Sistemas Operativos
File System II

Implementacion de un Filesystem

Conceptos clave:

- Requisitos de un filesystem
- vsfs

Very Simple File System

A continuación se verá la descripción de la implementación de **vsfv (Very Simple File System)** descrito en el capítulo 40 del [ARP]. Este file system es una versión simplificada de un típico sistema de archivos unix-like. Existen diferentes sistemas de archivos y cada uno tiene ventajas y desventajas.

Para pensar en un file system hay que comprender dos conceptos fundamentales:

- El primero es la estructura de datos de un sistema de archivos. En otras palabras cómo se guarda la información en el disco para organizar los datos y metadatos de los archivos. El sistema de archivos vsfs emplea una simple estructura, que parece un arreglo de bloques.
- El segundo aspecto es el método de acceso, como se machean las llamadas hechas por los procesos , como open(), read(), write(), etc. en la estructura del sistema de archivos.

Requisitos de diseño

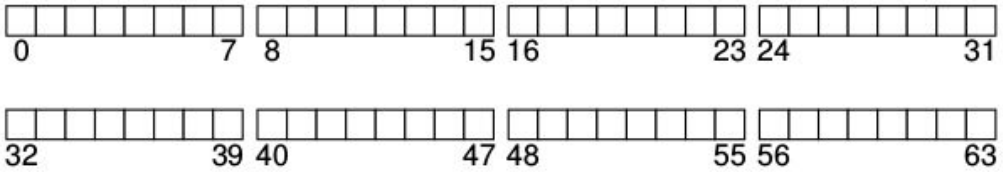
El objetivo principal del diseño de un filesystem es poder construir la abstracción de “archivo” sobre un medio de almacenamiento basado en bloques.

Por eso de alguna u otra forma, todos los filesystems contienen lo siguiente:

- La **estructura de índice** de un archivo proporciona una forma de localizar cada bloque del archivo. Las estructuras de índice suelen ser algún tipo de árbol para lograr escalabilidad y soportar la localidad.
- El **mapa de espacio libre** de un sistema de archivos proporciona una forma de asignar bloques libres para expandir un archivo.

Organización general

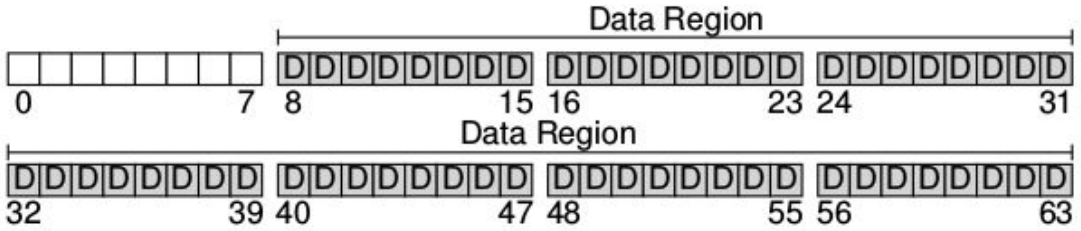
- Lo primero que se debe hacer es dividir al disco en bloques, los sistemas de archivos simples, como este suelen tener bloques de un solo tamaño. Los bloques tienen un tamaño de 4 kBytes.
- La visión del sistema de archivos debe ser la de una partición de N bloques (de 0 a N-1) de un tamaño de N * 4 KB bloques. si suponemos en un disco muy pequeño, de unos 64 bloques, este podría verse así:



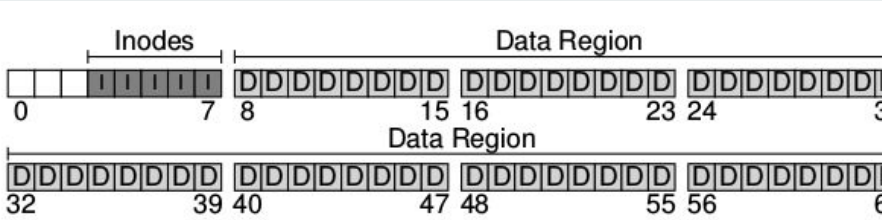
Organización general

- A la hora de armar un sistema de archivos una de las cosas que es necesario almacenar es …. por supuesto que datos, de hecho la mayor cantidad del espacio ocupado en un file system es por los datos de usuarios. Esta región se llama por ende data region.

- Otra vez en nuestro pequeño disco es ocupado por ejemplo por 56 bloques de datos de los 64:



Organización general



El sistema de archivos debe mantener información sobre cada uno de estos archivos. Esta información es la metadata y es de vital importancia ya que mantiene información como: qué bloque de datos pertenece a un determinado archivo, el tamaño del archivo, etc. Para guardar esta información, en los sistemas operativos unix-like, se almacena en una estructura llamada inodo.
Los inodos también deben guardarse en el disco, para ello se los guarda en una tabla llamada inode table que simplemente es un array de inodos almacenados en el disco:

Organización general

Cabe destacar que los inodos no son estructuras muy grandes, normalmente ocupan unos 128 o 256 bytes.

Suponiendo que los inodos ocupan 256 bytes , un bloque de 4KB puede guardar 16 inodos por ende nuestro sistema de archivo tendrá como máximo 80 inodos. Esto representa también la cantidad máxima de archivos que podrá contener nuestro sistema de archivos.

Sobre el calculo de inodos

Aquí tenemos que hacer una aclaración sobre el ejemplo presentado, que esta extraido de [Arpaci] y sobre el cual en nuestra cátedra diferimos.

En el ejemplo se puede ver que el autor toma 5 bloques de inodos donde cada bloque puede contener 16 inodos. En este sistema entonces habra un máximo de 80 inodos posibles.

Un inodo se asocia a un (y solo un) archivo. Y el archivo ocupará como mínimo un bloque (no se pueden compartir bloques entre archivos).

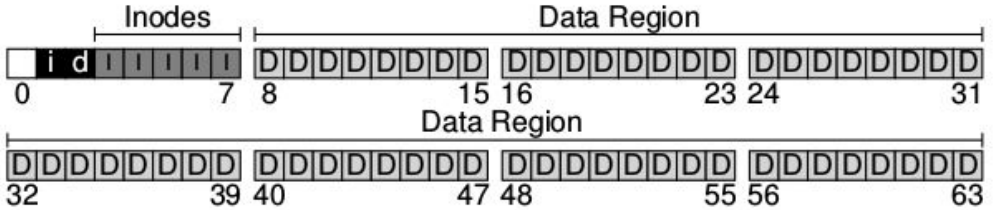
Si partimos de un disco que tiene 64 bloques, diseñar para 80 archivos es un desperdicio. Con 4 bloques de inodos hubiera sido suficiente.

Mas aun, como algunos de esos bloques se usaran no para archivos sino para superbloque, ibitmap,bbitmap y inodos, habra menor cantidad de archivos posibles (especificamente 56)

Organización general

El sistema de archivo tiene los datos (D) y los inodos (I) pero todavía nos falta. Una de las cosas que faltan es saber cuales inodos y cuáles bloques están siendo utilizados o están libres. Esta estructura de alocación es fundamental en cualquier sistema de archivos. Existen muchos métodos para llevar este registro pero en este caso se utilizará una estructura muy popular llamada bitmap. Una para los datos data bitmap ora para los inodos inode bitmap.

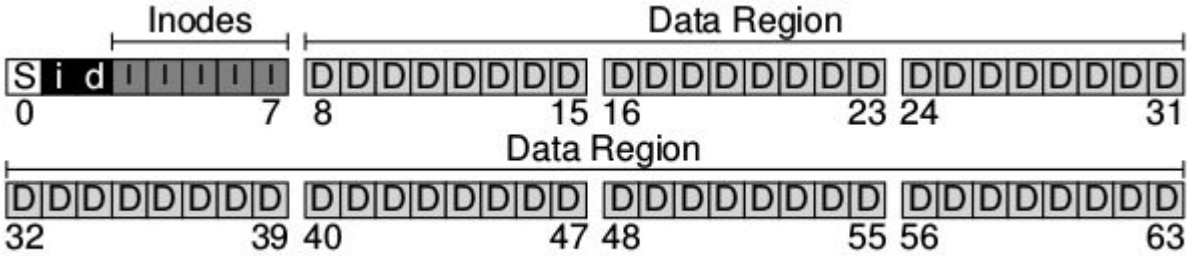
Un bitmap es una estructura bastante sencilla en la que se mapea 0 si un objeto está libre y 1 si el objeto está ocupado. En este cada i sería el bitmap de inodos y d seria el bitmap de datos:



Organización general

Se podrá observar que queda un único bloque libre en todo el disco. Este bloque es llamado Super Bloque (S). El superbloque contiene la información de todo el file system, incluyendo:

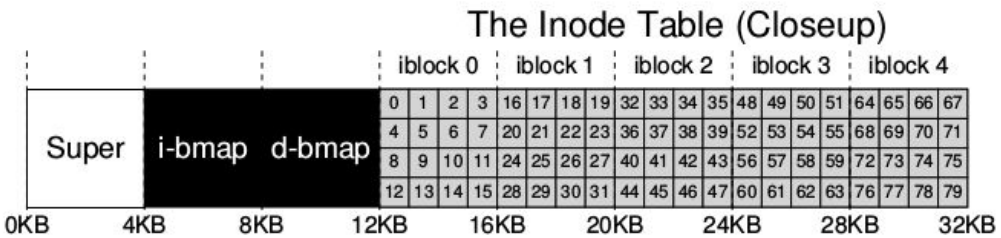
1. cantidad inodos
2. cantidad de bloques
3. donde comienza la tabla de inodos –> bloque 3
4. donde comienzan los bitmaps



Los Inodos

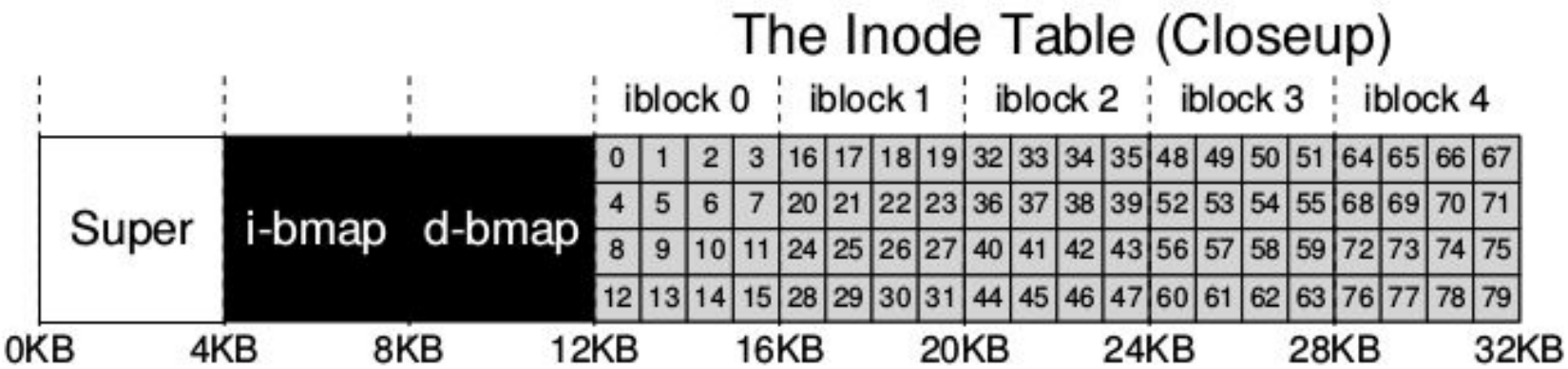
Esta es una de las estructuras almacenadas en el disco más importantes. Casi todos los sistemas de archivos unix-like son así. Su nombre probablemente provenga de los viejos sistemas UNIX en los que estos se almacenaban en un arreglo, y este arreglo estaba indexado de forma de cómo acceder a un inodo en particular.

Un inodo simplemente es referido por un número llamado inumber que sería lo que hemos llamado el nombre subyacente en el disco de un archivo. En este sistema de archivos y en varios otros, dado un inumber se puede saber directamente en que parte del disco se encuentra el inodo correspondiente



Los Inodos

Para leer el inodo número 32, el sistema de archivos debe: 1. debe calcular el offset en la región de inodos 32* sizeof(inode) = 8192 2. sumarlo a la dirección inicial de la inode table en el disco o sea 12Kb+ 8192 bytes 3. llegar a la dirección en el disco deseada que es la 20 KB.



Ejemplo de parcial

Diseñe un vsfs para un sistema de 1024 bloques, cada bloque 4kb, cada inodo 256 bytes.

Criterio de la catedra. Asumir que en un disco de N bloques entran N archivos*

Independientemente de los bloques especiales (además cuando N -> infinito, la suposicion se aproxima)

Resolucion:

- Cantidad de inodos necesarios: 1024
- Cantidad de inodos por bloque: 4096/256 = 16
- Cantidad de bloques de inodos: 1024/16 = 64
- Cantidad de los bitmap: 4096*8=32768 (sobra! El disco tiene 1024 bloques, y 1024 inodos)

* *se puede calcular un optimo de la cantidad máxima de inodos, que será generalmente menor a la cantidad de bloques. Hacerlo como ejercicio. Pero a menos que lo pidamos explícitamente, el criterio en rojo es lo que vale en los exámenes (además es más fácil)*

Ejemplo de parcial

Solucion:

- 1 superbloque
- 1 bloque de bitmap de inodos
- 1 bloque de bitmap de bloques
- 64 bloques de inodos
- 1024-64-1-1-1=957 bloques de datos

Casos de estudio

Conceptos clave:

- FAT
- FFS

FAT / FAT-32

Microsoft File Allocation Table (FAT) este file system se implementó en los 70, Fue el Sistema de archivos de MS-DOS y de las versiones tempranas de Windos.

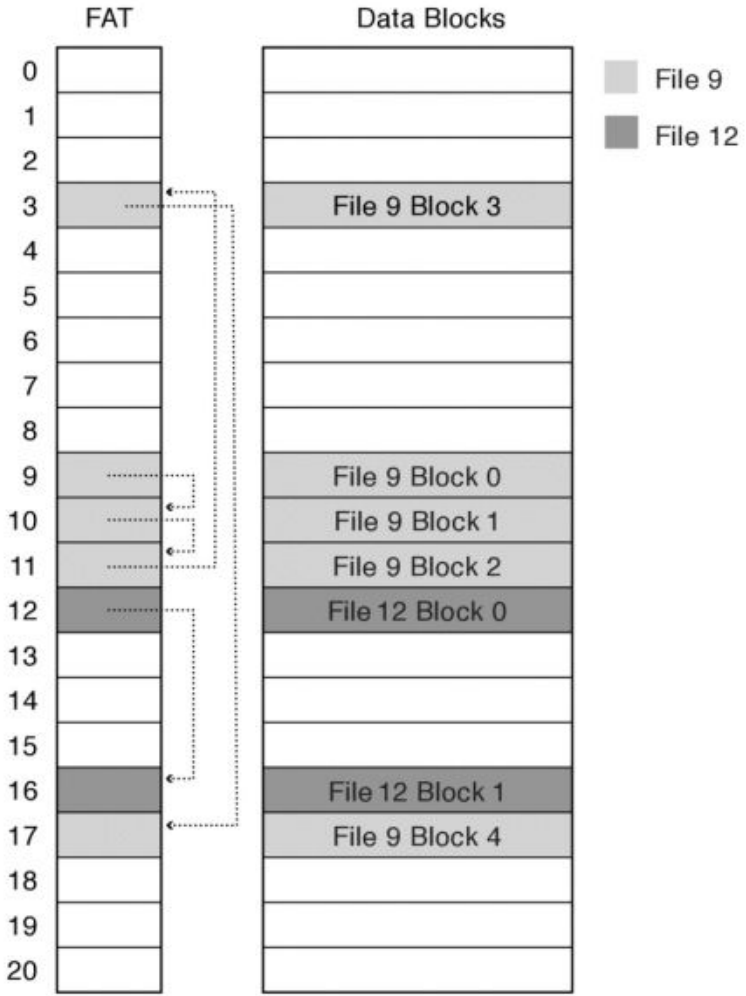
FAT fue mejorado por su versión FAT-32, el cual soporta volúmenes de 2^{32}-1 bytes.

FAt obtiene su nombre de la file allocation table, un arreglo de entradas de 32 bits, en un área reservada del volumen.

Cada archivo en el sistema corresponde a una lista enlazada de entradas en la FAT, en la que cada entrada en la FAT contiene un puntero a la siguiente entrada.

La FAT contiene una entrada por cada bloque de la unidad de disco o volumen

FAT


FAT
Data Blocks
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
File 9 Block 3
File 9 Block 0
File 9 Block 1
File 9 Block 2
File 12 Block 0
File 12 Block 1
File 9 Block 4
File 9
File 12

FAT

Los directorios asignan a los nombres de archivo a números de archivo, y en el sistema de archivos FAT, el número de un archivo es el índice de la primera entrada del archivo en la FAT.

Así, dado el número de un archivo, podemos encontrar la primera entrada FAT y bloque de un archivo, y dada la primera entrada FAT, podemos encontrar el resto de las entradas y bloques FAT del archivo.

Seguimiento de espacio libre:. La FAT también se utiliza para el seguimiento del espacio libre. Si el bloque de datos i está libre,entonces FAT[i] contiene 0. Por lo tanto, el sistema de archivos puede encontrar un bloque libre escaneando a través de la FAT para encontrar una entrada puesta a cero.

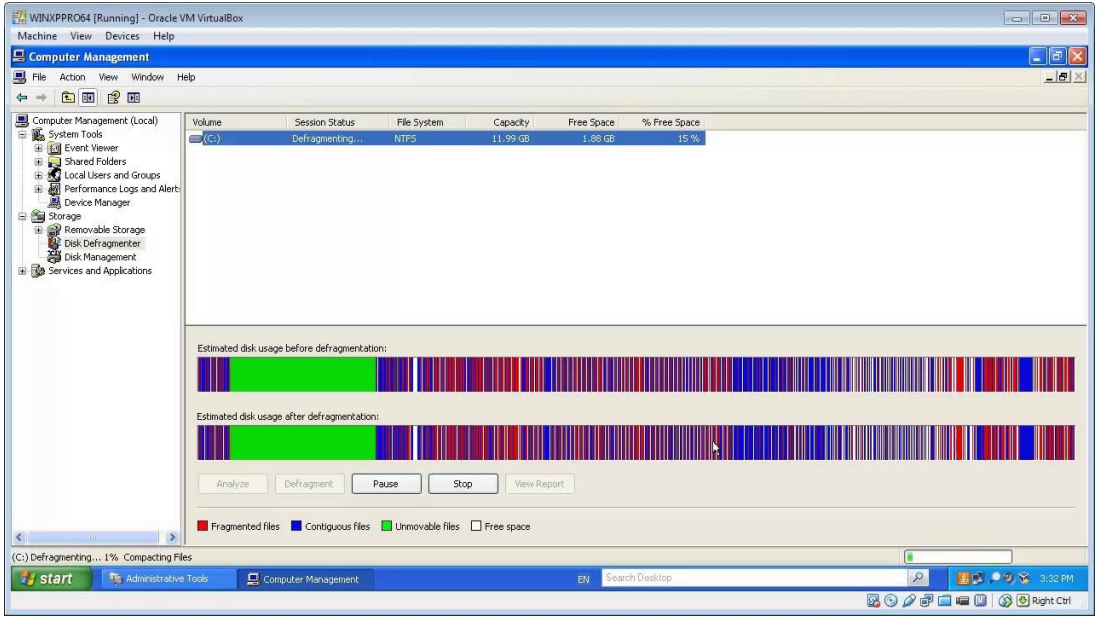
FAT

Heurísticas de localidad. Diferentes implementaciones de FAT pueden usar diferentes asignaciones estrategias, pero las estrategias de asignación de las implementaciones FAT suelen ser simples. Por ejemplo, algunas implementaciones usan un algoritmo de ajuste siguiente que escanea secuencialmente a través de la FAT a partir de la última entrada que se asignó y que devuelve la siguiente entrada libre encontrada.

Las estrategias de asignación simples como esta pueden fragmentar un archivo, esparciendo los bloques del archivo el volumen en lugar de lograr el diseño secuencial deseado. Para mejorar el rendimiento, los usuarios pueden ejecutar una herramienta de desfragmentación que lee archivos de sus ubicaciones existentes y los reescribe a nuevas ubicaciones con mejor localidad espacial. El desfragmentador FAT en

FAT

Windows XP, por ejemplo, intenta copiar los bloques de cada archivo que se distribuye múltiples extensiones a una sola extensión secuencial que contiene todos los bloques de un archivo.

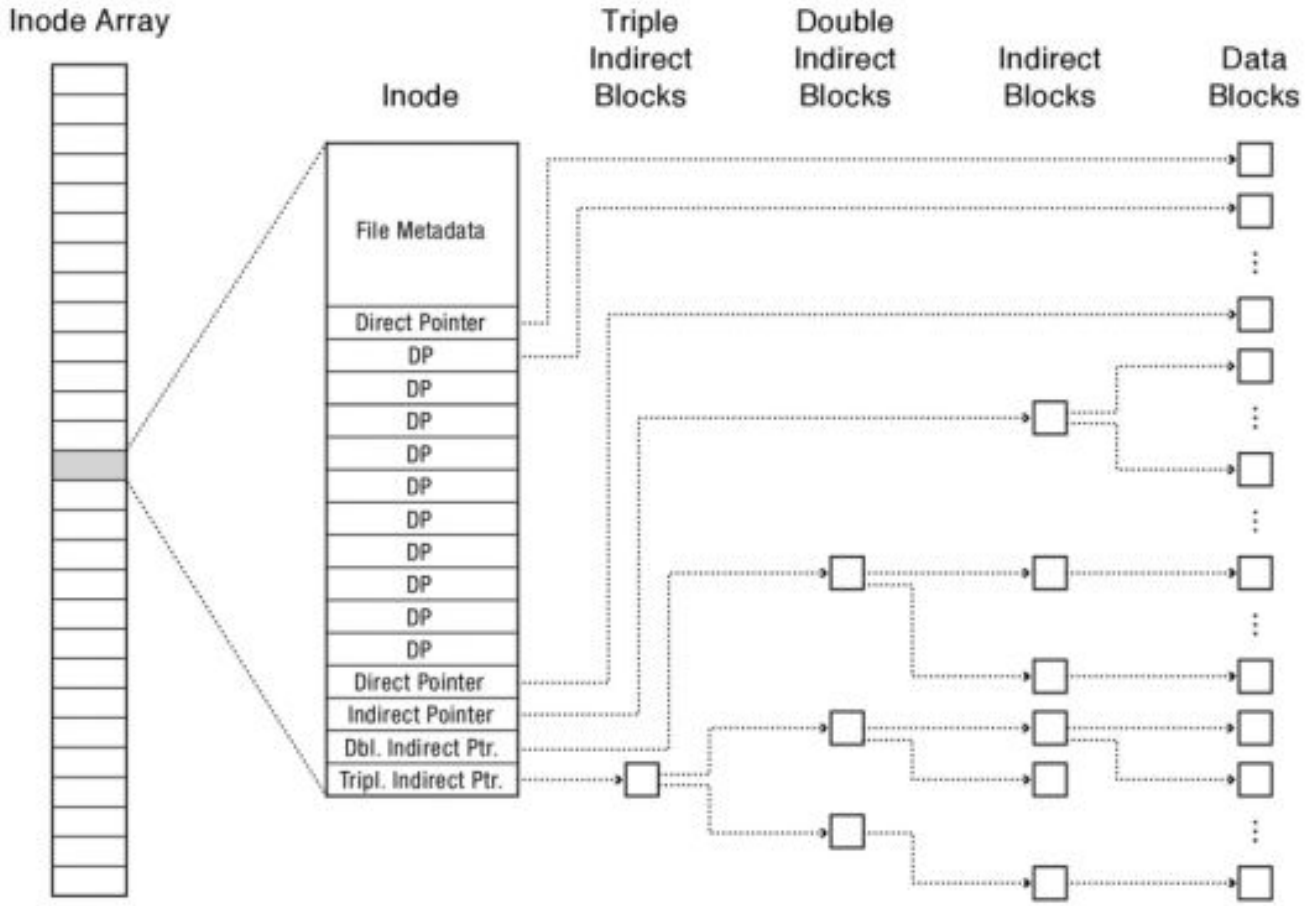


FFS: Fixed Tree

El Fast File System de Unix (FFS) ilustra ideas importantes tanto para indexar la información de un archivo de bloques para que puedan ubicarse rápidamente y para colocar datos en el disco para obtener una buena ubicación.

En particular, la estructura del índice de FFS, llamado índice multinivel, es un árbol cuidadosamente estructurado que permite a FFS localizar cualquier bloque de un archivo y que es eficiente tanto para grandes como para pequeños archivos

Dada la flexibilidad proporcionada por el índice multinivel de FFS, FFS emplea dos localidades heurísticas (colocación de grupos de bloques y espacio de reserva) que juntas suelen proporcionar buen diseño en disco.



FFS: Fixed Tree

FFS: Fixed Tree

El inodo (raíz) de un archivo también contiene una serie de punteros para ubicar los bloques de datos del archivo. (hojas). Algunos de estos punteros apuntan directamente a las hojas de datos del árbol y algunos de ellosapuntar a nodos internos en el árbol. Normalmente, un inodo contiene 15 punteros. los primeros 12

Los punteros son punteros directos que apuntan directamente a los primeros 12 bloques de datos de un archivo

El puntero 13 es un puntero indirecto, que apunta a un nodo interno del árbol llamado un bloqueo indirecto; un bloque indirecto es un bloque normal de almacenamiento que contiene una matriz de punteros directos.

FFS: Fixed Tree

Para leer el bloque 13 de un archivo, primero lee el inodo para obtener el indirecto puntero, luego el bloque indirecto para obtener el puntero directo, luego el bloque de datos. con 4 KB bloques y punteros de bloque de 4 bytes, un bloque indirecto puede contener hasta 1024 punteros, lo que permite archivos de hasta un poco más de 4 MB.

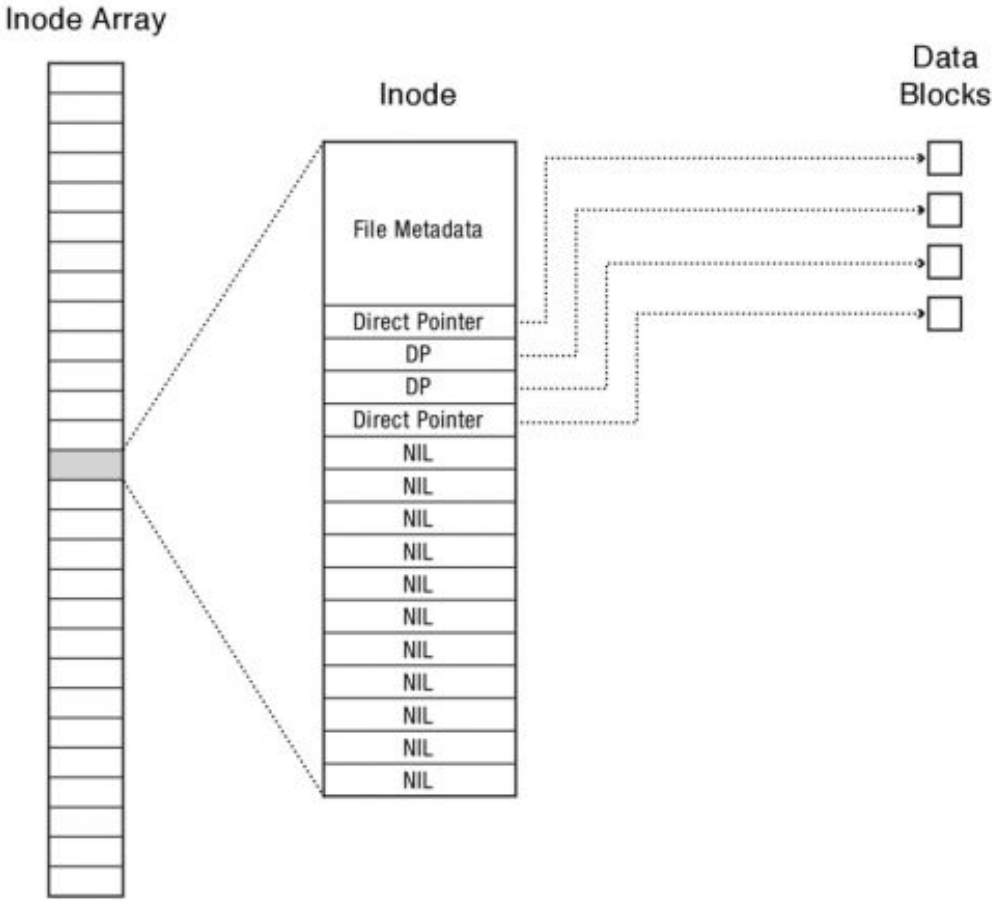
El puntero 14 es un puntero indirecto doble, que apunta a un nodo interno del árbol.

llamado doble bloqueo indirecto; un bloque indirecto doble es una matriz de punteros indirectos, cada uno de los cuales apunta a un bloqueo indirecto. Con bloques de 4 KB y punteros de bloque de 4 bytes, un doble .

El bloque indirecto puede contener hasta 1024 punteros indirectos. Así, una doble indirecta El puntero puede indexar hasta (1024)2 bloques de datos.

FFS: Fixed Tree

Finalmente, el puntero 15 es un puntero indirecto triple que apunta a un bloque indirecto triple que contiene una matriz de punteros indirectos dobles. Con bloques de 4 KB y punteros de bloque de 4 bytes, un puntero indirecto triple puede indexar hasta (1024) 3 bloques de datos que contienen 4 KB × 1024^3 = 212 × 230 = 242 bytes (4 TB).



Small FFS: de bloques de 4kb

FFS: Fixed Tree

El Fast File System de Unix (FFS) ilustra ideas importantes tanto para indexar la información de un archivo bloques para que puedan ubicarse rápidamente y para colocar datos en el disco para obtener una buena ubicación.

En particular, la estructura del índice de FFS, llamada índice multinivel, es un árbol cuidadosamente estructurado que permite a FFS localizar cualquier bloque de un archivo y que es eficiente tanto para grandes como para pequeños archivos

Dada la flexibilidad proporcionada por el índice multinivel de FFS, FFS emplea dos localidades heurísticas (colocación de grupos de bloques y espacio de reserva) que juntas suelen proporcionar buen diseño en disco.

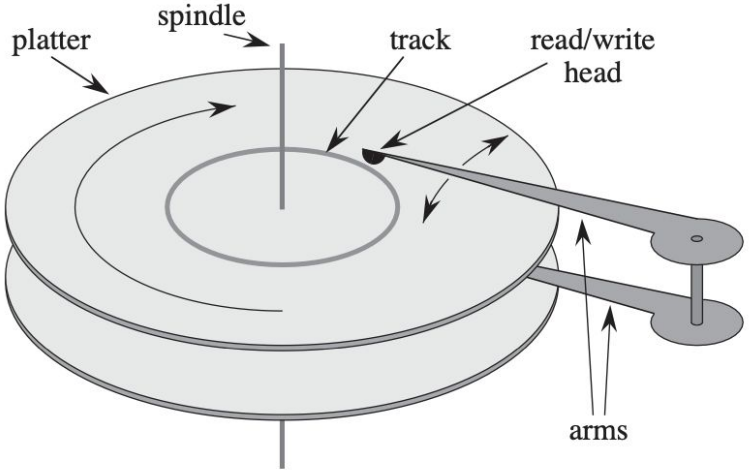
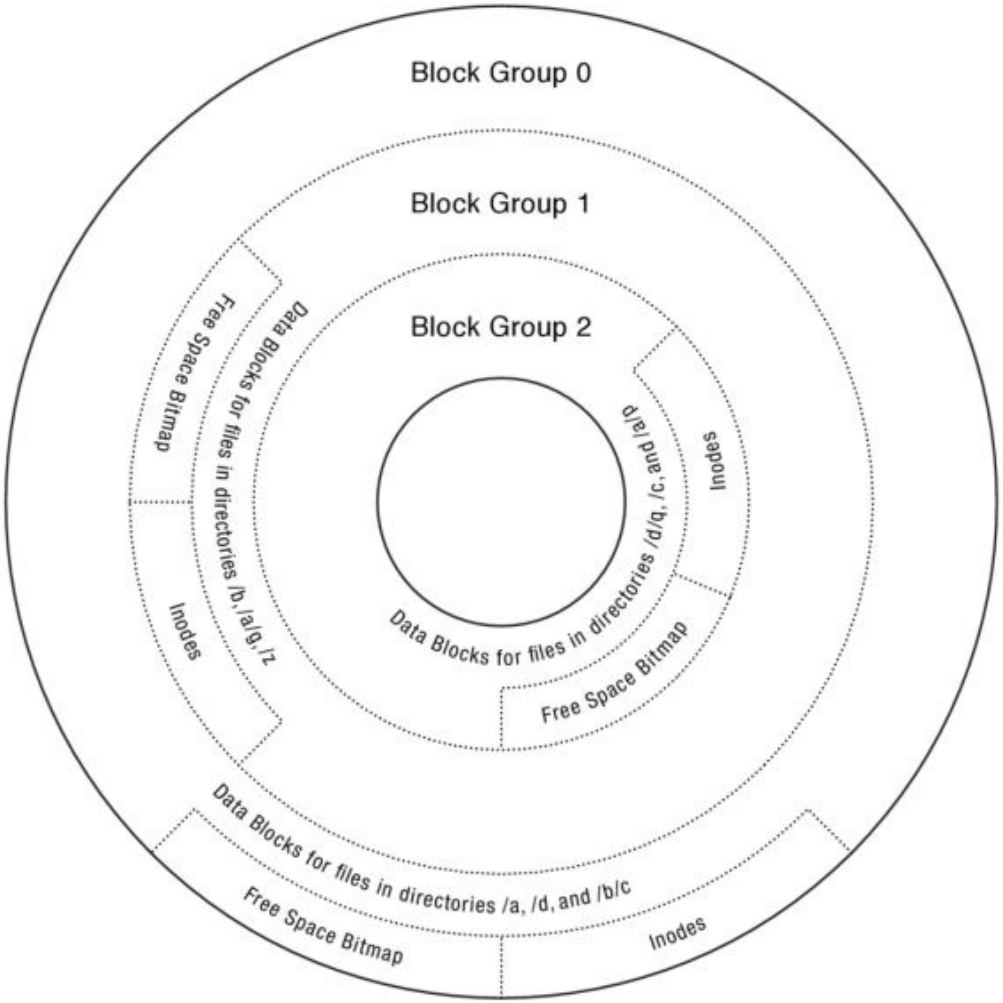


Figure 18.2 A typical disk drive. It comprises one or more platters (two platters are shown here) that rotate around a spindle. Each platter is read and written with a head at the end of an arm. Arms rotate around a common pivot axis. A track is the surface that passes beneath the read/write head when the head is stationary.

FFS: Fixed Tree



Buffer cache

Conceptos clave:

- FAT
- FFS

Tiempos tipicos de operaciones

Operation	Time (ns)	Time (us)
L1 cache reference	0.5 ns	
L2 cache reference	7 ns	
Mutex lock/unlock	25 ns	
Main memory reference	100 ns	
Send 1K bytes over 1 Gbps network	10,000 ns	10 us
Read 4K randomly from SSD*	150,000 ns	150 us
Read 1 MB sequentially from memory	250,000 ns	250 us
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 us (1 ms)
Disk seek	10,000,000 ns	10,000 us (10 ms)
Read 1 MB sequentially from disk	20,000,000 ns	20,000 us (20 ms)

Problema y solución

Los discos son muy lentos! Hay que guardar los bloques leídos en memoria principal

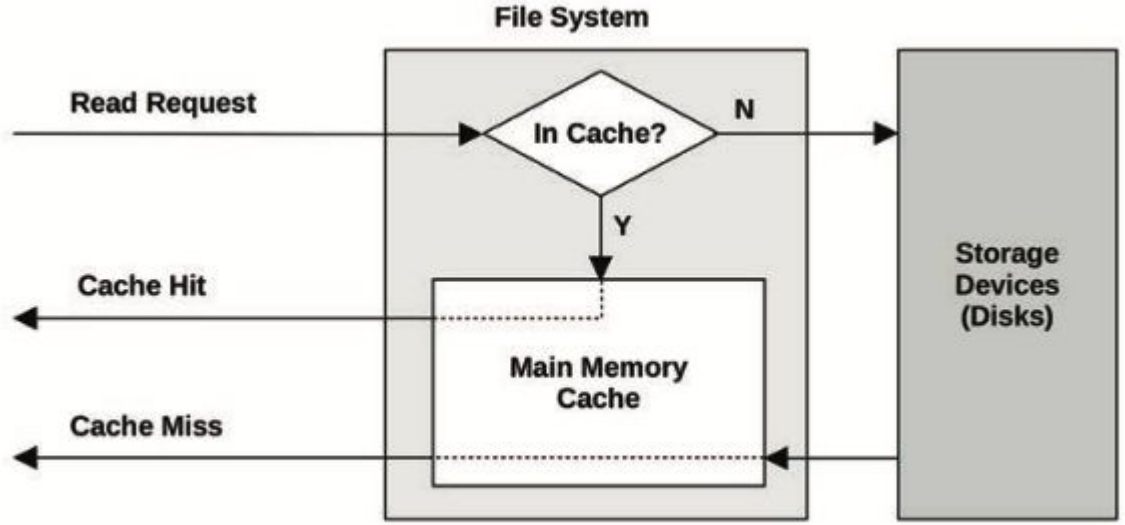


Figure 8.2 File system main memory cache

Buffer Cache

El buffer cache tiene dos funciones:

1. Sincronizar el acceso a los bloques de disco para asegurar que solo haya una copia de un bloque en la memoria y que solo un hilo del kernel use esa copia a la vez;

2. Almacenar en caché los bloques populares para que no sea necesario volver a leerlos desde el disco, que es más lento.

Page Cache en Linux

El **Page Cache** en Linux es una parte del sistema de administración de memoria que se utiliza para almacenar en memoria RAM los datos que se han leído o escrito en el disco, con el fin de acelerar el acceso a los archivos. Este mecanismo permite que las lecturas y escrituras a disco sean más rápidas, ya que evita acceder directamente al disco físico (más lento) si los datos ya están disponibles en la memoria.

El Page Cache ha reemplazado el Buffer Cache en Linux, y se considera a este último parte del sistema de caché del filesystem de Linux

Page Cache en Linux

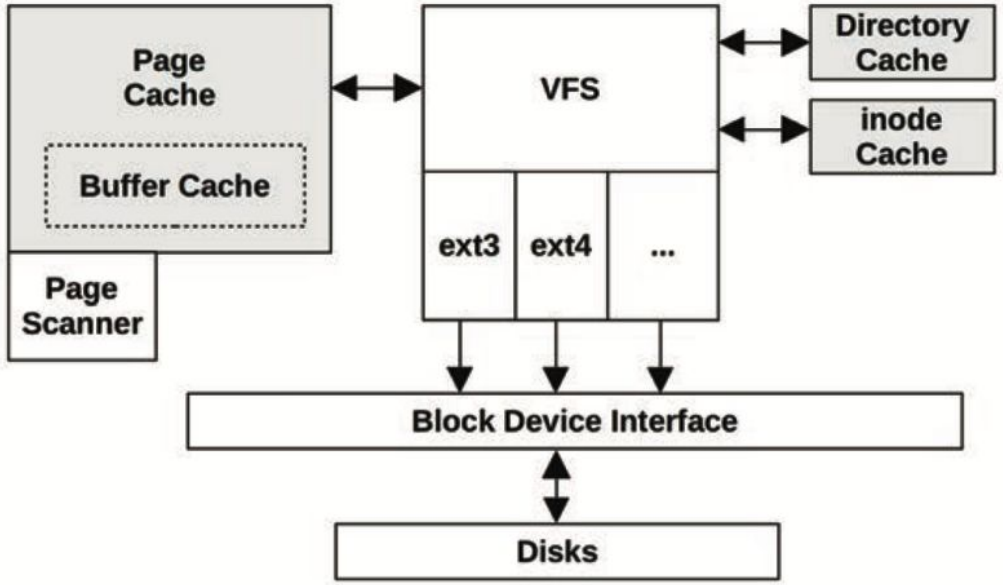
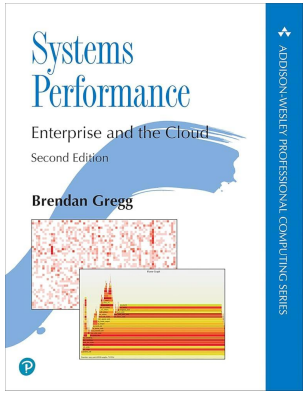


Figure 8.8 Linux file system caches

Véalo usted mismo!

```
ubuntu@primary:~$ sudo cachestat-perf
Counting cache functions... Output every 1 seconds.
    HITS   MISSES  DIRTIES     RATIO  BUFFERS_MB  CACHE_MB
    1234        0        0   100.0%          31       609
    1240        0        0   100.0%          31       609
    1228        0        0   100.0%          31       609
    1240        0        0   100.0%          31       609
    1237        2        3    99.8%          31       609
```



Brendan Gregg - Systems Performance, 2nd Edition

Ejemplo de aplicación: B-Tree

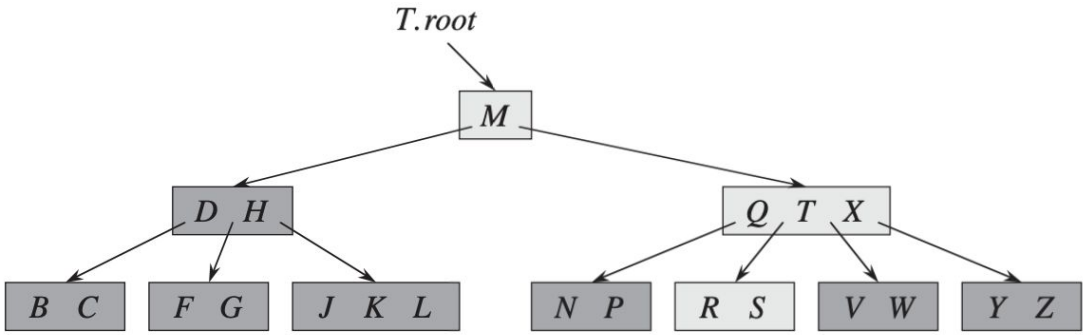
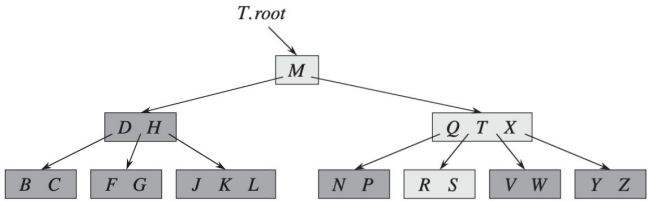


Figure 18.1 A B-tree whose keys are the consonants of English. An internal node x containing $x.n$ keys has $x.n + 1$ children. All leaves are at the same depth in the tree. The lightly shaded nodes are examined in a search for the letter R.

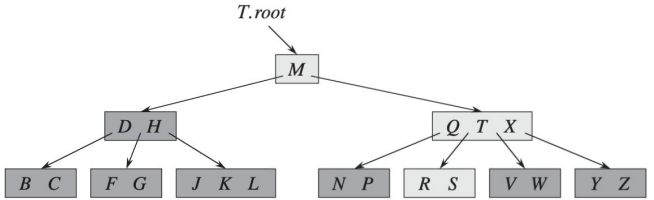
Ejemplo de aplicación: B-Tree

- El árbol B justifica su existencia porque los filesystems funcionan con bloques y no bytes.
- Es más “barato” leer muchos bytes de un disco rígido (tanto SSD como magnético)
- Además los bloques superiores se acceden más frecuentemente que los inferiores. Esto permite que una búsqueda en un árbol B se ejecute mayormente (o inclusive completamente) usando estructuras residentes en el cache de la RAM



Ejemplo de aplicación: B-Tree

- Por eso los sistemas de bases de datos se suelen diseñar en base a estructuras de datos que son múltiplos del tamaño del bloque.
- Postgres usa páginas de 8kb
- MySQL usa páginas por defecto de 16kb



Primitivas de sincronización: fsync

En sistemas modernos, para mejorar el rendimiento, el sistema operativo utiliza una técnica conocida como "buffered I/O", donde los datos escritos por write() primero se almacenan temporalmente en memoria (page cache) y luego, en algún momento futuro, el sistema operativo los escribe en disco. Este proceso puede ser diferido, y si ocurre un fallo del sistema antes de que los datos hayan sido escritos físicamente en el disco, la información puede perderse.

Una llamada a **write() no garantiza la persistencia inmediata de los datos en el disco**. La función write() solo asegura que los datos han sido transferidos al page cache del sistema operativo, pero no necesariamente que se hayan escrito físicamente en el disco.

Primitivas de sincronización: fsync

Para que los datos escritos sean persistentes inmediatamente (es decir, que estén almacenados de manera segura en el disco), se debe usar **fsync**() o **fdatasync**() después de la llamada a write(). Estas funciones fuerzan al sistema operativo a escribir cualquier dato pendiente del page cache asociado al archivo en el disco.

- **fsync()**: Fuerza a que se sincronicen tanto los datos del archivo como la metadata del mismo (por ejemplo, las marcas de tiempo de modificación, el tamaño del archivo, etc.).
- **fdatasync()**: Solo asegura que los datos del archivo se escriban en disco, pero no garantiza la sincronización de la metadata.

Primitivas de sincronización: mmap y msync

mmap() es una llamada al sistema en Linux que permite mapear un archivo o dispositivo directamente en el espacio de direcciones de un proceso.

Esto significa que, en lugar de leer o escribir en un archivo a través de funciones como read() o write(), el archivo se puede manipular directamente en memoria, como si fuera una región de memoria del proceso. Esto es útil para acceder a grandes cantidades de datos de manera eficiente, ya que elimina la necesidad de realizar múltiples llamadas al sistema y permite acceder a los datos directamente desde el page cache.

Es el mismo mmap que usamos para pedir mas memoria al kernel.

Primitivas de sincronización: mmap y msync

```
#include <sys/mman.h>

void *mmap(void addr[.length], size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void addr[.length], size_t length);
```

Primitivas de sincronización: mmap y msync

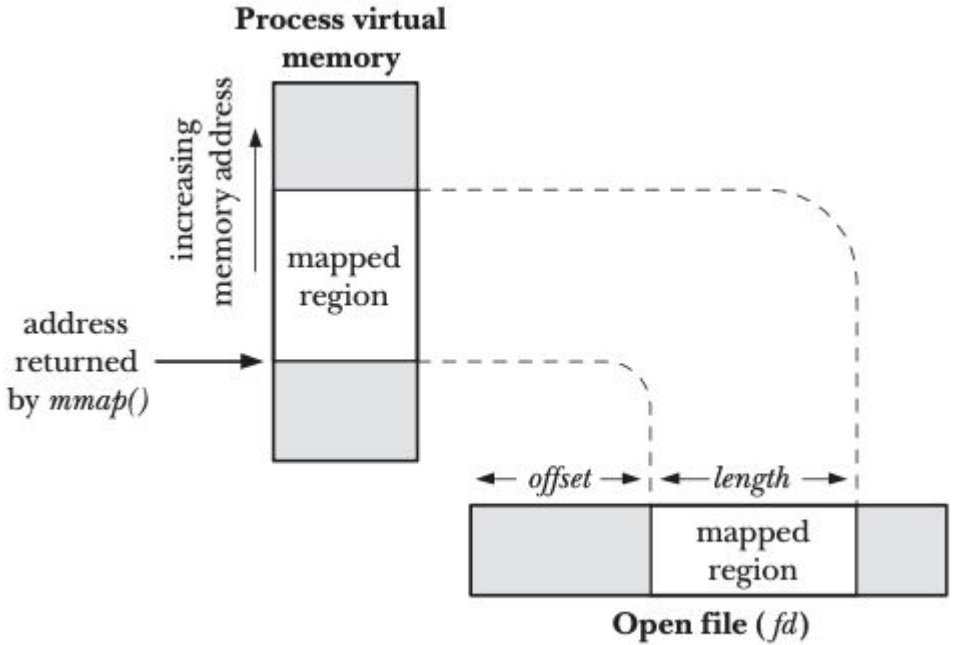


Figure 49-1: Overview of memory-mapped file

Primitivas de sincronización: mmap y msync

Cuando múltiples procesos crean mapeos compartidos de la misma región de un archivo, todos comparten las mismas páginas físicas de memoria. Además, las modificaciones en el contenido del mapeo se reflejan en el archivo.

Los mapeos de archivos compartidos cumplen dos propósitos:

- E/S mediante memoria mapeada e IPC (comunicación entre procesos)
- Consideramos cada uno de estos usos a continuación.

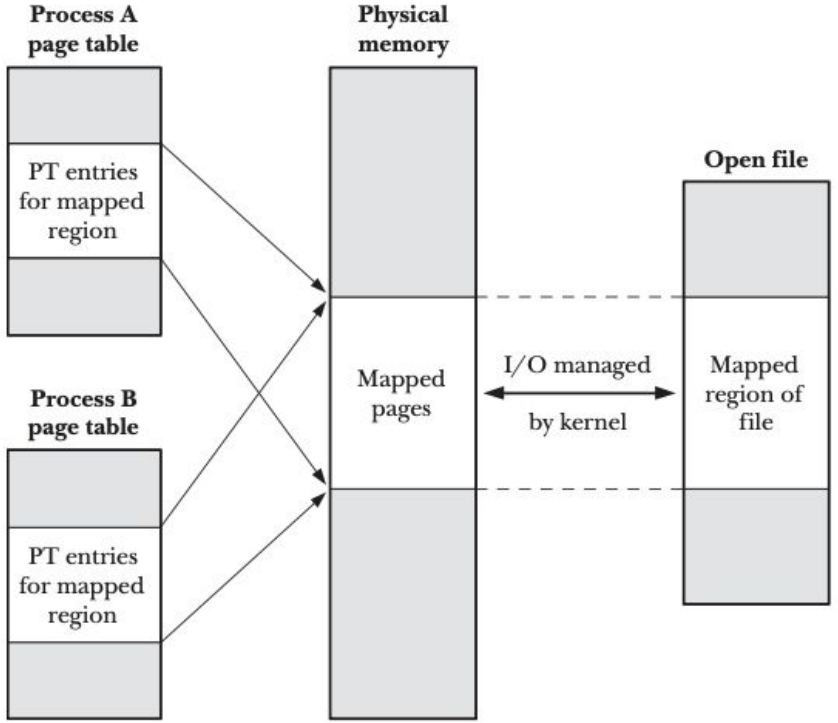


Figure 49-2: Two processes with a shared mapping of the same region of a file

Primitivas de sincronización: mmap y msync

¿Es persistente mmap() por sí solo?

No. Si se hacen cambios a una región de memoria mapeada usando mmap(), esos cambios pueden no ser persistentes hasta que el sistema decida escribir el contenido del page cache en el disco. Si ocurre un fallo antes de que esos datos sean sincronizados con el disco, los cambios pueden perderse.

¿Cómo garantizar la persistencia con mmap()?

Para garantizar que los cambios realizados en una memoria mapeada se escriban de manera persistente en el disco, se puede utilizar una llamada explícita a **msync**().

Primitivas de sincronización: mmap y msync

```
#include <sys/mman.h>

int msync(void addr[.length], size_t length, int flags);
```

- msync permite sincronizar solo la parte del archivo (mapeado) que nos interesa.

Resumen de buffering y caches

A nivel usuario, también se usan buffers. La biblioteca de C incluye un buffer además del buffer cache a nivel kernel.

Otros lenguajes pueden implementar sus propios buffers en memoria de usuario.

Pero recordar que existe otro cache.

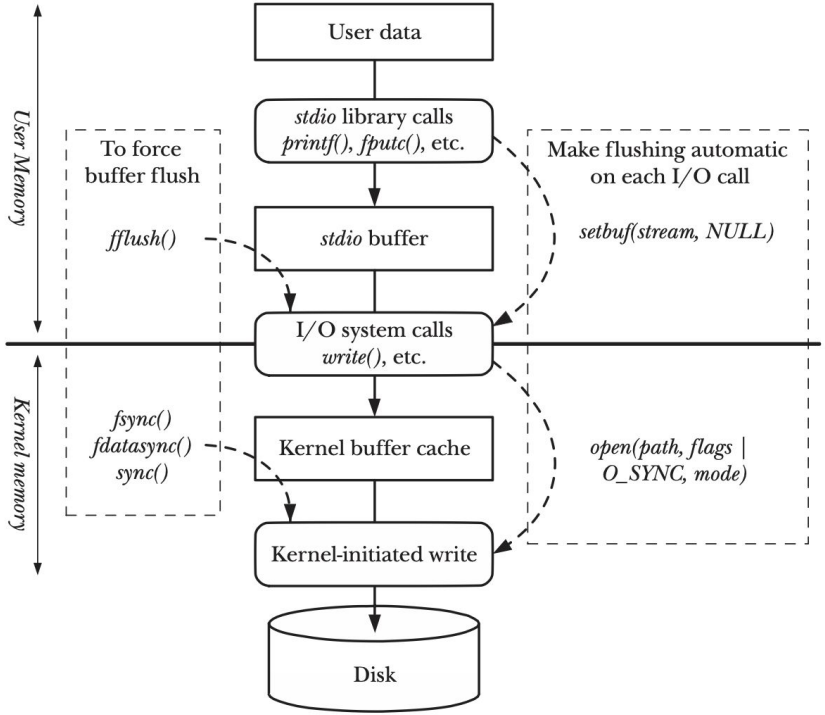


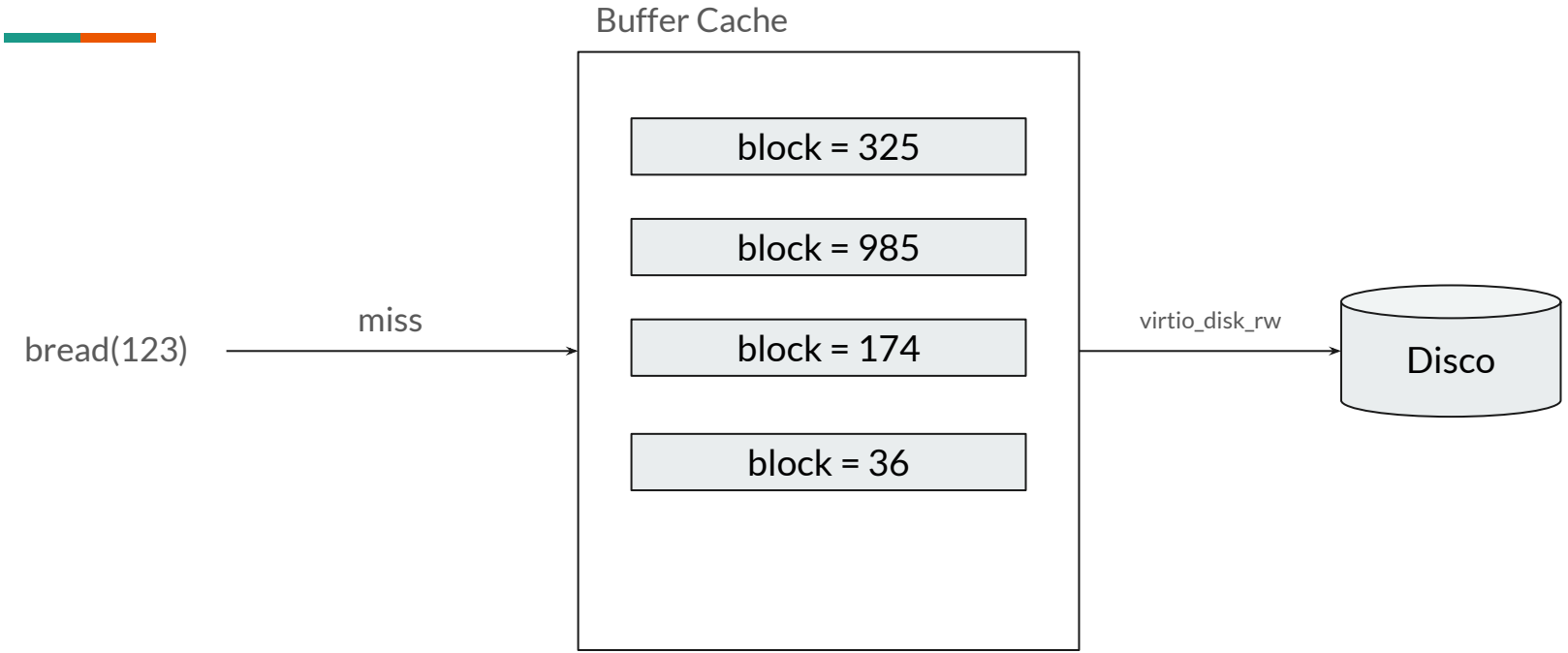
Figure 13-1: Summary of I/O buffering

Caso de estudio: xv6

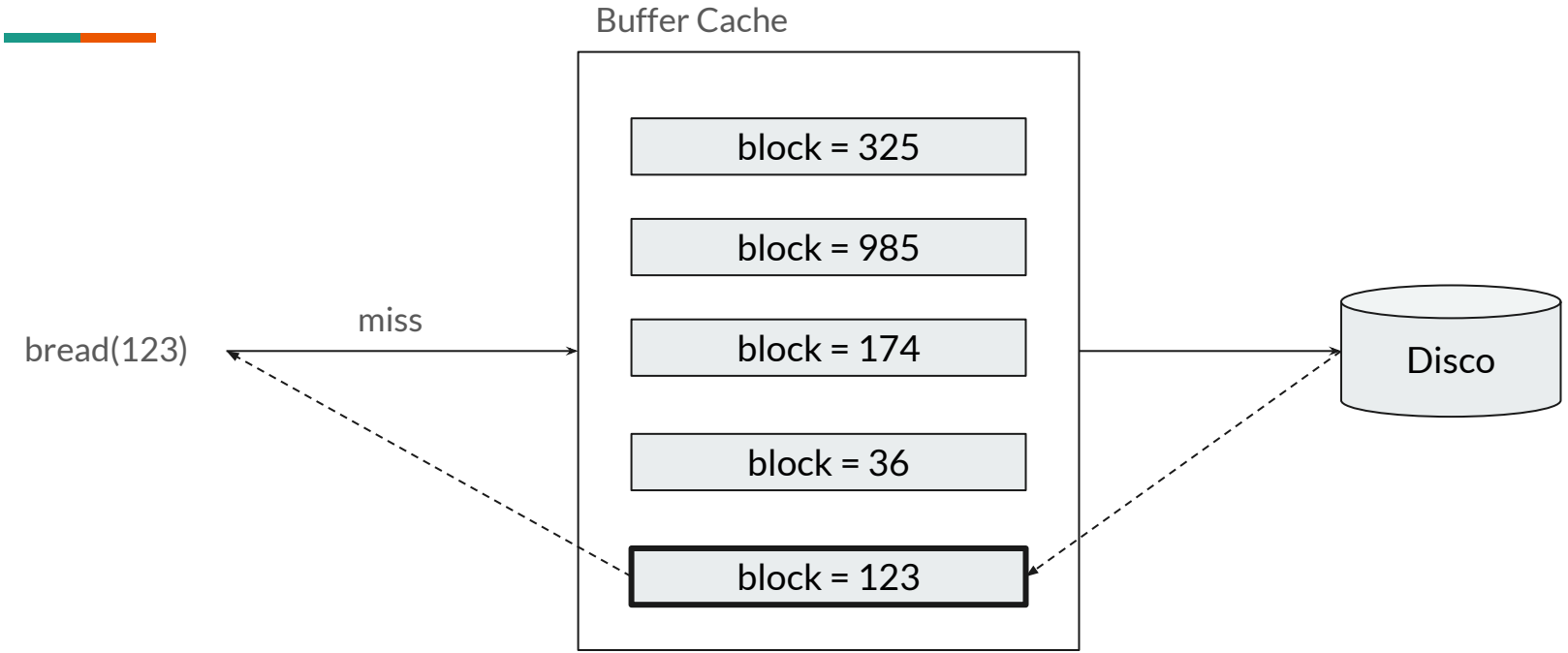
Conceptos clave:

- BufferCache
- Log

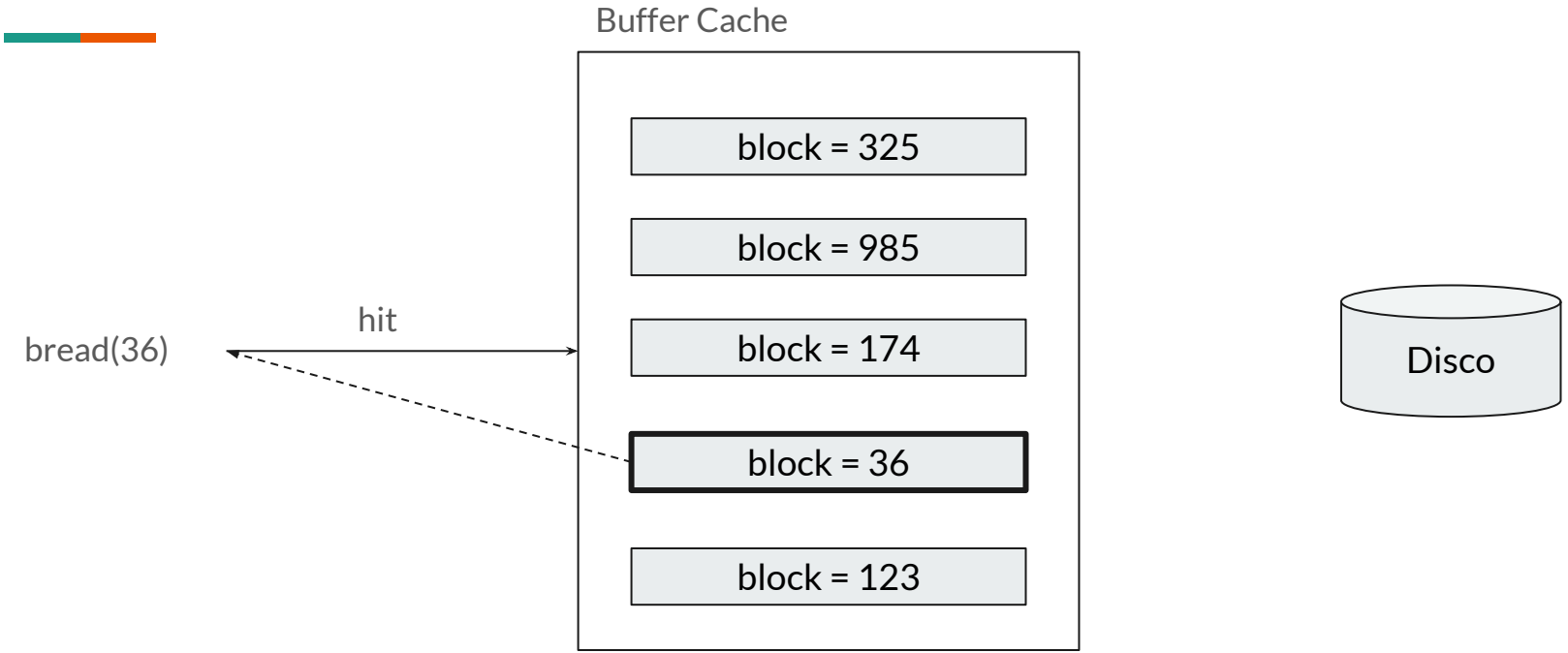
Ejemplo de miss



Ejemplo de miss



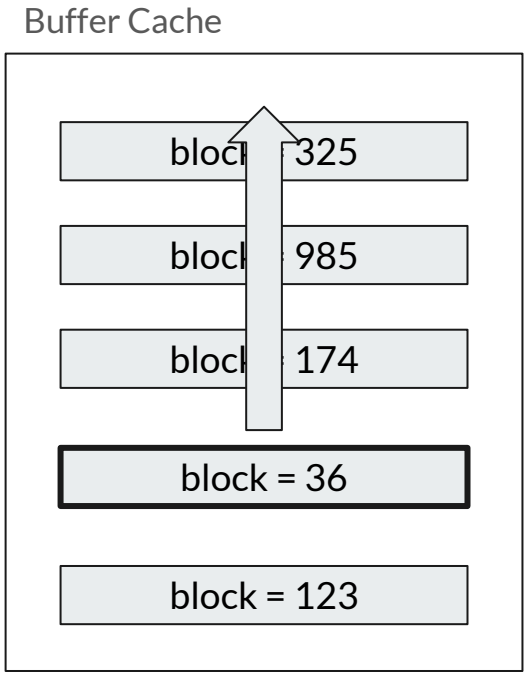
Ejemplo de hit



Remplazos LRU

Tanto para hits como para misses el bloque termina en el cache.

Cuando un bloque se utiliza, el mismo se mueve al tope de la pila

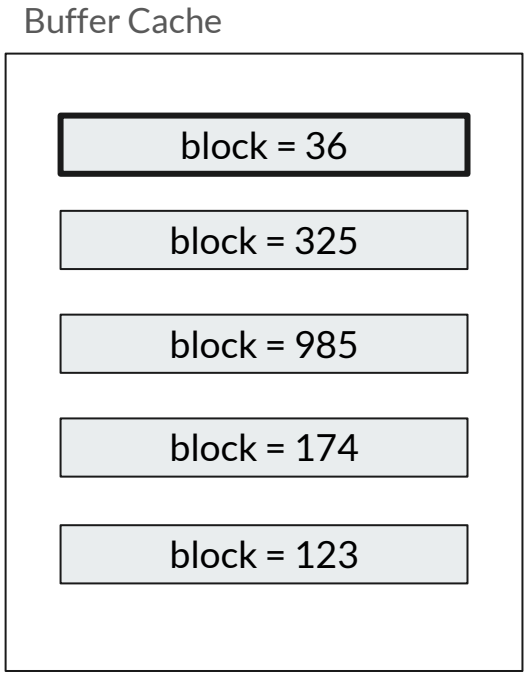


Remplazos LRU

Tanto para hits como para misses el bloque termina en el cache.

Cuando un bloque se utiliza, el mismo se mueve al tope de la pila.

Consecuentemente, los menos utilizados van a ir al fondo.

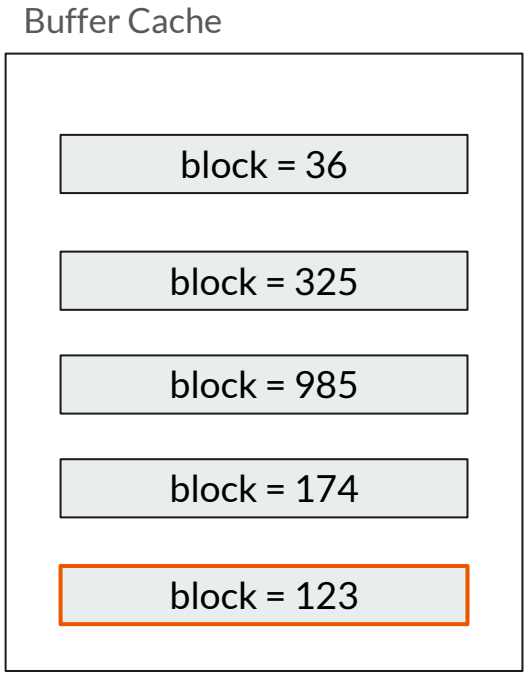


Remplazos LRU

Cuando se desea cargar un bloque nuevo y no hay espacio en el cache hay que **desalojar un bloque.**

Xv6 usa una politica **LRU, Least Recently Used.**

Esto es, buscar de abajo hacia arriba el primer bloque que no esté marcado como en uso, y desalojarlo

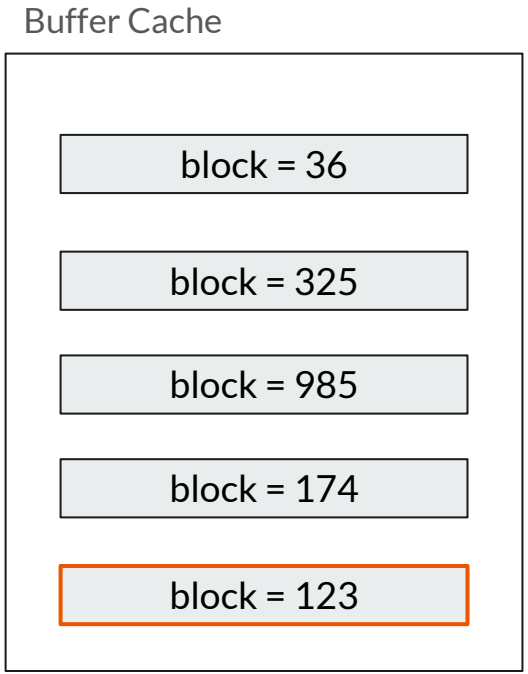


Remplazos LRU

Cuando se desea cargar un bloque nuevo y no hay espacio en el cache hay que **desalojar un bloque.**

Xv6 usa una politica **LRU, Least Recently Used.**

Esto es, buscar de abajo hacia arriba el primer bloque que no esté marcado como en uso, y desalojarlo

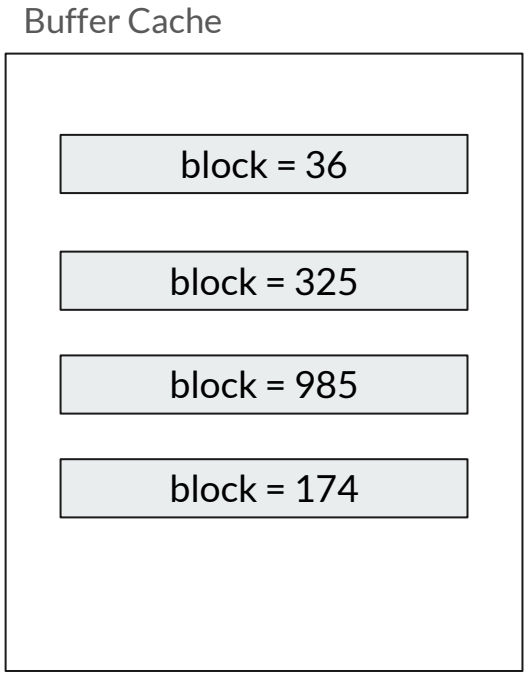


Remplazos LRU

Cuando se desea cargar un bloque nuevo y no hay espacio en el cache hay que **desalojar un bloque.**

Xv6 usa una politica **LRU, Least Recently Used.**

Esto es, buscar de abajo hacia arriba el primer bloque que no esté marcado como en uso, y desalojarlo



Estructura del sistema de filesystem en xv6

El resto del subsistema de filesystem de xv6 sigue una estructura de capas.

Es importante notar que la única capa que llama al driver del disco (virtio_disk_rw)

El buffer cache es la unica forma de “entrar y salir” del disco

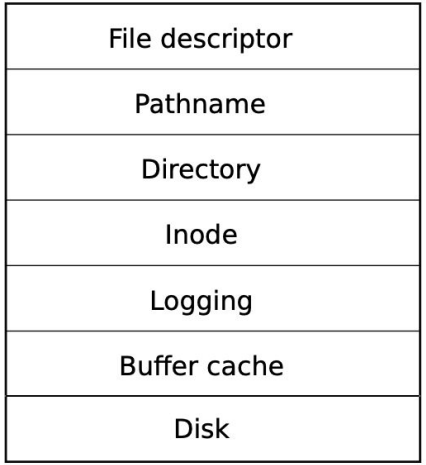
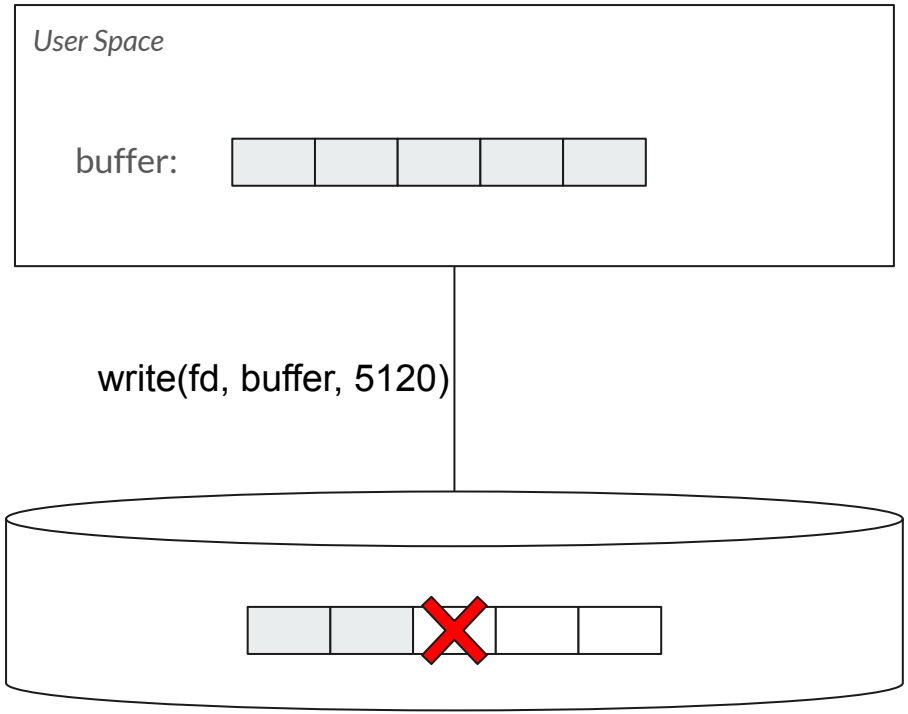


Figure 8.1: Layers of the xv6 file system.

Cómo tolerar fallas

Se cortó la luz!
Quedó el archivo por la mitad.
No hubo tiempo de hacer nada para salvar la situación



Cómo tolerar fallas

Estrategia resumida:

- Escribir el registro primero en un log
- Después escribirlo en el bloque en si
- Si hay fallas re-ejecutar el log para reescribir los bloques

Log en xv6

El log consiste de un header, que contiene informacion de log:

- Cantidad de bloques en el log
- Flag indicando que una transacción está o no en progreso.
- Los bloques que posteriormente se van a escribir en la zona de datos.

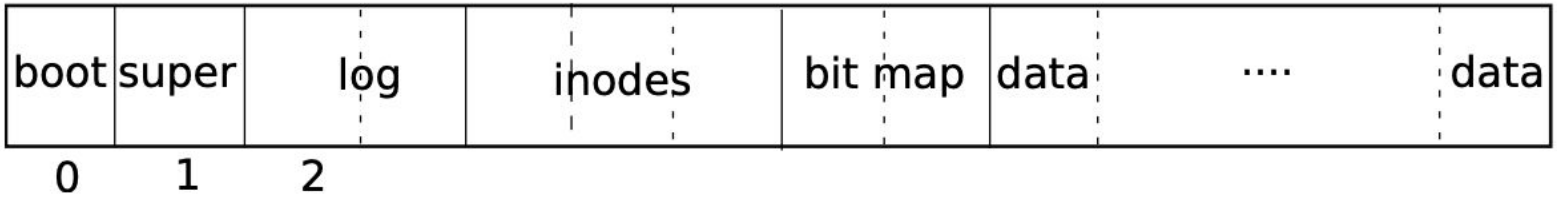


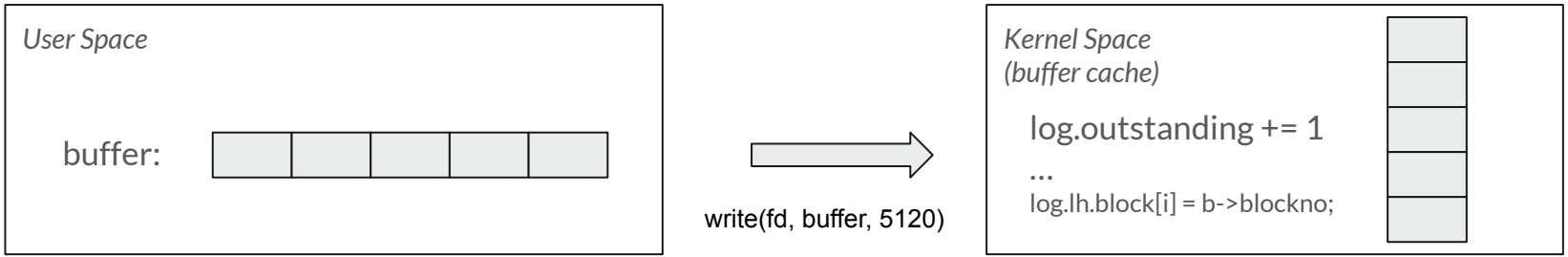
Figure 8.2: Structure of the xv6 file system.

Log en xv6

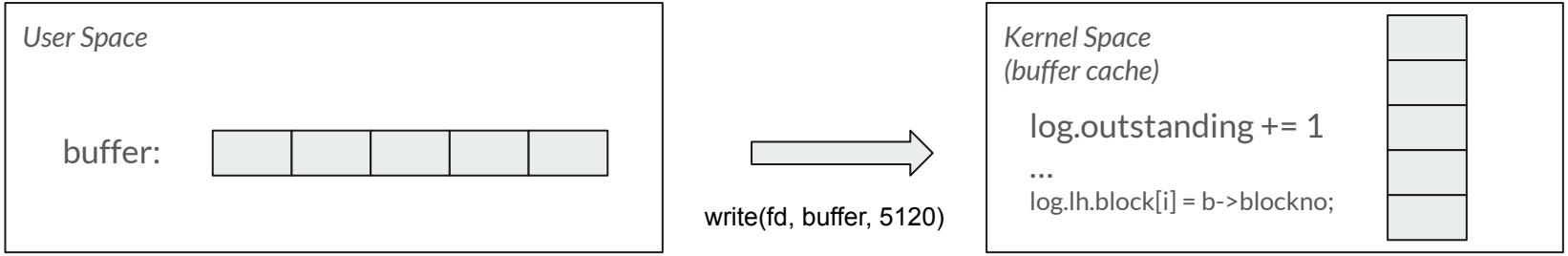
- El log sirve para implementar la abstracción de transacción.
- Una transacción es un conjunto de operaciones (en este caso escrituras de bloques) que se ejecutan completamente o nada. Todo o nada
- En el codigo se marcan con un bloque entre begin_op y end_op
- Begin_op y end_op van a realizar operaciones administrativas para lograr cumplir la premisa d la transaccion

```
begin_op();
...
bp = bread(...);
bp->data[...] = ...;
log_write(bp);
...
end_op();
```

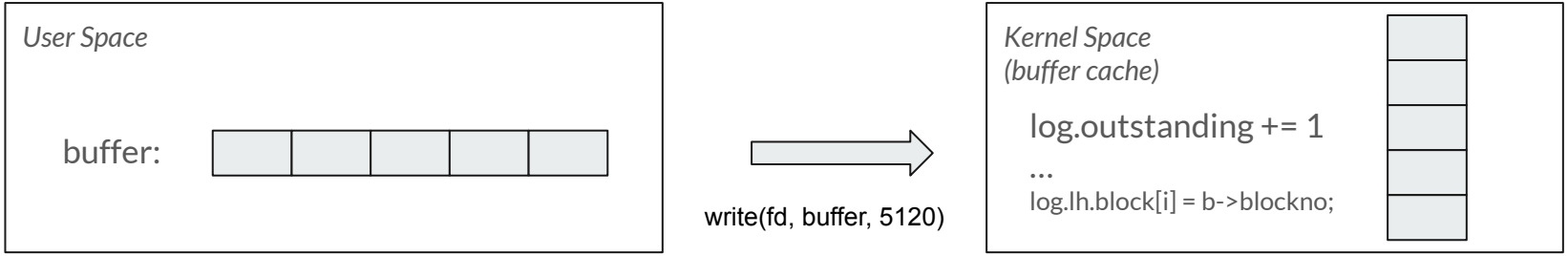
Implementación de transacciones



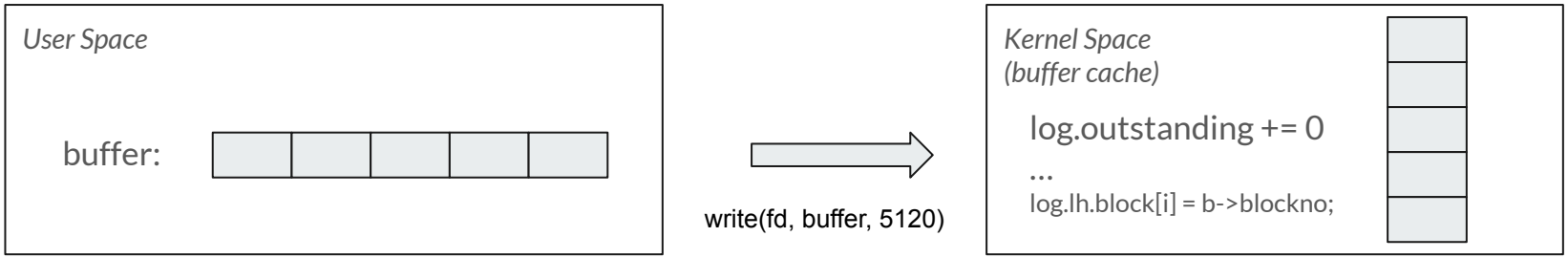
Implementación de transacciones



Implementación de transacciones



Implementación de transacciones



Implementación de transacciones

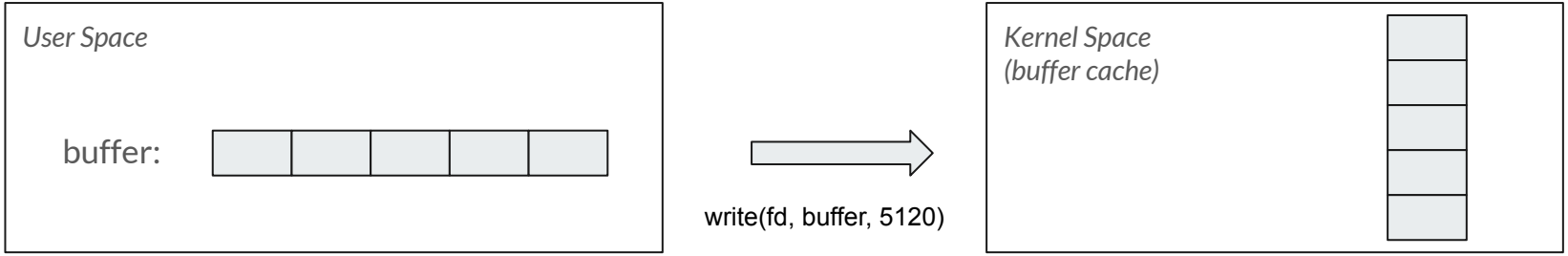
Por que funciona esto?
Falta un componente crucial. El procedimiento de recuperación

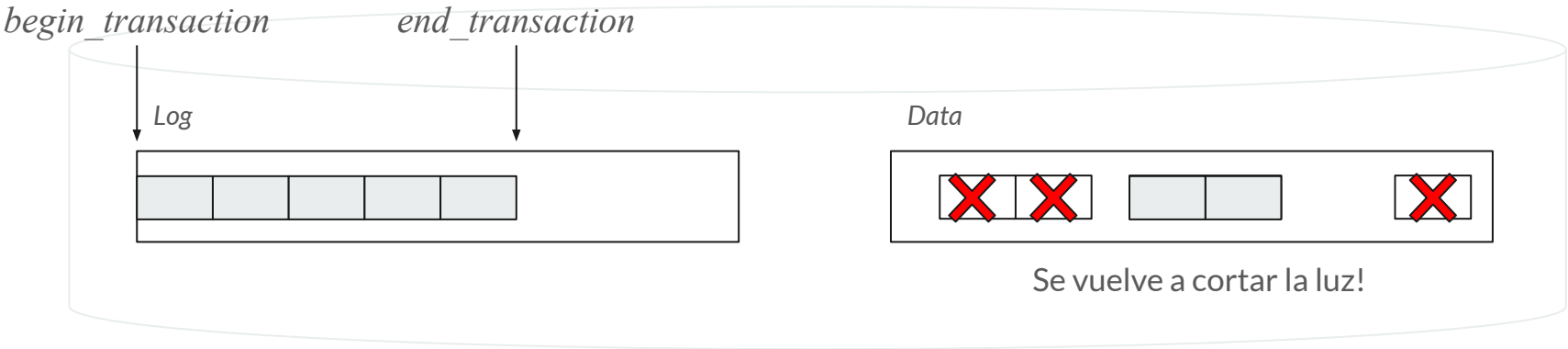
Siempre que se inicia el sistema operativo, se ejecuta la recuperación, antes de iniciar cualquier otro proceso (es decir, no se puede modificar el disco hasta que termina este proceso).

El procedimiento de recuperacion ejecuta lo siguiente:

- Verifica si hay algo en el log.
- Si no hay nada, termina
- Si hay una transacción no comiteada (eg. sin end_transaction) es como si no hubiera existido. Se elimina el contenido del log.
- Si hay una transacción comiteada, se escriben los bloques en el disco. Si los bloques se habian escrito parcialmente no pasa nada, se sobreescriben. Finalmente se elimina la transaccion del log

Implementación de transacciones





Implementación de transacciones