Persistencia

Ejercicio 1
El disco Seagate Exos 7E8 de 8 TiB e interfaz SAS tiene una velocidad de rotación de 7200 RPM, **4.16 ms** de latencia de búsqueda y 215 MiB/s de tasa de transferencia.
(a) Indicar cuántos ms tarda en dar una vuelta completa.

7200 RPM (Revolución por minuto)
7200/60 = 120 RPS (Revolución por segundo)

120 —> 1s
1 —> 0.00833s (tiempo en segundos que tarda en dar una vuelta)

1s —> 1000ms
0.00833s —> 8.33ms (tiempo en milisegundos que tarda en dar una vuelta)

RTA = 8.33 ms.

Con 14400 RPM (doble de 7200 RPM) => 4.166 ms en dar una vuelta

RTA = Si se duplica la RPM, entonces se divide a la mitad el tiempo que tarda en dar una vuelta.

(b) Indicar la **tasa de transferencia** de lectura al azar de bloques de 4096 bytes.

Latencia promedio de rotación = 8.33ms / 2 = **4.165ms** (promedio)
4096/(215*2^20) = 4096/ ≈ 1.817*10^(-5) = 18.17*10^(-6) = 18.17µS = **0.018ms**

T(I/O) = T(seek) + T(rotation) + T(transfer)
T(I/O) = **4.16ms** + **4.16ms** + **0.018ms** = **8,338ms**

R(I/O) = Size of the transfer / T(I/O)
R(I/O) = 4096 / 8.338ms = 4096 / 0.008338 = 488762 B/s = 488 KiB/s = **0.488 MiB/s**

Límites:

- *215 MiB/s max*
- *0.488 MiB/s R(I/O 4K) ~ 215/0.488 = **430x más lento***

215MiB/s transfer => (215MiB/s)/1000 = 0.215MiB/ms?
= 220,160?KiB/ms

*4096 bytes = 4KiB = (4) * MiB / 2^10*
215 MiB - 1 seg
0.0038 MiB (4 KiB) - 0.00001767seg = 0.01767ms

(4KiB/220,160KiB)/s = 0.00001816 s = 0.01816 ms = 18.16 us

Ejercicio 2
Compute el tamaño de la FAT para:
(a) Un diskette de doble cara doble densidad 360 KiB (~1982): FAT12, cluster de 512 bytes.
(b) Un disco duro de 4 GiB (~1998): FAT16, cluster de 4096 bytes.
(c) Un pendrive 32 GiB (~2014): FAT32, cluster de 16384 bytes.

(a)

FAT12 , los numeritos que están dentro de la FAT tienen 12 bits = 1.5 bytes.
12 bits ¿Cuántos objetos distintos puede nombrar? = 2^12 = 4096

Supongamos Cluster 1KiB => 4096 KiB = 4 MiB.
Supongamos Cluster 4KiB => 16384 KiB = 16 MiB.

Cada cluster = 512bytes = 0.5 KiB
Diskette 5.25” 360KiB, cuantos clusters necesito?

360*1024/512 = 720 clusters

Si cada cluster en la FAT ocupa 1.5 bytes ¿Cuánto ocupa TODA la FAT?

720*1.5 bytes = **1080 bytes**

(b)

FAT16 , los numeritos que están dentro de la FAT tiene 16 bits = 2 bytes.
16 bits ¿Cuántos objetos distintos puede nombrar? = 2^16 = 65536

Cada cluster = 4096bytes = 4 KiB
Un disco duro de 4 GiB, ¿cuantos clusters necesito?

4*1073741824 / 4096 = 1048576 clusters

Si cada cluster en la FAT ocupa 2 bytes ¿Cuánto ocupa TODA la FAT?

1048576 * 2 bytes = 2097152 bytes = **2 MiB**

(c)

Un pendrive 32 GiB (~2014): FAT32, cluster de 16384 bytes.

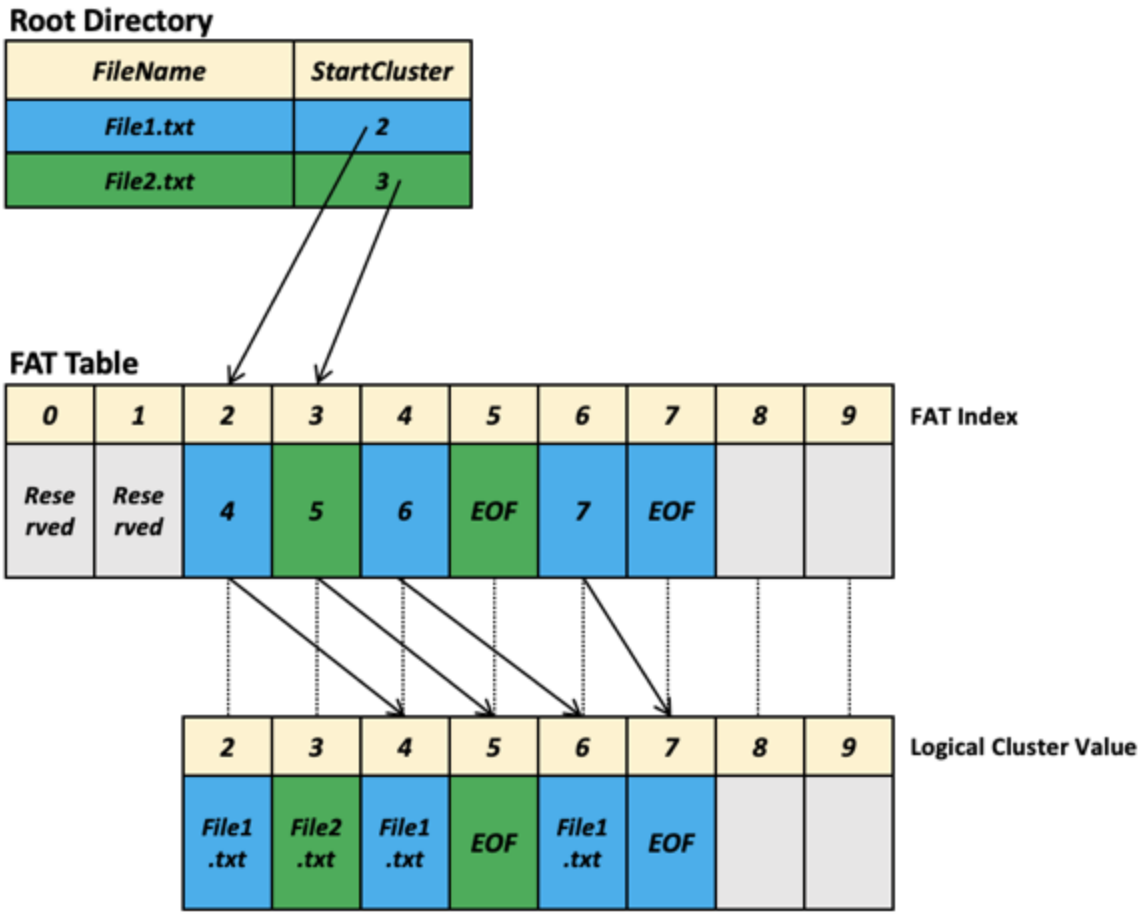
FAT32 , los numeritos que están dentro de la FAT tiene 32 bits = 4 bytes.
32 bits ¿Cuántos objetos distintos puede nombrar? = 2^32 = 4294967296

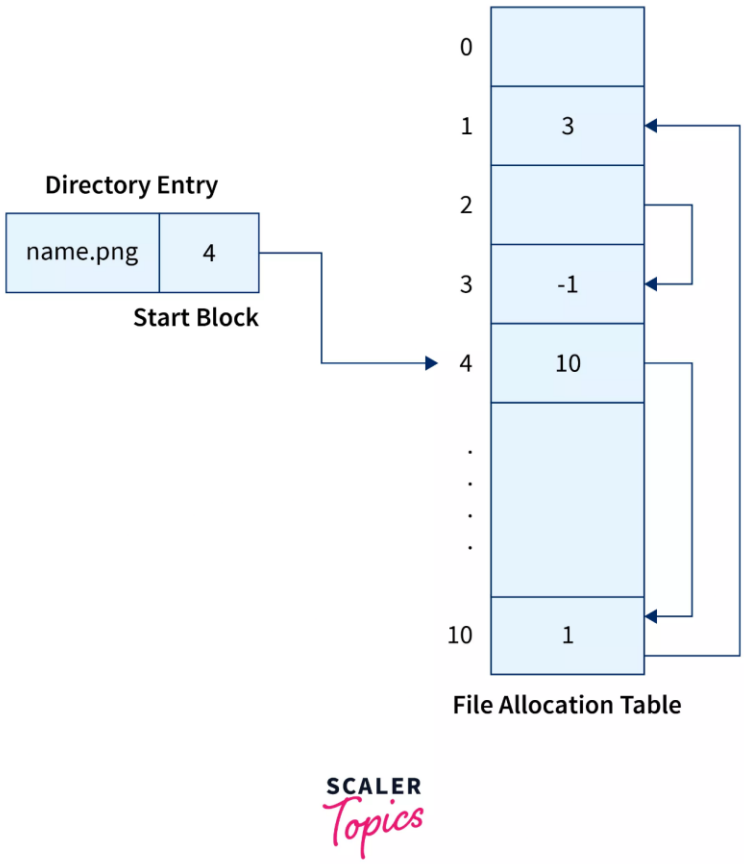
Cada cluster = 16384 bytes = 16 KiB
Un pendrive de 32 GiB, ¿cuantos clusters necesito?

32*1073741824 / 16384 = 2097152 clusters

Si cada cluster en la FAT ocupa 4 bytes ¿Cuánto ocupa TODA la FAT?

2097152 * 4 bytes= 8388608 bytes = **8 MiB**


Root Directory
FileName
StartCluster
File1.txt
2
File2.txt
3
FAT Table
0
1
2
3
4
5
6
7
8
9
FAT Index
Reserved
Reserved
4
5
6
EOF
7
EOF
2
3
4
5
6
7
8
9
Logical Cluster Value
File1.txt
File2.txt
File1.txt
EOF
File1.txt
EOF



Ejercicio 3

El sistema de archivos de xv6 es una estructura *à la* Fast-Filesystem for UNIX (UFS), con parámetros: bloque de 512 bytes, 12 bloques directos, 1 bloque indirecto, índices de bloque de 32 bits.

(a) Calcule el tamaño máximo de un archivo.

(b) Calcule el tamaño de la sobrecarga para un archivo de tamaño máximo.

(c) ¿Se podrían codificar los números de bloque con menos bits? ¿Qué otros efectos produciría utilizar la mínima cantidad de bits?

(a)

Cada índice de bloque es de 32 bits, es decir, 4 bytes.

Bloques directos = 512 Bytes x 12
Cantidad de Bloques Indirectos = 512 Bytes / 4Bytes = 128 Bloques indirectos
Bloques indirectos = 128 * 512 bytes
Max_file_size = Bloques Directos + Bloques Indirectos = 512 * 12 + 512 * 128 = 71680 Bytes = 70KiB

Max Disk Size = 512 * 2^32 = 2^32 * 2^9 = 2^41 = 2TB

Cantidad de archivos posible = 2TB / 71680 bytes = 59,918.62 archivos

(b)
Pregunta cuántos bloques indirectos, doble indirectos, triple indirectos se necesitan para saber dónde están los bloques que almacenan el archivo en sí. Vendría a ser parecido a cuando ocupa el PD y las PTs.
Fíjate que solo hay un indirecto y esto es 512 bytes = 0.5 KiB.

Tamaño de sobrecarga relativo = 0.5KiB/70KiB = 0.71%

Reducimos a 16 bits

(c)
BS = 512 bytes
BD = 12
BI = 1
index_block_size = 16 bits = 2 bytes (cada indice)

Tamaño máximo de archivo:
Cada índice de bloque es de 16 bits, es decir, 2 bytes.
Bloques directos = 512 bytes x 12
Cantidad de Bloques Indirectos = 512 bytes / 2 bytes = 256 Bloques indirectos
Bloques indirectos = 256 * 512 bytes

Bloques Directos + Bloques Indirectos = 512 * 12 + 512 * 256 = 137216 bytes

= 134 KiB

Al tener menos bits de índice, lo que logras es aumentar la cantidad de bloques que puedes redireccionar. Por ende aumentas el tamaño máximo de archivo.

Tamaño de sobrecarga relativo = 0.5KiB/134KiB = 0.37%
Max Disk Size = 512 * 2^16 = 2^16 * 2^9 = 2^25 = 32 MiB
Cantidad de archivos posible = 32 MiB / 0,13 MiB = 244.53 archivos

---

Reducimos a 31 bits

BS = 512 bytes
BD = 12
BI = 1
index_block_size = 31 bits = 4 bytes (cada indice)

Tamaño máximo de archivo:
Cada índice de bloque es de 32 bits, es decir, 3.875 bytes.
Bloques directos = 512 bytes x 12
Cantidad de Bloques Indirectos = 512 bytes / 3.875 bytes = 132.12
Bloques indirectos
Bloques indirectos = 132 * 512 bytes
Bloques Directos + Bloques Indirectos = 512 * 12 + 512 * 132 = 73,728 bytes

Tamaño de sobrecarga:
Tenemos 12 bloques directos de 3.875 bytes.
Tenemos 3.875 byte más para el índice del bloque indirecto.
Finalmente tenemos 132.12 * 3.875 bytes = 511.96

Tamaño de sobrecarga relativo = 0.5KiB/72KiB = %0,69

Max Disk Size: 2^31 = 1TiB
Cantidad de archivos posible = 2^31 / 73,728 **= 29,127.11 archivos**

Reducimos a 8 bits

BS = 512 bytes
BD = 12
BI = 1
index_block_size = 8 bits = 1 bytes (cada indice)

Tamaño máximo de archivo:
Cada índice de bloque es de 8 bits, es decir, 1 byte.
Bloques directos = 512 bytes x 12
Cantidad de Bloques Indirectos = 512 bytes / 1 bytes = 512 Bloques indirectos
Bloques indirectos = 512 * 512 bytes
Bloques Directos + Bloques Indirectos = 512 * 12 + 512 * 512 = 268288 bytes

= **262 KiB**

Tamaño de sobrecarga relativo = 0.5KiB/262KiB = %0,19
Max Disk Size: 2^8 * 2^9 = 2^17 = 128KiB
Cantidad de archivos posible = 128 KiB / 268288 = 0,48 archivos

---

Conclusiones:

- Al reducir los bits de direccionamiento a la mitad, no llegamos a duplicar por poco el tamaño de archivo, sin embargo, reducimos el peso del índice de bloque, lo que genera que en un bloque entren más índice de bloque, pudiendo direccionar más.
- Baja el tamaño de sobrecarga (cuánto ocupan las estructuras).
- Baja el tamaño del disco. (Se redireccionan menos bloques)

- Baja la cantidad de archivos que entran en el disco
  (menos espacio / más pesa un archivo).
- Puede ocurrir que no entre ningún archivo en el disco.
  Por cada bit que le quitamos reducimos a la mitad la cantidad de archivos que entran en el disco y el espacio del disco.

Al reducir la cantidad de bits que pueden indexar los bloques, reducimos la cantidad de bloques direccionables del mismo tamaño (512 bytes) por lo que perdemos mucha capacidad de almacenamiento.

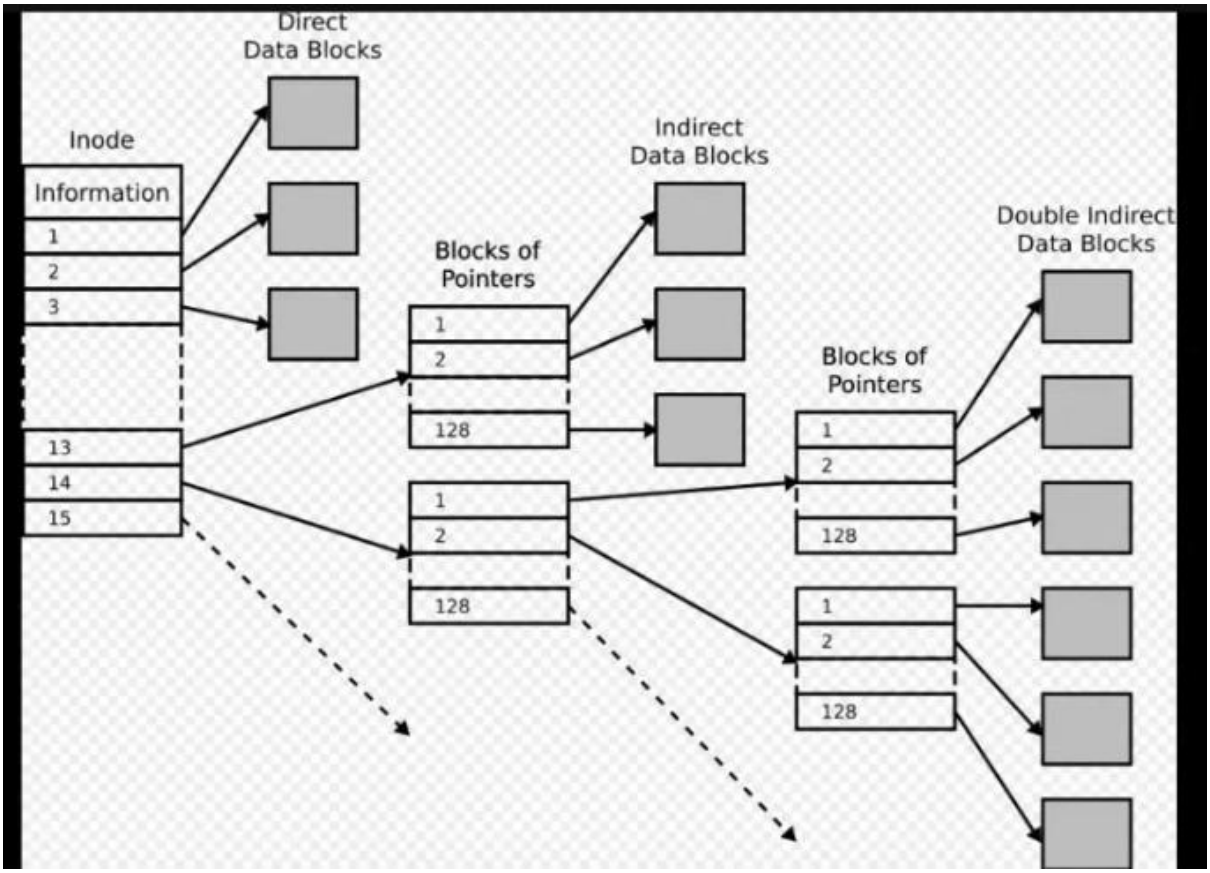
Ejercicio 4

Para un sistema de archivos xv6: bloque de 512 bytes, 12 bloques directos, 1 bloque indirecto, índices de bloque de 32 bits.

(a) Indique qué número de bloque hay que leer para acceder al byte 451, al byte 6200 y al byte 71000.

(b) Dar la expresión matemática que indica a partir del byte que número de bloque hay que acceder.

Se puede usar división por casos, ejemplo positivo(x) = 1 si $0 < x$, 0 si $x \leq 0$.



(a)

BS = 512 bytes
DB = 12
IB = 1
block_index = 32 bits = 4 bytes

byte 451:

451 / 512 = 0 → Se encuentra en el bloque directo 0

byte 6200:

6200 / 512 = 12 → Se encuentra en el primer bloque indirecto

byte 71000:

71000 / 512 = 138 → Se encuentra en el bloque indirecto 138 - 12 = 126

(b)

byte / block_size	si byte / block_size < 12
(byte / block_size) - 12	si byte / block_size ≥ 12

Ejercicio 5

Considere un disco de 16 TiB (244 bytes) con **bloques de 4 KiB** (212 bytes) e **índices de bloque de 32 bits**. Suponga que el sistema de archivos está organizado con i-nodos de **hasta 3 niveles de indirección** y hay **8 punteros a bloques directos.** ¿Cuál es el máximo tamaño de archivo soportado por este sistema de archivos? Justifique su respuesta.

BS = 4 KiB
direct_blocks = 8
indirect_blocks = 3
block_index = 32 bits = 4 bytes

max_indirects_blocks = 4 KiB / 4 bytes = 1024 indirect blocks per block.

Max_file_size = 4KiB(8 + 1024 + 1024^2 + 1024^3) = 4299165728 KiB
= 4100 GiB
= 4 TiB

Ejercicio 6

Considere el sistema de archivos de xv6. Indique qué bloques directos o indirectos hay que crear. Suponga que el i-nodo ya está creado en disco. Puede usar esquemas.
(a) Se crea un archivo nuevo y se agregan 6000 bytes
(b) Se agregan al final 1000 bytes más.

Para un sistema de archivos xv6: bloque de 512 bytes, 12 bloques directos, 1 bloque indirecto, índices de bloque de 32 bits.

(a) En total el sistema de archivos de xv6 puede almacenar:
2^32 * 512 bytes = 2^41 = 2TiB Por ende si entra un archivo de 6mil bytes.

Usando la fórmula:

byte / block_size	si byte / block_size < 12
(byte / block_size) - 12	si byte / block_size ≥ 12

6000 / 512 = 11.719 Es decir se necesitan 11 bloques y un poco más pero no más de 12, así que con los primeros 12 bloques directos nos alcanzaría.

- **Crear y asignar 12 bloques directos**

(b) Si se agregaran 1000 bytes más tendríamos:

7000 / 512 = 13.672 Necesitamos más de 13 bloques así que tenemos que crear 13.672 - 12 = 1.672 → 2. Es decir con un bloque de indirección, creamos el bloque 0 y 1 para almacenar los 7000 bytes.

- **Crear y asignar 1 bloque indirecto para almacenar las referencias.**
- **Crear y asignar 2 bloques de datos adicionales, referenciados desde el bloque indirecto.**

Ejercicio 7

Supongamos que se borró todo el mapa de bits con los bloques en uso de un filesystem tipo UNIX. Explique el procedimiento para recuperarlo.

fsck:

- Al momento de bootear se revisa todo el sistema de archivos, específicamente se escanea los inodos para reconstruir la información de bloques usados.

data journaling:

- Solo se recuperan aquellos bloques cuyas transacciones hayan sido commiteadas pero no llegaron al checkpoint.
- **Limitación:**
    - Solo cubre las operaciones recientes (transacciones registradas en el journal).
    - Si el journal no tiene información sobre bloques anteriores al fallo, esos bloques no se pueden recuperar.

metadata journaling

- Restablece los metadatos correspondientes a aquellas transacciones que se hayan commiteado.
- **Limitación:**
    - El journal no contiene una copia completa del mapa de bits, por lo que no puede reconstruirlo en su totalidad.
    - El journal tiene un tamaño limitado, lo que restringe la cantidad de transacciones que puede almacenar.

Ejercicio 8

Un programa que revisa la estructura del filesystem (fsck) con el método del ejercicio anterior construyó la siguiente tabla de bloques que aparecen en uso vs. bloques que se indican como libres en el bitmap del disco.

In use:	1 0 1 0 0 1 0 1 1 0 1 0 0 1 0
Free:	0 0 0 1 1 1 0 0 0 1 0 1 1 0 1

¿Hay errores? De ser así, ¿resultan importantes?
Ayuda: analice los 4 casos posibles y discuta cada uno.

n use	1	0	1	0
ree	0	1	1	0
Problema?	Un inodo está apuntando al	Ningún inodo apunta al bloque,	Es consistente. No hay problema.	Es consistente. N hay problema.

	bloque, pero el bitmap del mismo figura como libre. Esto puede generar que otro inodo lo tome y reasigne el bloque.	pero el bitmap del mismo está ocupado. Es una “fuga de espacio”.		

Ejercicio 9

Suponiendo

a) no hay nada en la caché de bloques de disco,

b) cada consulta a un i-nodo genera exactamente una lectura de bloque,

c) cada directorio entra en un bloque,

d) cada archivo también entra en un bloque.

Describa la secuencia de lecturas de bloque para acceder a los

archivos:

(a) `/initrd.img`

(b) `/usr/games/moon-buggy`

Archivo	Secuencia de lecturas de bloque	Total de lecturas
`/initrd.img`	Raíz (/) → Bloque del archivo initrd.img	2 bloques
`/usr/games/moon-buggy`	Raíz (/) → usr → games → Bloque del archivo	4 bloques

Ejercicio 10

Un novato en diseño e implementación de sistemas de archivos sugiere que la primera parte del contenido de cada archivo UNIX se almacene en el i-nodo mismo. ¿Es una buena idea? Explique.

Para archivos chicos es razonable ya que reducimos las operaciones de E/S al no tener que leer otros bloques, con leer el inodo nos basta.

Para archivos grandes representa una sobrecarga de trabajo, es más eficiente una lectura secuencial desde los bloques, antes que una por partes para con los datos del archivo. Además, es más pesado al tener que leer todos los datos y luego los punteros, quizás solo necesitabas leer los punteros.

En conclusión, separar claramente los datos y los metadatos permite a los sistemas de archivos optimizar independientemente para diferentes patrones de uso. Mezclar ambos podría complicar el diseño y mantenimiento. Esta implementación puede ser útil cuando se trata de archivos pequeños, sin embargo, para archivos grandes se agrega un nivel de complejidad más.

Ejercicio 11

El campo link count dentro de un i-nodo resulta redundante. Todo lo que dice es cuántas entradas de directorios apuntan a ese i-nodo, y esto es algo que se puede calcular recorriendo el grafo de directorios. ¿Por qué se usa este campo?

Se usa para saber cuando es necesario “borrar un archivo” ya que eliminar archivos o directorios, es equivalente a decir que ese inodo tendrá un enlace menos, al momento de tener 0 enlaces el sistema marca el bitmap del inodo y del bloque como libres.

Ejercicio 12

Ya vimos que las estructuras de datos en disco de los FS mantienen información redundante que debería ser consistente. Piense en más pruebas de consistencia de debería realizar un filesystem checker en un sistema de archivos tipo:
(a) FAT.

- Que cada archivo contenga un puntero a EOF (end of file).
- Que los bloques apuntados tengan datos.
- Que los punteros sean consistentes.
    - Punteros cíclicos o punteros rotos, donde un bloque apunta a un bloque no válido o no existe.
    - Punteros en cadenas de bloques de archivos que se cruzan o apuntan a bloques fuera del rango del sistema de archivos.
    - Bloques no enlazados correctamente (lo que resultaría en una “fragmentación lógica”).

(b) UNIX.

- Chequeo de los bloques indirectos
    - Que sean los suficientes para cubrir el tamaño del archivo actual
    - Bloques indirectos que apuntan a bloques inválidos o que no existen.
    - Punteros indirectos que apuntan a bloques vacíos o fuera de rango, lo que resulta en un acceso de datos fallido.

Ejercicio 13
Explique por qué no se pueden realizar enlaces duros de directorios en un sistema de archivos UNIX.

```
$ mkdir dir1
$ ln dir1 dir2
ln: ‘dir1’: hard link not allowed for directory
```

Ayuda: pensar en el campo link count y en dos directorios que se autoreferencian.

Supongamos que funciona, y hacemos ln dir1 dir2
dir1 tiene solo las dos entradas especiales, `.` que se referencia así mismo y `..` que referencia al padre. Ahora dir2 que también contiene estas dos entradas, apunta a dir1. Por lo que el contador de enlaces de dir1 aumenta, y a su vez, dir2 también tendría un enlace a dir1. Creando una referencia recursiva.

Ejercicio 14
Explique la diferencia entre un log-filesystem y un journaling-filesystem.

Data journaling:
1. Journal write: Write the contents of the transaction (containing TxB and the contents of the update) to the log; wait for these writes to complete.
2. Journal commit: Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction is now committed.
3. Checkpoint: Write the contents of the update to their final locations within the file system.
4. Free: Some time later, mark the transaction free in the journal by updating the journal superblock.

Metadata journaling:
1. Data write: Write data to final location; wait for completion (the wait is optional; see below for details).
2. Journal metadata write: Write the begin block and metadata to the log; wait for writes to complete.
3. Journal commit: Write the transaction commit block (containing TxE) to the log; wait for the write to complete; the transaction (including data) is now committed.
4. Checkpoint metadata: Write the contents of the metadata update to their final locations within the file system.
5. Free: Later, mark the transaction free in journal superblock

One at time:

some file systems do not commit each update to disk one at a time (e.g., Linux ext3); rather, one can buffer all updates into a global transaction. In our example above, when the two files are created, the file system just marks the in-memory inode bitmap, inodes of the files, directory data, and directory inode as dirty, and adds them to the list of blocks that form the current transaction. When it is finally time to write these blocks to disk (say, after a timeout of 5 seconds), this single global transaction is committed containing all of the updates described above. Thus, by buffering updates, a file system can avoid excessive write traffic to disk in many cases.

LFS:

The new type of file system Rosenblum and Ousterhout introduced was called LFS, short for the Log-structured File System. When writing to disk, LFS first buffers all updates (including metadata!) in an in memory segment; when the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk. LFS never overwrites existing data, but rather always writes segments to free locations. Because segments are large, the disk (or RAID) is used efficiently, and performance of the file system approaches its zenith.

The large chunk of updates LFS writes at one time is referred to by the name of a segment. Although this term is over-used in computer systems, here it just means a large-ish chunk which LFS uses to group writes. Thus, when writing to disk, LFS buffers updates in an in-memory segment, and then writes the segment all at once to the disk. As long as the segment is large enough, these writes will be efficient

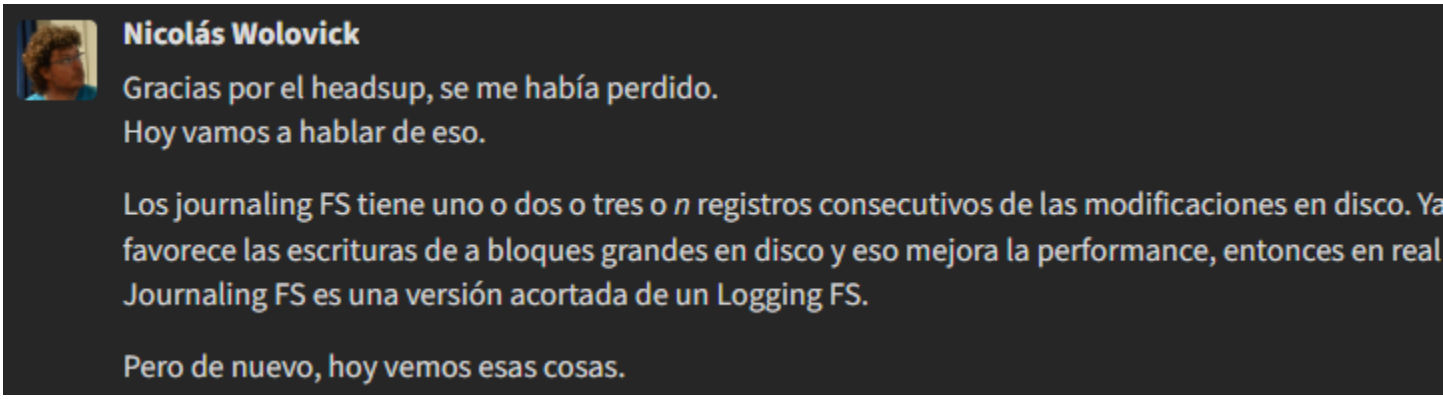
- **Journaling File System** (JFS) **hace un log de las actualizaciones** antes de escribirlas en el disco. El log puede ser de **metadata** (solo estructuras de datos) o **data** (incluyendo datos). En **data journaling**, primero se escribe el log con los cambios, y luego se actualizan los

datos finales. En **metadata journaling**, se actualiza primero los datos en su ubicación final y luego se escribe el log con la metadata.
- **Log-structured File System** (LFS) **acumula todas las actualizaciones en un segmento de memoria** y las escribe todas juntas en el disco cuando el segmento se llena, usando una escritura secuencial grande. No sobrescribe, sino que escribe en nuevas ubicaciones del disco.

Diferencia clave:

- **JFS**: Es más pequeño, va escribiendo cambios de forma individual (log + actualización en el disco).
- **LFS**: Acumula cambios en un segmento y los escribe todos juntos de una sola vez.

Sí, el journaling es una **versión más acotada** del log-structured file system, donde el log es más pequeño y las actualizaciones no se hacen todas a la vez, sino de forma incremental.



Ejercicio 15

Enumere y explique brevemente las tres (3) capas en las que se estructura un sistema de archivos. Dar dos (2) ejemplos que muestran la conveniencia de esta división.

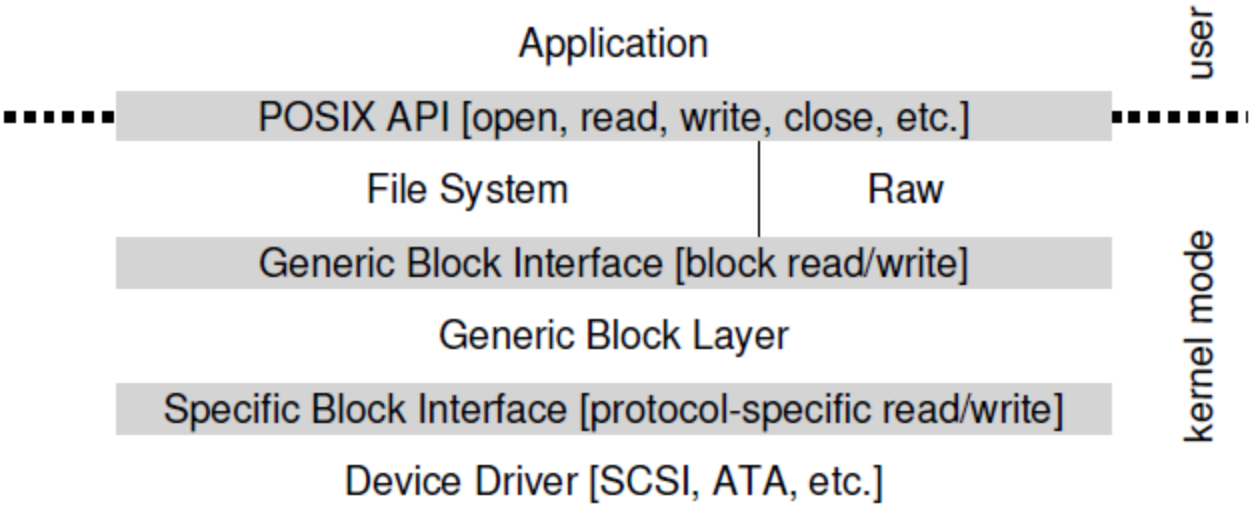


Figure 36.4: **The File System Stack**

1. **Capa de syscalls**: Interfaz entre el usuario y el sistema de archivos (ej., `open()`, `read()`, `write()`).
2. **Capa de VFS (Virtual File System)**: Capa de abstracción que gestiona las operaciones de archivo y las traduce a acciones específicas de sistemas de archivos. Permite que el sistema operativo trabaje con diferentes sistemas de archivos de forma unificada.
3. **Capa de Device Driver**: Interactúa con el hardware de almacenamiento y gestiona la lectura/escritura a nivel físico en el dispositivo de almacenamiento.

Ejemplo:

1. El **usuario** realiza una llamada a `open()` para abrir un archivo.
2. La **capa VFS** traduce esa solicitud de apertura de archivo a una operación específica sobre el sistema de archivos (por ejemplo, ext4).

3. El **driver del dispositivo** lee o escribe los bloques de datos del archivo desde el dispositivo de almacenamiento físico (como un disco duro o un SSD).

Ejercicio 16

¿Verdadero o Falso? Explique.

(a) La entrada/salida programada (por interrupciones) y el DMA liberan a la CPU de hacer pooling a los dispositivos. V

- En lugar de que la CPU esté haciendo consultas constantes (polling) al dispositivo para ver si está listo para enviar o recibir datos, el dispositivo envía una interrupción a la CPU cuando necesita atención.

(b) Los sistemas de archivos tipo UNIX tienen soporte especial para que cuando un directorio crece en cantidad de archivos y no entra más en un bloque, pida más bloques. F

- Es un archivo más. Ya está implementado.

(c) Es posible establecer enlaces duros (hardlinks) entre archivos de diferentes particiones. F

- No porque interseccionan los valores de los inodos, es decir, los inodos son específicos para cada sistema de archivos (para cada partición). Se puede colgar del árbol como una nueva partición pero los inodos no se comparten. Los sistemas de archivos no necesariamente son los mismos para dos particiones diferentes.

(d) Los dispositivos de I/O se dividen en I/O mapped y memory mapped. T

Son dos métodos para comunicarse con los dispositivos:

- El primero son instrucciones privilegiadas para comunicarse con ellos

- El segundo, el hardware crea registros disponibles como si fueran posiciones de memoria, se hace load o store de un registro particular en la dirección indicada.

(e) Un disco duro en formato UFS2 puede no estar lleno y aún así no poder almacenar más archivos. V

- Creo archivos de tamaño 0 hasta llenarlo.

- Si los bits para direccionar bloques son muy pocos, lo que puede suceder es que ningún archivo entre en el espacio de almacenamiento. **No se puede modificar un sistema de archivos.**

- Puede suceder que tenemos espacio, pero no quedan màs inodos.

(f) El tamaño de un archivo no es un atributo realmente necesario en los i-nodos, se puede calcular a partir de los bloques ocupados (FAT o UFS). F

- En FAT si es innecesario
- En UFS si o si los necesitas, es muy costoso y no hay forma de saber dónde termina un archivo (EOF)
- Esta información es crucial para poder realizar operaciones de manera más eficiente, sin tener que estar calculando el tamaño en bloques todo el tiempo.

(g) El uso de espacio en disco es siempre mayor o igual a la longitud del archivo. V

- Siempre el uso en disco va a ser mayor igual a la longitud del archivo.

(h) Un archivo borrado no se puede recuperar. F

- Si se puede recuperar porque solo se elimina el puntero que apunta a él, en algún momento esa información es reemplazada por nueva, pero mientras tanto el archivo sigue ahí.

(i) Los sistemas de archivos como FAT o UFS no sufren de fragmentación externa. F

- En FAT o UFS puede suceder que liberemos un archivo y dejamos espacios libres no contiguos que generan fragmentación externa.

(j) Los sistemas de archivos modernos como EXT4 o ZFS tienen problemas de escalabilidad en la cantidad de entradas de un directorio. F

- A medida que más archivos tengo en un directorio, más tarda la búsqueda de archivos. Es lineal respecto a la búsqueda. Eso se soluciona con estructuras nuevas.

(k) La syscall truncate sirve para recortar el tamaño del archivo. V

- La función de la llamada al sistema (`syscall`) `truncate` es efectivamente recortar o ajustar el tamaño de un archivo.
- man truncate: Shrink or extend the size of each FILE to the specified size

---

Ejercicio 17
¿Por qué resulta conveniente en una estructura tipo UFS tener un bitmap de i-nodos libres? Exactamente lo mismo se puede conseguir utilizando un bit especial en la tabla de i-nodos que indique si está libre o no. Discuta.

La idea es no tener dispersa la información. En general tanto la RAM como el HDD y SDD tienen el mejor desempeño cuando leen de a bloques y todo el bloque leído se usa para el cómputo.

Situación un bit dentro del inodo para indicar si está libre u ocupado:

- Tenemos que leer toda la inode table del disco y de cada inodo (128 bytes para poner un ejemplo) leído solo usamos 1 bit.
  Desperdiciamos más del 99% de la información que trajimos.

Situación un bit por inodo en una tabla aparta, el inode bitmap:

- Leemos todos los bloques que componen la inode bitmap todos los bits son necesarios, el desperdicio de ancho de banda es del 0%.

Ejercicio 18

Los FS con asignación de espacio no-contiguo (FAT, UFS, etc.) suelen tener un desfragmentador para que todos los bloques de un archivo estén contiguos y el seek-time no impacte tanto. ¿Cómo se podría mejorar la estructura de datos que mantiene la disposición o secuencia de bloques de FAT o UFS para aprovechar que siempre se intenta que la secuencia bloques sea una secuencia creciente?

Cuando un archivo se crea, guarda un poco más de espacio por si este archivo quiere crecer, este podrá hacerlo de manera contigua.