently used o LRU) le graba en el header los datos de device y bloque y lo mueve a la Hash Queue correspondiente. Luego de moverlo, lo retira de la lista de libres y lo marca como Ocupado. Finalmente devuelve el bloque al proceso que lo solicitó.

3. El buffer no está en su Hash Queue y el buffer de la lista de libres está marcado como "Delay Write".

El Kernel intenta buscar un bloque en el buffer cache, para eso hashea el device number y el block number para poder determinar en qué hash queue debería encontrarse el bloque. Una vez hecho esto, comienza a buscar en la Queue correspondiente sin suerte, ya que no encuentra un buffer con el número de device y bloque buscados. Dado que no lo encontró, toma un buffer de la lista de libres, pero esta vez el buffer de está marcado como "Delay Write".
Esta Marca quiere decir que el contenido de ese bloque debe ser bajado a disco antes de eliminarse. El motivo por el cual puede aparecer un "Delay Write" se verá más adelante. Para escribir el bloque en disco, se pide una escritura asincrónica, y se toma el siguiente bloque de la lista de libres. En caso de que este bloque también tenga la marca de "Delay Write", se solicita la escritura asincrónica y se continúa con el siguiente en forma sucesiva. Cuando finaliza la escritura, el buffer libre se coloca al principio de la lista de libres nuevamente. Esto se debe hacer para evitar que el buffer de una vuelta innecesaria en la lista antes de volver a ser utilizado. Ni bien se obtiene un buffer libre que pueda ser utilizado, se le graba en el header los datos de device y bloque y se lo mueve a la Hash Queue correspondiente. Luego de moverlo, se lo retira de la lista de libres y se lo marca como Ocupado. Finalmente el Kernel devuelve el bloque al proceso que lo solicitó.

4. El buffer no está en su Hash Queue y no quedan buffers libres.

El Kernel intenta buscar un bloque en el buffer cache, para eso hashea el device number y el block number para poder determinar en qué hash queue debería encontrarse el bloque. Una vez hecho esto, comienza a buscar en la Queue correspondiente sin suerte, ya que la lista de libres está vacía. En este caso el proceso que pidió el bloque se duerme hasta que algún otro proceso libere un buffer con la primitiva brelse que veremos más adelante. Recién cuando se ejecute esta primitiva, el kernel despertará a los procesos dormidos para que vuelvan a intentar obtener el bloque solicitado. Cuando se despierta el proceso, este debe volver a buscar en la Hash queue, pues existe la posibilidad de que otro proceso dormido hubiera accedido al bloque que solicito nuestro proceso. En ese caso puede pasarse a cualquiera de los otros escenarios dispuestos aquí.

5. El buffer está en su Hash Queue y el estado es ocupado.

El Kernel intenta buscar un bloque en el buffer cache, para eso hashea el device number y el block number para poder determinar en qué hash queue debería encontrarse el bloque. Una vez hecho esto, comienza a buscar en la Queue correspondiente hasta que encuentra un buffer con el número de device y bloque buscados, el cual está marcado como "busy". En este caso nuestro proceso B marca el bloque como "in demand" y se duerme, esperando que el proceso A que ocupó el bloque en primera instancia termine de utilizarlo. Cuando el Proceso B despierta, debe verificar que el bloque que solicitó esté libre y además que ningún otro proceso C haya estado esperando el mismo buffer y haya bloqueado el buffer al despertar antes que el proceso B. Además el proceso B debe verificar que el buffer contenga el bloque que él solicitó, ya que el proceso C podría haber tomado el buffer y cambiado el contenido por el de otro bloque. En este caso, el proceso B debe volver a comenzar con el algoritmo. Finalmente el Proceso B v a a poder encontrar el bloque que busca (probablemente utilizando un buffer de la lista de libres).
Es importante que un algoritmo de obtención de buffers sea seguro y que no ponga a dormir indefinidamente a los procesos, así mismo deben los procesos

poder encontrar un buffer aunque más no sea luego de un par de vueltas dentro del algoritmo.

```
Algoritmo getblk /* get block */
Input:  número de bloque del filesystem y device.
Output: buffer bloqueado que puede ser utilizado para el bloque
{
       While (no se encuentra el buffer)
       {
              If(el bloque está en su hash queue)
              {
                     If(buffer ocupado) /* Escenario 5 */
                     {
                            Sleep (hasta que el buffer esté libre);
                            Continue; /* Volver al while */
                     }
                     Marcar el buffer como ocupado; /* Escenario 1 */
                     Quitar el buffer de la lista de libres;
                     Return buffer;

              }
              Else /* el bloque NO está en su Hash Queue */
              {
                     If( no hay buffers en la lista de libres) /* Escenario 4 */
                     {
                            Sleep(hasta que algún buffer esté libre)
                            Continue; /* Volver al while */

                     }
                     Quitar el buffer de la lista de libres;
                     If( si el buffer está como "Delayed Write") /* Escenario 3 */
                     {
                            Escribir el buffer asincrónicamente a disco;
                            Continue; /* Volver al while */
                     }
                     /* Escenario 2 – es encontró un buffer libre */
                     Quitar el buffer de la lista de libres;
                     Poner el buffer en SU hash queue;
                     Return buffer;
              }
       }
}
```

Lectura y Escritura de Bloque en disco

Para la lectura y escritura en el disco se utiliza el algoritmo *getblk*, que permite buscar en el buffer cache el bloque y ahorrar un acceso a disco. Si el *getblk* no encuentra el bloque, deberá solicitar al driver de disco correspondiente la lectura del bloque, para transferirlo a la memoria.
El algoritmo que realiza esta lectura es el *bread.* Solicita la lectura en disco en caso de que el *getblk* no haya encontrado el bloque en el buffer cache con datos válidos. El algoritmo bread es el siguiente:

```
Algoritmo bread /* block read */
Input:  número de bloque del filesystem y device.
Output: buffer conteniendo los datos solicitados
{
       Obtener un buffer para el bloque (getblk)
       If (el buffer contiene datos válidos)
              Return buffer;
       Iniciar lectura desde disco;
       Sleep(hasta que se complete la lectura desde disco);
       Marcar el buffer como válido;
       Return buffer;
}
```

También existe el algoritmo *breada*, que intenta optimizar las lecturas de bloques continuos en disco. Lo que hace el *breada* es, además de leer el bloque solicitado, verificar si el siguiente bloque ya se encuentra en el buffer cache, y en caso de que no lo encuentre, iniciar la lectura en forma asincrónica, para de esta forma precargar el bloque en el buffer caché y dejarlo disponible para que el proceso que lo solicitó . De esta forma se mejoran por ejemplo las lecturas de un archivo secuencial, pues el algoritmo permite anticiparse a las necesidades del proceso.

Para poder escribir los datos de un bloque a disco vamos a utilizar otra función que se llama *bwrite.* Al llamar a estar función, se le pide al Kernel que informe al driver correspondiente que tiene un buffer que quiere grabar a disco. De esta forma, el driver va a programar la operación de I/O y surge la posibilidad de que la escritura sea sincrónica o asincrónica. En caso de ser sincrónica, el proceso que llamó a esta escritura deberá esperar hasta que los datos estén correctamente guardados en disco, mientras que en una escritura asincrónica, el proceso recupera el control, mientras el kernel inicia la escritura. Es el kernel quien luego de finalizar la operación liberará el buffer definitivamente.
Si recordamos un poco el esquema antes visto, podremos observar que uno de los estados posibles de un buffer es "delayed write". Esta marca la coloca el kernel en los casos en los que libera el buffer pero no lo manda a escribir a disco (no inicia la operación de I/O), pero igualmente llama la función *brelse.* En este caso brelse lo que hace es liberar el buffer y marcarlo como "delayed write". Lo que se intenta el kernel con esta acción es que algún proceso (el mismo u otro) utilice la información contenida en el Buffer antes de que sea escrito a disco, para de esta forma ahorrar 2 operaciones de I/O (las de escritura y posterior lectura de los mismos datos). No hay que confundir escritura asincrónica con escritura demorada (delayed write). La asincrónica implica que el kernel manda a escribir efectivamente los datos a disco (hay operación de I/O), mientras que la escritura demorada simplemente deja a disponibilidad el buffer en cuestión e intenta prolongar su permanencia en memoria lo máximo posible para ahorrar una operación de I/O y ganar en performance.
En caso de haber escrito el buffer a disco, el kernel va a marcar el buffer como "old" para que al terminar la operación de I/O, el kernel a utilizando *brelse*, pueda colocarlo al principio de la lista de buffers libres.

La función *brelse* despierta a todos los procesos dormidos luego de actuar sobre el buffer que recibe. Si el buffer a liberar tiene datos válidos, entonces lo coloca al final de la lista de libres, en caso contrario lo coloca al principio.

Ventajas y desventajas del Buffer Cache

- El uso de buffers, permite al kernel abstraerse y operar siempre sobre buffers, sin importarte si los datos corresponden a un archivo, un inodo o un superblock. De esta forma, el diseño del sistema se vuelve más simple.

- El uso de buffer cache reduce la cantidad de operaciones de I/O, mejorando la performance general del sistema.

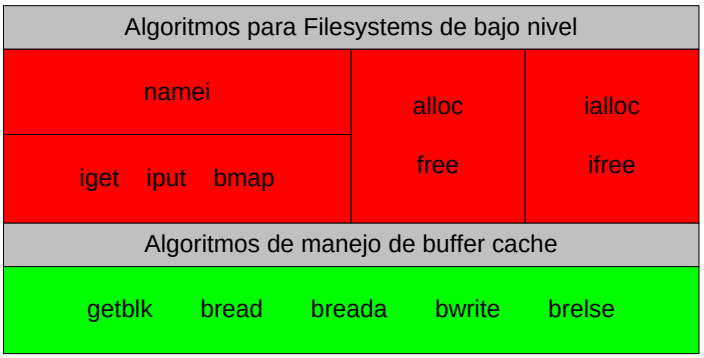
- Al proveer una única interface para acceder a los bloques del disco, se mantiene la integridad de los datos, ya que le buffer cache asegura que únicamente un proceso acceda a los datos por vez, serializando el acceso y previniendo la corrupción de datos.

- En caso de una falla del sistema, todas las copias de datos en memoria podrían perderse, aún cuando los procesos hubieran enviado correctamente la petición de escritura.

Representación interna de los datos

Hasta ahora nos ocupamos de la capa inferior de acceso a los datos. A partir de este momento comenzaremos a utilizar una capa intermedia que permite el acceso a la unidad básica de almacenamiento dentro de un filesystem, los llamados inodos.

El esquema de capas sería similar al siguiente:



Inodos

Cada archivo dentro del fs se identifica por medio de un inodo. El inodo contiene la siguiente información:

- Identificación del dueño (owner y group owner)
- Tipo de archivo (regular, directorio, FIFO, block, character)
- Permisos de acceso (owner, group, everyone else)
- Tiempos de acceso, modificación del inodo y el archivo
- Cantidad de links que tiene el archivo
- Punteros a los bloques de datos
- Tamaño del archivo

Para poder manipular la información del inodo, el kernel, debe subirlo a memoria (in-core inodo) y al hacerlo le agrega la siguiente información:

- Número de dispositivo lógico
- Número de inodo
- Estado del inodo
- Puntero al área de datos
- Puntero al próximo en la hash queue
- Puntero al anterior en la hash queue
- Puntero al próximo en la lista de libres
- Puntero al anterior en la lista de libres
- Contador de referencias

Si observamos bien, veremos que un inodo in-core es muy parecido a un buffer del buffer cache, y esto se debe a que el kernel realiza un tratamiento muy similar con los inodos al que realiza con los bloques en el buffer cache.

Los estados posibles para un inodo son:

- Bloqueado
- Un proceso se encuentra esperando que el inodo se libere
- El inodo in-core difiere de la copia en disco del inodo debido a un cambio en el inodo.
- El inodo in-core difiere de la copia en disco del inodo debido a un cambio en el archivo.
- El archivo es un punto de montaje.

La mayoría de los procesos de los inodo in-core son iguales a los de los buffers con algunas pequeñas diferencias. La mayor de las diferencias está en la aparición de un contador de referencias.

Este contador representa la cantidad de referencias activas al archivo. Una referencia activa podría sumar por ejemplo al abrir un archivo.

Por otra parte un inodo in-core estará dentro de la lista de libres únicamente cuando su contador de referencias sea cero.

Asignación de Inodos

Para poder cargar un inodo en memoria se utiliza el algoritmo *iget*. El algoritmo *iget* es muy similar al *getblk* del buffer cache.

```
Algoritmo iget /* inode get */
Input:  número de inodo del filesystem y device.
Output: inodo bloqueado
{
       While (no haya terminado)
       {
              If(el inodo está en el cache)
              {
                     If(inodo ocupado)
                     {
                            Sleep (hasta que el inodo esté libre);
                            Continue; /* Volver al while */
                     }
                     /* procesamiento especial para puntos de montaje */
                     if (inodo en lista de inodos libres)
                            Quitar el inodo de la lista de libres;
                     Incrementar el contador de referencias del inodo
                     Return inodo;

              }
              Else /* el inodo NO está en su Hash Queue */
              {
                     If( no hay inodos en la lista de libres)
                     {
                            Return (error)

                     }
                     Quitar el inodo  de la lista de libres;
                     Resetear el número de inodo y file system;
                     Poner el inodo en SU hash queue;
                     Leer el inodo del disco (algoritmo bread);
                     Inicializar el inodo (contador de referencias en 1);
                     Return inodo;
              }
       }
}
```

Para poder leer el inodo a memoria, debe leer el bloque de disco que contenga el inodo, esto lo calcula sabiendo cuantos inodos entran por bloque, y el numero de inodo que se quiere acceder (BREAD). Calcula el offset dentro del bloque y lee el inodo, incrementando su contador en uno.

Si no encuentra el inodo en la hash queue correspondiente y no hay inodos en la lista de inodos libres, entonces el algoritmo reporta un error.

Para liberar un inodo se utiliza el algoritmo IPUT. Este algoritmo decrementa el contador de referencias del inodo en 1, y si luego de esto el contador queda en 0, el Kernel escribe este inodo a disco (si es que fue modificado). Luego coloca el inodo en la lista de libres. Se entiende por modificación cualquier cambio en los datos, el tiempo de acceso, el owner, o los permisos de acceso.

El Kernel procederá a liberar los bloques de datos del archivo en caso de que la cantidad de links al archivo sea 0.

```
Algoritmo iput /* inode put */
Input: Puntero a un incore inode
Output: nada
{
	bloquear el inodo si no lo está;
	contador_de_referencias-- ;
	If(contador_de_referencias == 0)
	{
		If(contador de links ==0)
		{
			Liberar los bloques de disco del archivo (algoritmo free);
			Setear el tipo de archivo a 0;
			Liberdad el Inodo (algoritmo ifree);

		}
		If(se accedió al archivo ó cambió el inodo ó cambió el archivo)
		{
			Actualizar el inodo en Disco;
			}
		Colocar el inodo  en la lista de libres;
	}
	Quitar marca de Bloqueado del Inodo;
}
```

Estructura de un archivo

Como vimos hasta ahora, un inodo contiene la tabla de contenidos que permite ubicar los datos del archivo en el disco. Esta tabla estará formada por una serie de números de bloque que podrán ser accedidos directamente. Si un archivo ocupara una sección contigua de disco, con guardar el bloque de inicio y la longitud del archivo nos alcanzaría para obtener todo su contenido. Sin embargo, esto no es lo que ocurre habitualmente. Los archivos son guardados en porciones dispersas en el disco y es por eso que la estructura de los inodos debe adaptarse para permitir ubicar todos y cada uno de los fragmentos del archivo.

Para ello cada inodo contendrá un 13 punteros que nos permitirán acceder a los bloques de disco, según la siguiente estructura.

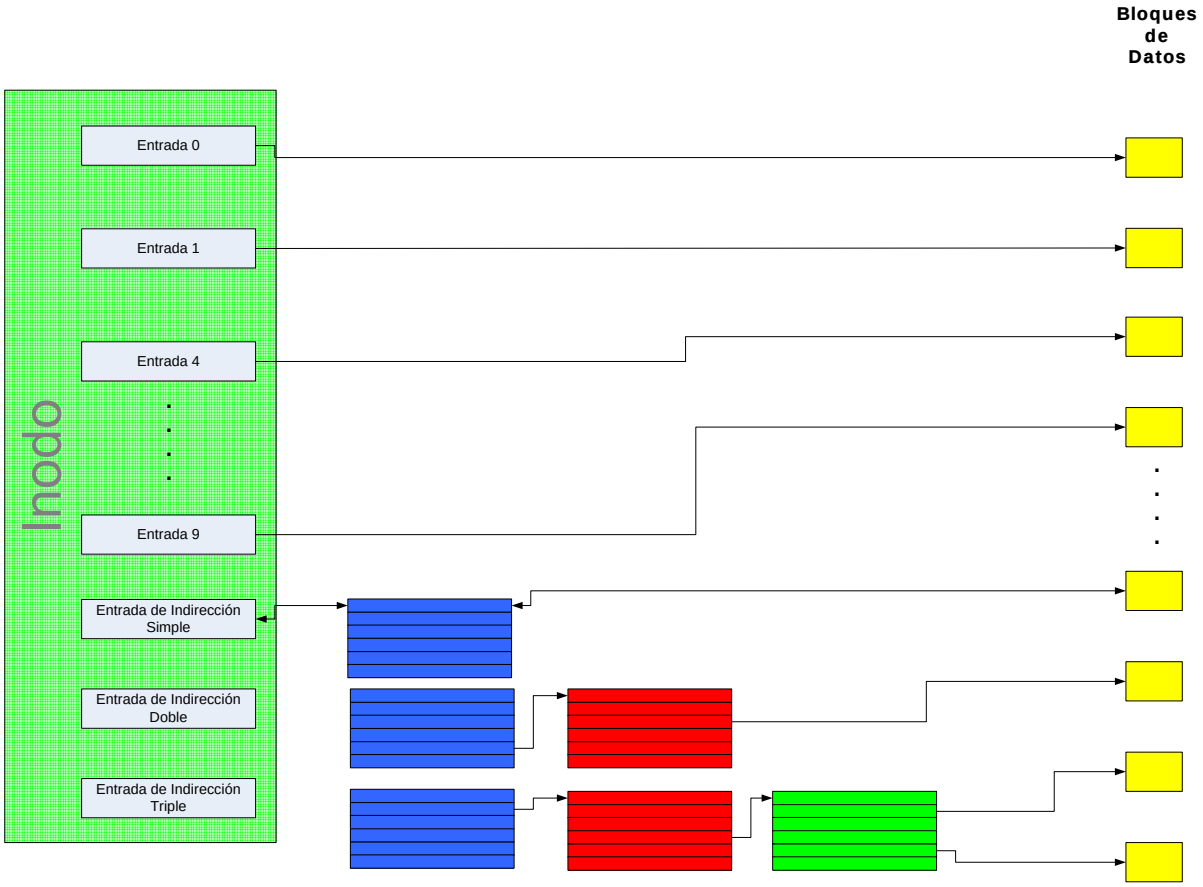
Las Primeras 10 entradas contendrán la dirección del bloque en el disco y permitirán acceder directamente a los datos. Es por ello que llamaremos a estas primeras 10 entradas, entradas de acceso directo.

La siguiente entrada (número 10) apunta a otro inodo, el cuál a su vez apunta a los datos en disco. Esta entrada se llamará de indirección simple. Pues tendremos que acceder a un inodo adicional para llegar a los datos en disco.

La entrada número 11 es de indirección doble ya que apuntará a un inodo que a su vez apuntará en cada una de sus entradas a otros tantos inodos, para recién desde allí apuntar a los bloques en disco. Estas entradas serán de indirección doble, ya que se deberán pasar por 2 inodos adicionales para llegar a los datos en disco.

Por último siguiente la misma lógica que hasta ahora llegaremos a la entrada de indirección triple. En este caso tendremos 3 niveles de indirección para llegar a los datos.

Veamos si queda más claro con un esquema.



En un principio todas las entradas contendrán un 0 sin ocupar ningún bloque de datos, ya que no contendrían información.

Este tipo de estructura ha resultado más que suficiente para los archivos utilizados en la actualidad, pero aún así resulta fácilmente extensible para soportar cuádruple o hasta quíntuple indirección.

Pero dado que los archivos no trabajan por bloques, sino por acceso directo a los bytes, esta estructura debe proveer algún algoritmo para traducir las indirecciones y bloques en los bytes solicitados por las funciones del programador. Aquí es dónde entra en juego al algoritmo *bmap*.

Supongamos que tenemos un filesystem de estas características:
Bloques de 1KB cada uno, números de bloques de 32 bits (entonces un bloque puede tener 256 entradas)
Entonces un inodo podría apuntar a:

10 bloques directos de 1KB	10KB
1 bloque de indirección simple con 256 bloques directos	256KB
1 bloque de indirección doble con 256 bloques indirectos	64 MB
1 bloque de indirección triple con 256 bloques de indirección doble	16GB

Igualmente el tamaño está limitado por la entrada en el inodo que marca el tamaño del archivo que es un entero de 32 Bits. Por eso un archivo no puede tener más de 4GB de tamaño en una arquitectura de 32 Bits.

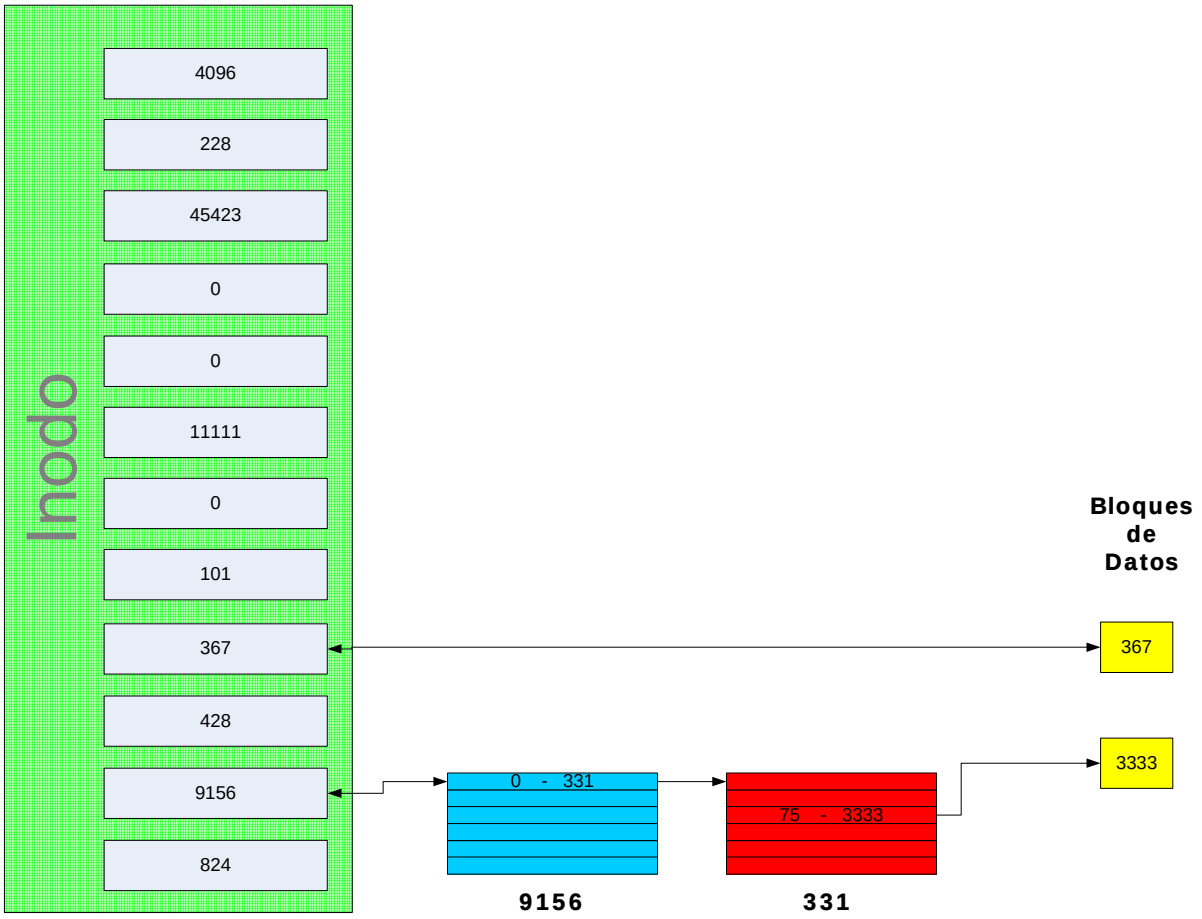
Bloques directos de 1KB	Bytes 0 a 10240
Bloques de indirección simple con 256 bloques directos	Bytes 10240 a 272384
Bloques de indirección doble con 256 bloques indirectos	Bytes 272384 a 67381248
Bloques de indirección triple con 256 bloques de indirección doble	Bytes 67381248 a 17247250432

Ejemplos:

Si quisiéramos acceder al byte 9000 de un archivo, deberemos calcular primero qué bloque debemos leer. El byte 9000 corresponde a la entrada 8 del inodo (9000 / 1KB = 8.7). Ahora deberemos buscar dentro del bloque (que tiene tamaño 1024KB) entonces calculamos nuevamente (9000 - 8*1024 = 808). El byte a leer será el byte 808 dentro del bloque apuntado por el inodo en la entrada 8.

Si quisiéramos leer el byte 350.000 del archivo. Vemos que cae dentro de una indirección doble (350.000/1024 = 341.79..). Al entrar en el inodo de indirección 2, comenzamos nuevamente con la búsqueda, pero el byte que buscaremos aquí dentro será el 77.616 (350.000 – 256k – 10k). Como cada bloque accede a un máximo de 256K, entonces 77.616 corresponde a la entrada 0 del bloque de indirección doble en el que estamos. Así al pasar de bloque calculamos que el bloque corresponde la entrada 75 (recordemos que hay 256 entradas que apuntan cada una a un bloque de 1kb en este inodo).

Veámoslo con un gráfico:



Directorios

Los directorios son los que le dan la forma jerárquica a un filesystem y son un tipo de archivo especial que guarda una secuencia de nombres de archivo y sus correspondientes inodos. La estructura según System V está formada por 2 bytes para el número de inodo y 14 bytes para el nombre de archivo, de esta forma cada entrada ocupará un total de 16 bytes.
La primer entrada corresponde siempre al inodo propio del directorio y se llama . (punto). La segunda entrada corresponde al inodo del directorio padre y es llamado .. (punto-punto).
El tratamiento que hace el Kernel de los directorios es exactamente igual que el que hace con cualquier otro archivo. La única diferencia surge en la parte d elos permisos, dónde lectura implica la posibilidad de leer el contenido del directorio, escritura implica crear o borrar una entrada de directorio y ejecutar implica buscar dentro del directorio. Veremos luego la diferencia entre estos 2 accesos.

Conversión de un Path a Inodo

Si bien cuando uno accede a un archivo, ingresa el path completo, el kernel trabaja internamente utilizando Números de Inodo. Es por ello que necesitamos un traductor que permita ir convirtiendo componente a componente, esa estructura jerárquica en número que el kernel pueda comprender. Para ello utilizaremos el algoritmo *Namei.*
Namei comienza recorriendo el path que le entregan (por ejemplo con un chdir) levantando a memoria subsecuentemente los inodos necesarios y procesando el directorio en forma secuencial hasta encontrar el componente buscado. Para cada paso en la búsqueda se verifica que el usuario que llamó el proceso tengas los permisos necesarios para realizar la operación y que el componente sea un directorio.

Ejemplo:

Pwd = /home/datos
$ Chdir /etc/apache

1. Primero se busca el inodo de /
2. Se carga el inodo de /
3. Se verifica que el usuario tenga acceso al directorio /
4. Se lee el directorio secuencialmente hasta encontrar una entrada para etc
5. Se busca y carga el inodo de etc y se libera el de /
6. Se verifica que el usuario tenga acceso al directorio /etc
7. Se lee el directorio secuencialmente hasta encontrar una entrada para apache
8. Se busca y carga el inodo de apache y se libera el de etc
9. Se verifica que el usuario tenga acceso al directorio /etc/apache

El superblock

El file system esta compuesto por:

- Boot Sector
- Superblock
- Lista de inodos
- Bloques del disco

El superblock contiene información del file system:

- El tamaño del fs.
- Cantidad de bloques libres en el fs
- Lista de bloques libres
- Índice del próximo bloque libre
- Cantidad de inodos libres en el fs
- Lista de inodos libres
- Índice del próximo inodo libre
- Estado del fs

Asignación de inodos a nuevos archivos

El algoritmo *ialloc* asigna un inodo del disco a un nuevo archivo. El superblock contiene una lista de inodos libres (Un inodo está libre si el tipo especificado es 0). Si el Superblock se queda sin inodos libres, le pide al kernel que busque en el disco más inodos libres y carga tantos como le sean posibles, para acelerar las próximas búsquedas de *ialloc*.

Si la lista tiene inodos, entonces toma el inodo, asigna una copia en memoria de este inodo por medio del algoritmo *iget*, inicializa el inodo y lo devuelve bloqueado (actualiza este inodo a disco).

Si la lista de inodos esta vacía, el Kernel busca en la lista de inodos del disco y guarda tantos inodos libres como sea posible en el array del superblock, recordando siempre cual es el mayor número de inodo almacenado.

De esta manera, la próxima vez que necesite levantar números de inodos del disco, sabe que a partir del número recordado tiene que empezar la búsqueda. Y cada vez que asigna un inodo a un archivo, decrementa la cantidad de inodos libres en el array dentro del superblock.

Cuando se libera un inodo se utiliza el algoritmo *ifree*. El Kernel chequea si hay lugar en la lista para el número de inodo liberado, si hay lugar entonces coloca el número en el array. Si no hay lugar, compara el número de inodo con el número del inodo recordado y solo si es menor, lo almacena dentro del array, sacando del array al inodo recordado. Este pasa a ser ahora el inodo recordado.

Asignación de bloques de discos

El superblock del File System contiene un array que es utilizado para cachear los números de bloques libres dentro del fs. El programa MKFS organiza los bloques del disco en una lista linkeada, tal que cada link de la lista es un bloque del disco y contiene un array de números de bloques libres, y una de las entradas del array es el número del próximo bloque linkeado del array.

Cuando el Kernel quiere asignar un bloque del file system, asigna el próximo bloque disponible en la lista del superblock (*alloc*). Una vez asignado, no puede ser reasignado hasta que sea liberado.

Si el bloque que lee de la lista es el último bloque libre, entonces toma el contenido, que es un puntero al próximo bloque del link y lee ese bloque, llenando con los datos del mismo la lista de bloques libres del array en el superblock.

Luego trabaja con el número de bloque que tenia libre y procede a asignarlo, luego de inicializar su contenido.

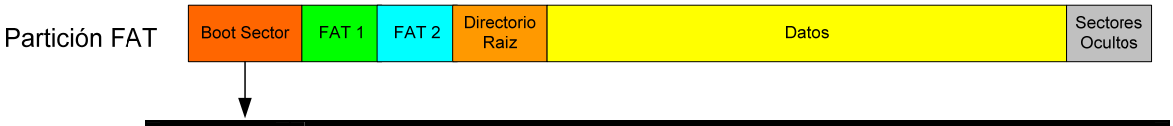
Cuando se ejecuta el MKFS, se van cargando los números de bloques de manera de que tengan cierta relación que luego pueda mejorar la lectura de los bloques asignados, para aquellos archivos que contendrán gran cantidad de bloques de datos.

El algoritmo *free* procede de manera inversa. Si la lista en el superbloque no esta llena, entonces el número del bloque liberado se carga en ella. Si esta llena, entonces el bloque liberado es un nuevo bloque de la lista linkeada, es decir, se copia en el bloque el array del superblock en el bloque y se graba a disco. El array del superblock queda con un elemento apuntando al bloque recién grabado.

Ejemplos de Filesystems

FAT[n]

FAT significa File Allocation Table y es el más simple y mayormente distribuido de los filesystems que vamos a tratar en la materia. Existen distintos tipos de especificaciones que varían de acuerdo a la cantidad de bits (n) que se utilizan para referenciar los sectores.



Byte	Descripción
0-2	Salto a la primera instrucción del programa de carga.
3-10	Nombre OEM
11-12	Bytes por sector
13	Sectores por Cluster
14-15	Reservado
16	Cantidad de copias de la FAT
17-18	Cantidad de entradas en el directorio Raíz
19-20	Cantidad total de Sectores en la partición
21	Media Descriptor (HDD, Floppy, etc.)
22-23	Cantidad de Sectores que ocupa la FAT
24-25	Cantidad de Sectores por Pista (Tracks)
26-27	Cantidad e Cabezas (Heads)
28-29	Cantidad de Sectores Ocultos
30-509	Rutinas de Booteo
510	55h
511	AAh

El boot sector difiere un poco entra la versión de FAT12/16 y la de FAT32. A fines prácticos vamos a tomar la definición de FAT12/16.

Tabla FAT

La tabla FAT la vamos a encontrar 2 veces dentro de un filesystem que utilice FAT. La segunda copia se utiliza en caso de que la primera copia de la FAT se encuentre dañada, para así poder restaurarla. El método con el que se graban ambas FAT fue cambiando con el tiempo, así como también en algunos casos su ubicación dentro de la partición.
Una tabla FAT crea la correspondencia entre un cluster y un archivo y a su vez encadena los sucesivos clusters al cluster original del archivo.

Entrada en la tabla	Valor
0	0000h
1	FFF7h
2	0023
…	
10	0000h
11	0000h
…	
20	0002h
…	
23	FFFFh

Son válidos para una Tabla FAT los siguientes valores:

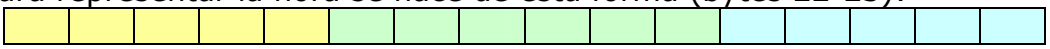
Valor	Significado
0000h	Cluster Libre
0002h-FFEFh	Próximo cluster asignado al mismo archivo.
FFF0h-FFF6h	Reservado
FFF7h	Cluster dañado
FFF8h-FFFFh	Ultimo cluster del Archivo (Marca de fin de archivo)

Directorios

Cada directorio es un archivo dentro del filesystem. Estos archivos contienen en su interior una serie de registros de 32 bytes llamados *entradas de directorio* que en el caso particular del filesystem FAT tiene la siguiente estructura:

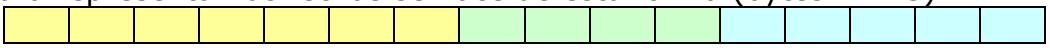
Byte	Descripción
0-7	Nombre del Archivo
8-10	Extensión del archivo
11	Atributos 0- read only 1- read only oculto 2- systemfile 3- Etiqueta de Volumen 4- Subdirectorio 5- Si fue modificado desde el último Backup 6- Sin uso 7- Sin uso
12-21	Reservados
22-23	Hora de Creación / Ultima modificación
24-25	Fecha de Creación / Ultima modificación
26-27	Dirección del primer cluster (Relativo al último cluster del directorio)
28-31	Tamaño del archivo expresado en bytes

Para representar la hora se hace de esta forma (bytes 22-23):



Hora (5 bits) Minutos (6 bits) Segundos (5 bits)

Para representar las fechas se hace de esta forma (bytes 24-25):



Año (7 bits) Mes (4 bits) Día (5 bits)

- La entradas vacías de completan con ceros
- La primer entrada en un directorio apunta siempre a si mismo (Directorio .)
- La segunda entrada en un directorio apunta siempre al directorio padre (Directorio ..)
- Al borrar una entrada de directorio, se pone E5h en el primer carácter del nombre y 0 en la entrada de la FAT correspondiente al primer cluster que utilizaba el archivo.
- Las búsquedas dentro del directorio son siempre secuenciales.
- El tamaño máximo de un archivo es de 2GB

VFAT

VFAT no es en si mismo un filesystem nuevo, en realidad está sobre la base del filesystem FAT, pero propone una serie de características nuevas.

Características nuevas aportadas por VFAT:

- Acepta nombres de archivo largo (máx. 256 caracteres)
- La posición del directorio raíz ya no es más fija en el disco rígido
- Guarda una tabla con la cantidad de espacio libre en el disco (al principio de la FAT)
- Implementa el concepto de Dirty Bit, que indica al Sistema operativo si el sistema de archivos terminó de hacer todas las operaciones de sincronización necesarias antes de desmontarse.
- FAT 1 y FAT 2 son actualizadas simultáneamente.
- Se guarda una copia del Boot sector en el sector 6 de la partición, cuestión de poder recuperarlo en caso de falla.
- Se pasan a codificar los caracteres utilizando UNICODE en lugar del tradicional código ASCII, con lo que se aumenta la compatibilidad principalmente con los idiomas asiáticos

Al introducir la posibilidad de utilizar nombre de archivos largos (a partir de ahora LFN), la estructura del filesystem debió ser modificada, pero dado que FAT es el filesystem más distribuido, la compatibilidad con versiones anteriores (backward compatibility) fue un punto muy importante al diseñar la nueva estructura. El cambio más significativo a la estructura de la FAT tradicional fue la inclusión de los nombre de archivo largo. La solución que se propuso es la siguiente:

Las entradas de directorio se modificaron para utilizar varias entradas para representar un nombre largo.

- Cada entrada de directorio de un LFN guarda 13 caracteres del nombre en codificación UNICODE
- Se alocan tantas entradas de directorio como sean necesarias antes de la entrada de directorio correspondiente al nombre en forma 8.3
- Dentro de las entradas de directorio se lleva la cuenta de cuántas fueron necesarias para formar el LFN completo.
- En el offset 13 del byte 1 de cada entrada se guarda un checksum para evitar corrupción en el filesystem.
- Los atributos tiene la marca 0Fh para poder distinguir los que corresponden a LFN

BSD - Berkeley Software Distribution

En el fs de Unix con el cual se venía trabajando, cada drive esta dividido en uno o mas particiones, cada una de estas particiones podían contener un file system. Pero un fs no se expande a través de varias particiones. Un fs esta descripto por el Superblock, que contiene información básica del mismo.

Dentro del fs encontramos archivos, y el acceso a los mismos normalmente incurre en una larga búsqueda desde el inodo de un archivo hasta los datos del mismo. Los archivos de un directorio no están asignados en bloques consecutivos de inodos, causando que muchos bloques que no están contiguos deban ser accedidos cuando se ejecutan operaciones en inodos de muchos archivos en un mismo directorio.

La asignación de bloques a los archivos es causante de un limitado throughput del fs. El fs no transmite más de 512 bytes por transacción y suele encontrarse con que la próxima lectura no se encontrará en el próximo cilindro, forzando a la realización de seeks.

Para mejorar esto se cambio el tamaño del bloque, haciendo que la transferencia de datos sea el doble y que muchos archivos pudiesen ser accedidos sin la necesidad de leer bloques de indirección.

Un problema que aun persistía era que la lista de bloques libres pronto se convertía en una lista con bloques dados en forma random, haciendo que los bloques de un archivo no estuviesen ubicados en forma estratégica para mejorar las lecturas.

En el fs llamado BSD, que es una mejora al System V, cada drive contiene uno o más fs. Los fs son descriptos por los superblocks, localizados al comienzo de cada partición de los fs. Debido a que los superblocks contienen información critica, se replican para protección en caso de perdidas o fallas. Esto se hace cuando el fs se crea.

Para asegurar la posibilidad de crear archivos grandes sin tener que utilizar mas de dos niveles de indirección, el mínimo tamaño del bloque de un fs es 4096 bytes (o >= 4096 bytes en potencia de dos). El tamaño del bloque de un fs se graba en el Superblock, entonces es posible que fs con tamaños distintos de bloques puedan ser accedidos en el mismo sistema. El tamaño solo puede ser decidido al crear el fs y no se puede modificar.

Este nuevo fs divide la partición en una o más áreas llamadas grupos de cilindros. Un grupo de cilindro es un número de cilindros consecutivos en el disco. Asociado a cada grupo de cilindro encontramos información que incluye espacio para inodos, un bitmap describiendo bloques disponibles dentro del cilindro e información acerca del uso de los bloques dentro del cilindro, además de una copia del Superblock. El bitmap de bloques disponibles dentro del grupo de cilindros reemplaza la tradicional lista de bloques libres del Superblock. Para cada grupo de cilindros un número definido de inodos se asigna en la creación del fs. La política que se sigue determina alocar un inodo por cada 2048 bytes de espacio en el grupo.

La información de cada grupo de cilindros podría ser guardada siempre al comienzo de cada uno. Sin embargo, si se siguiese esta idea, toda la información crucial, quedaría siempre guardada en el primer plato, y si este sufriese algún daño, se perdería toda la información. Por eso, la información se guarda en distintos offsets dentro de cada grupo, en general se guarda como en espiral con respecto de todos los grupos de cilindros.
Se presenta un problema con respecto al tamaño de los bloques (4096 bytes): la mayoría de los archivos en UNIX son archivos pequeños. De esta manera, al asignar un bloque de 4096 bytes a un archivo pequeño, se esta desperdiciando mucho espacio.

Entonces para poder seguir utilizando bloques grandes que mejoran la transferencia de datos y aceleran los tiempos de lectura, sin sufrir las consecuencias del desperdicio de espacio, se divide al bloque en fragmentos (cada uno de los cuales es direccionable). El mapa de bloques asociado a cada grupo de cilindros guarda el espacio disponible basándose en fragmentos. Para decidir si un bloque esta disponible, se analizan sus fragmentos.

Entonces es posible guardar dentro de un bloque, fragmentos de distintos archivos, pero un archivo no puede tener fragmentos en distintos bloques.

Cada vez que se escriben datos a un archivo, el sistema chequea si el tamaño del archivo ha crecido. Si el archivo necesita ser expandido para albergar más información, una de estas tres condiciones existe:

1. Hay suficiente espacio en el bloque o fragmento asignado para albergar la nueva información.
   =>
   Esta es guardada en el espacio disponible.

2. El archivo no contiene fragmentos y el último bloque del mismo no tiene suficiente espacio para albergar la nueva información.
   =>
   Se completa el bloque con parte de la nueva información, si lo que queda por almacenar alcanza para llenar un bloque entero, entonces se aloca un bloque y se almacena la información. Se continúa así hasta que la información ocupe menos de un bloque. Entonces un bloque con los fragmentos necesarios se asigna y se guarda la información.

3. El archivo contiene uno o mas fragmentos y estos no contienen suficiente espacio para albergar la nueva información. Si el tamaño de los datos en los fragmentos mas el tamaño de la nueva información superan la capacidad de un bloque entonces se asigna un bloque y se guardan allí los datos de los fragmentos mas la nueva información. Y luego se continúa como en el punto 2. Si la suma de información no ocupa más de un bloque, entonces se asigna un bloque y se almacenan los fragmentos necesarios para albergar la información.

Para poder realizar esto, se necesita que el fs no se llene. Para cada fs existe un parámetro que determina el porcentaje mínimo de bloques libres que deben existir. Si este mínimo no existiese entonces el administrador es el único capaz de poder continuar asignando bloques. Este porcentaje puede ser modificado.

Un logro de este nuevo fs es la capacidad que tiene de poder parametrizar las aptitudes y características de almacenamiento para que los bloques puedan ser asignados en una configuración optima. (Velocidad del procesador, capacidad de transferencia de datos, etc.).

Este fs trata de asignar nuevos bloques en el mismo cilindro del bloque anterior de un archivo. Y gracias a los parámetros del hardware, se ubicarán convenientemente según el tiempo rotacional. Se trata también de distribuir información que no se relaciona a través de los diferentes grupos de cilindros.
Los inodos de archivos dentro de un mismo directorio son frecuentemente accedidos en forma grupal. Se trata entonces de ubicar los inodos de archivos de un mismo directorio en el mismo grupo de cilindro. Para asegurarse de que los archivos sean distribuidos a través del disco, una política diferente se sigue. Un nuevo directorio se va a ubicar en el grupo de cilindros que tenga mas cantidad de inodos libres y la menor cantidad de directorio en el.

Como los bloques de un mismo archivo en general se acceden en forma conjunta, la policita establece que trate de ubicar los bloques del archivo en el mismo grupo de cilindros, preferentemente en una posición rotacional conveniente en el mismo cilindro. El problema que se presenta es que los archivos grandes van a ocupar rápidamente el espacio en un grupo de cilindros, y esto hará que tanto el archivo como otros archivos tengan que ubicar sus datos en otros grupos. Para evitar esto, y hacer que los grupos de cilindros traten de nunca estar llenos, se sigue la siguiente política: una vez que un archivo exceda los 48 bytes, se cambiará de grupo de cilindros y se volverá a cambiar una vez que llegue al mb de datos. El nuevo grupo de cilindros será elegido de aquellos que tengan el mayor número de bloques libres.

A la hora de asignar un bloque a un archivo y no encontrar el próximo bloque disponible, se pueden dar las siguientes situaciones:

1. Se elige el próximo bloque rotacionalmente cercano y disponible en el mismo cilindro.
2. Si no hay bloques disponibles en el mismo cilindro, se usa un bloque dentro del mismo grupo de cilindros.
3. Si el grupo de cilindros esta completamente lleno, se hashea el nro de cilindro para elegir otro grupo de cilindros y buscar un bloque libre.
4. Finalmente, si el hash falla, se aplica una búsqueda exhaustiva en todos los grupos de cilindros.