

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

Implementación de sistemas de archivos, FFS y journaling

Del inodo al bloque, y cómo sobrevivir a un apagón

Gerson Garrido

Semana 14

Hoy

Sesión A · Implementar un sistema de archivos

1. VSFS: superbloque, bitmaps, inodos, datos
2. Punteros directos, indirectos y extents
3. Recorrer una ruta: cuántas E/S cuesta
4. Caché y write buffering
5. El formato del disco de xv6 y `bmap()`

Sesión B · FFS y consistencia ante fallos

6. FFS: grupos de cilindros y localidad
7. ¿Qué pasa si se va la luz? `fsck`
8. Journaling de datos y de metadatos
9. Revocación, soft updates, copy-on-write
10. El log de xv6: `begin_op`, `end_op`, `commit`

Lectura

OSTEP cap. 40 *File System Implementation*, cap. 41 *Locality and The Fast File System* y cap. 42 *Crash Consistency: FSCK and Journaling*. Código: `code/week-14`.

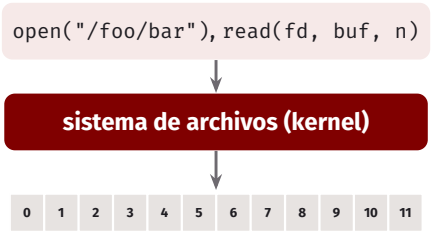
SECCIÓN 01

Un sistema de archivos simple

01

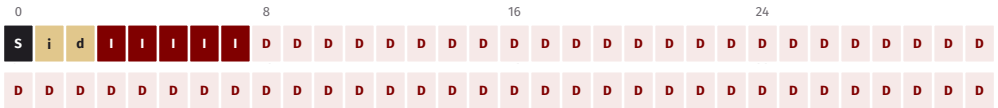
De la API al disco

- La semana 13 vimos la **API**: `open`, `read`, `write`, `link`, `fsync`...
- Hoy abrimos la caja. Un sistema de archivos es software puro: solo tiene un arreglo de bloques.
- Dos preguntas guían todo el diseño:
  1. ¿Qué **estructuras de datos** hay en el disco?
  2. ¿Qué **métodos de acceso** las recorren en cada llamada?



¿qué bloque leo? ¿cuál escribo? ¿en qué orden?

VSFS: repartir 64 bloques



Superbloque (0), bitmap de inodos (1) y de datos (2), tabla de inodos (3–7: 80 inodos de 256 B) y 56 bloques de datos (8–63). Implementado en `code/week-14/vsfs.c`.

El inodo: todo lo que el sistema sabe de un archivo

Campo	Para qué
tipo	archivo, directorio…
tamaño	bytes del archivo
`nlink`	cuántos nombres lo apuntan
owner, permisos	control de acceso
tiempos	acceso, modificación
punteros	dónde están los datos

Es lo que devuelve `stat` (semana 13). El *nombre* no está aquí: está en el directorio.

Número de inodo *(i-number)*

Cada inodo se identifica con un número. Con él, el sistema **calcula** dónde está en el disco: la tabla de inodos es un arreglo.

Metadatos y datos

El inodo es **metadato**: datos sobre los datos. Los bitmaps, los directorios y el superbloque también. El journaling usará esta distinción.

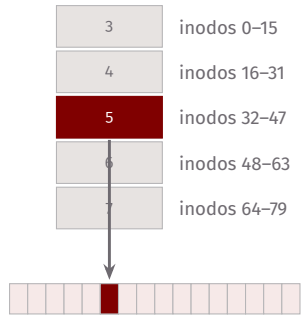
Ejemplo: ¿dónde está el inodo 37?

VSFS: inodos de 256 B, la tabla empieza en el bloque 3, sectores de 512 B.

1. Byte dentro de la tabla: $37 \cdot 256 = 9472$
2. Bloque de la tabla: $\lfloor 9472/4096 \rfloor = 2 \Rightarrow$ bloque $3 + 2 = 5$
3. Byte dentro del bloque: $9472 \bmod 4096 = 1280$
4. Sector: $\lfloor (5 \cdot 4096 + 1280)/512 \rfloor = 42$

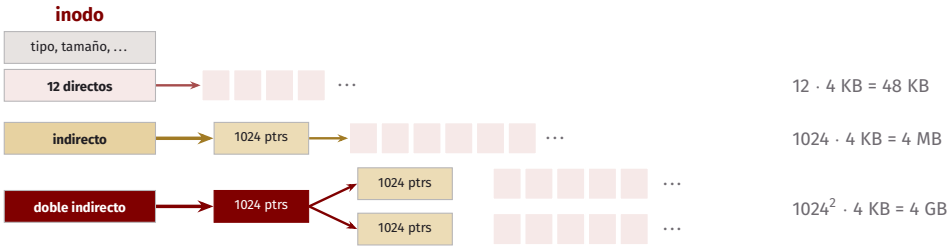
Solo aritmética

No hay búsqueda: `./vsfs img stat 37` imprime `bloque 5, byte 1280`.



bloque 5: 16 inodos; el 37 es el sexto

Punteros: directos, indirectos, doblemente indirectos



Un **árbol desequilibrado** a propósito: los archivos chicos (la mayoría) no pagan nada extra; los grandes, unas lecturas más. ext2/ext3 usan este esquema.

Ejemplo resuelto: el tamaño máximo de un archivo

Bloques de 4 KB, punteros de 4 B, 12 directos y un indirecto.

1. Punteros por bloque: $4096/4 = 1024$
2. Bloques alcanzables: $12 + 1024 = 1036$
3. Tamaño: $1036 \cdot 4\ \text{KB} = 4144\ \text{KB} \approx 4\ \text{MB}$
4. Con un doble indirecto se suman 1024^2 bloques: ≈ 4 GB. Con un triple, ≈ 4 TB.

bash

```
$ ./bmapcalc -max 4096 4 12
1024 punteros por bloque
12 + 1024 = 1036 bloques = 4144 KB
```

Mediciones reales

La mayoría de los archivos es pequeña, pero la mayoría de los *bytes* está en archivos grandes.

Directorios en disco y alternativas a los punteros

Un directorio es un archivo con una lista de pares (nombre, número de inodo):

inodo	reclen	strlen	nombre
5	12	2	`.`
2	12	3	`..`
14	12	6	`notas`
0	16	9	`tarea1.c` (borrada)
31	28	25	`un_nombre_bastante_largo`

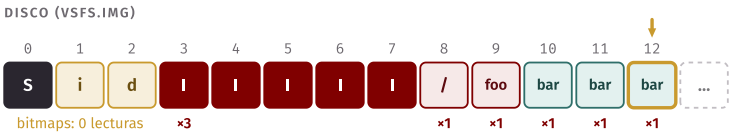
Borrar pone el inodo en 0; `reclen` permite reutilizar el hueco. XFS y ext4 usan árboles o hashes para directorios grandes.

Extents: un puntero y una longitud.



- Compacto si el archivo es contiguo; ext4, XFS y btrfs los usan.
- Lista enlazada de bloques: FAT. Simple, pero un acceso aleatorio recorre toda la cadena.

Recorrer una ruta: `open("/foo/bar")` sin caché



Comprobarlo con `vsfs`

```bash
$ ./vsfs -v /tmp/vsfs.img read /foo/bar
open()
    leer      bloque  3  (inodos 0-15)
    leer      bloque  8  (datos)
  en el inodo 2: "foo" -> inodo 3
    leer      bloque  3  (inodos 0-15)
    leer      bloque  9  (datos)
  en el inodo 3: "bar" -> inodo 4
    leer      bloque  3  (inodos 0-15)
    -> 5 lecturas, 0 escrituras
```

- **5 lecturas** solo para abrir: dos por directorio, más el inodo final.
- Crecen con la profundidad de la ruta y con el tamaño de los directorios.
- Leer no toca los bitmaps: solo se usan para *reservar*.

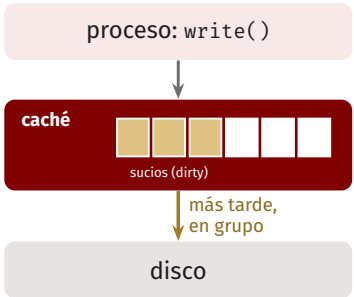
Crear `/foo/bar` y escribir 12 KB: la cuenta de E/S

	bitmap inodos	bitmap datos	inodos (bloque 3)	datos de /	datos de foo	datos de bar	
`create()`	L E		L L L E L E	L	L L E		8 L + 4 E
`write()` 0		L E	L E			E	2 L + 3 E
`write()` 1		L E	L E			E	2 L + 3 E
`write()` 2		L E	L E			E	2 L + 3 E

Cada escritura que reserva un bloque cuesta 5 E/S: leer y escribir el bitmap, escribir el dato, leer y escribir el inodo. Total: 14 L + 13 E para 12 KB. El bloque 3 se lee 7 veces. Por eso ningún sistema real funciona sin una **caché** en memoria.

Caché de páginas y write buffering

- **Lecturas**: los bloques usados quedan en memoria. Los directorios de una ruta frecuente casi nunca se leen del disco.
- Antes: caché *fija* (~10 % de la RAM). Hoy: una caché de páginas unificada *(page cache)* que comparte la memoria con la memoria virtual.
- **Escrituras**: se retrasan 5–30 s *(write buffering)*. Así se *agrupan* (un solo bitmap para muchos cambios), se *ordenan* y a veces se *evitan* (un archivo temporal que se borra antes).



El precio

Si se va la luz, se pierde lo que estaba en la caché. Quien no lo acepte usa `fsync` (semana 13) o `O_DIRECT`.

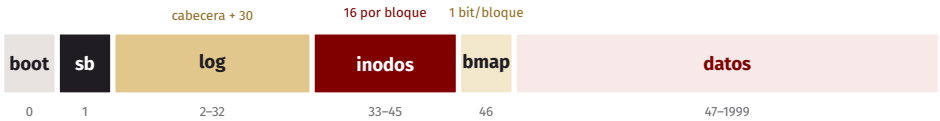
SECCIÓN 02

El sistema de archivos de xv6

02



El formato del disco de xv6




```
bash
$ ./mkfs/mkfs fs.img README user/_cat ...
nmeta 47 (boot, super, log blocks 31, inode blocks 13, bitmap blocks 1) blocks 1953 total 2000
```

- Bloques de **1 KB** (`BSIZE`); 2000 bloques en total (`FSSIZE`).
- 200 inodos de 64 B; raíz = inodo **1**; el 0 no se usa.
- No hay bitmap de inodos: un inodo está libre si su `type` es 0.
- Lo mismo calcula `./bmapcalc` con las fórmulas de `mkfs/mkfs.c`.

`kernel/fs.h`: el formato en C

kernel/fs.h (sin comentarios)

```
struct superblock {
  uint magic;
  uint size;
  uint nblocks;
  uint ninodes;
  uint nlog;
  uint logstart;
  uint inodestart;
  uint bmapstart;
};

#define NDIRECT   12
#define NINDIRECT (BSIZE / sizeof(uint))
#define MAXFILE   (NDIRECT + NINDIRECT)
```

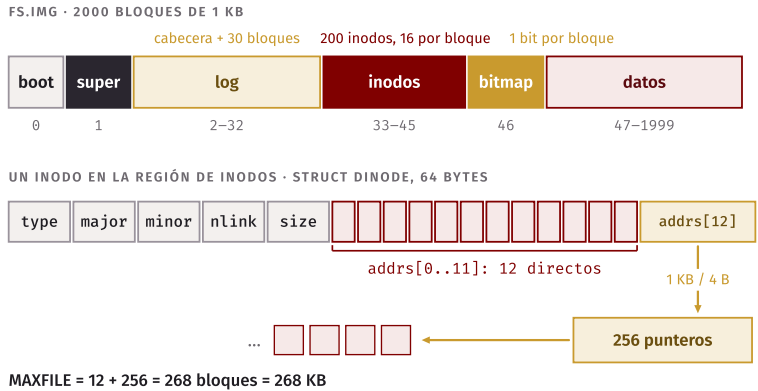
kernel/fs.h (sin comentarios)

```
struct dinode {
  short type;
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT + 1];
};

#define IPB (BSIZE / sizeof(struct dinode))
#define IBLOCK(i, sb) ((i) / IPB + sb.inodestart)
#define BPB (BSIZE * 8)
#define BBLOCK(b, sb) ((b) / BPB + sb.bmapstart)
```

- $4 \cdot 2 + 4 + 13 \cdot 4 = 64$ B por inodo $\Rightarrow$ `IPB` = 16. `NINDIRECT` = 1024/4 = 256 $\Rightarrow$ `MAXFILE` = 268 bloques = 268 KB.
- `IBLOCK` es la cuenta del inodo 37, en una macro: el inodo 25 está en el bloque $25/16 + 33 = 34$. `BBLOCK` hace lo mismo con el bitmap.

El disco y el inodo de xv6



MAXFILE = 12 + 256 = 268 bloques = 268 KB

Un `struct dinode` de 64 B: 16 por bloque de 1 KB. El último puntero lleva a un bloque de 256 punteros.

`bmap()`: del bloque lógico al bloque del disco

kernel/fs.c

```
static uint
bmap(struct inode *ip, uint bn)
{
  uint addr, *a;
  struct buf *bp;

  if (bn < NDIRECT) {
    if ((addr = ip->addrs[bn]) == 0) {
      addr = balloc(ip->dev);
      if (addr == 0)
        return 0;
      ip->addrs[bn] = addr;
    }
    return addr;
  }
  bn -= NDIRECT;
```

kernel/fs.c (sigue; luego panic)

```
  if (bn < NINDIRECT) {
    if ((addr = ip->addrs[NDIRECT]) == 0) {
      addr = balloc(ip->dev);
      if (addr == 0)
        return 0;
      ip->addrs[NDIRECT] = addr;
    }
    bp = bread(ip->dev, addr);
    a = (uint *)bp->data;
    if ((addr = a[bn]) == 0) {
      addr = balloc(ip->dev);
      if (addr) {
        a[bn] = addr;
        log_write(bp);
      }
    }
    brelse(bp);
    return addr;
  }
```

`bn` = `off / BSIZE`, el bloque *lógico*. Devuelve el bloque del *disco*; si no existe, lo reserva.

Ejemplo: leer el byte 70 000 de un archivo en xv6



1. Bloque lógico: $\lfloor 70\,000/1024 \rfloor = 68$; byte $70\,000 - 68 \cdot 1024 = 368$.
2. $68 \geq 12$ (`NDIRECT`): está en el indirecto, entrada $68 - 12 = 56$ (bytes 224–227 del bloque indirecto).
3. Leer el inodo: `addrs[12]` = 812 (números de bloque de ejemplo)
4. Leer el bloque 812: la entrada 56 dice 1290.
5. Leer el bloque 1290 y copiar desde el byte 368: 3 lecturas sin caché.

`balloc()`: buscar un bit libre

kernel/fs.c: balloc(uint dev)

```
bp = 0;
for (b = 0; b < sb.size; b += BPB) {
  bp = bread(dev, BBLOCK(b, sb));
  for (bi = 0; bi < BPB && b + bi < sb.size; bi++) {
    m = 1 << (bi % 8);
    if ((bp->data[bi / 8] & m) == 0) { // Is block free?
      bp->data[bi / 8] |= m;           // Mark block in use
      log_write(bp);
      brelse(bp);
      bzero(dev, b + bi);
      return b + bi;
    }
  }
  brelse(bp);
}
printk("balloc: out of blocks\n");
return 0;
```

- Recorre el bitmap **desde el bloque 0**: el primero libre gana *(first fit)*. Sin ninguna idea de localidad.
- `log_write` en lugar de `bwrite`: el cambio va al **log**. Lo vemos al final.
- `bzero` llena de ceros el bloque nuevo, también por el log.
- `ialloc` hace lo mismo con la tabla de inodos: busca un `dinode` con `type == 0`.

Adelanto de P5: archivos grandes

- Hoy un archivo de xv6 llega a $12 + 256 = 268$ bloques: 268 KB.
- P5 pide un bloque **doblemente indirecto** para llegar a $11 + 256 + 256 \cdot 256 = 65\,803$ bloques.
- Restricción: el `dinode` sigue midiendo 64 B. Un puntero directo cede su lugar.
- Dónde se trabaja: `bmap()` e `itrunc()` en `kernel/fs.c`; las constantes en `kernel/fs.h` y `kernel/file.h`.

Laboratorio de esta semana

Dibujen el formato del disco de xv6 con los números de `mkfs` y el `dinode` con sus 13 punteros. Marquen dónde iría el doble indirecto y qué cuentas tendría que hacer `bmap()` para un bloque lógico grande.

Ojo

`itrunc()` también debe liberar los bloques nuevos; si no, el disco se llena con `bigfile`.

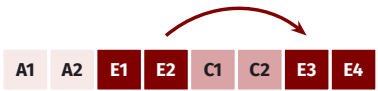
SECCIÓN 03

FFS: la localidad importa

03

El sistema de archivos de Unix original era lento

- Formato: superbloque, inodos, datos. Igual que VSFS.
- Trataba al disco como memoria de acceso aleatorio: el inodo lejos de sus datos, bloques de 512 B.
- Con el uso, el espacio libre se **fragmenta**.
- Resultado: cerca del 2 % del ancho de banda del disco.



E (4 bloques) queda partido en dos

FFS: grupos de cilindros



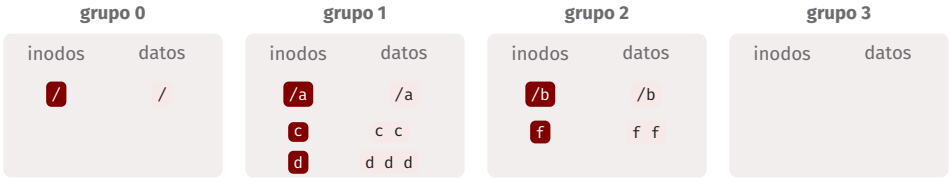
un grupo = pistas vecinas

- FFS (*Fast File System*, Berkeley, 1984) divide el disco en grupos de cilindros *(cylinder groups)*: pistas vecinas, donde moverse cuesta poco.
- Hoy los discos no revelan su geometría: ext2/3/4 usan **grupos de bloques** *(block groups)*, rangos consecutivos de bloques.



cada grupo es un VSFS en miniatura, con copia del superbloque

Política de ubicación: lo relacionado, junto



- **Directorios**: a un grupo con pocos directorios y muchos inodos libres. Así se reparten (`/a` y `/b` en grupos distintos).
- **Archivos**: sus bloques en el mismo grupo que su inodo, y el inodo en el mismo grupo que su directorio. `/a/c` y `/a/d` quedan juntos.
- La apuesta: los archivos de un mismo directorio se usan juntos (compilar un proyecto, `ls -l`). Las mediciones lo confirman.

La excepción de los archivos grandes



repartido en trozos *(chunks)*,
uno por grupo

FFS: los 12 bloques directos van en el grupo del inodo; cada bloque indirecto, con todo lo que apunta, en otro grupo.

¿De qué tamaño los trozos? Cada salto cuesta un posicionamiento. Disco: 8 ms de posicionamiento, 100 MB/s.

- 50 % del ancho de banda: transferir 8 ms por cada 8 ms de salto:
  $100\ \text{MB/s} \cdot 8\ \text{ms} = 800\ \text{KB}$.
- 90 %: transferir $9 \times 8 = 72$ ms: 7,2 MB.

Amortización

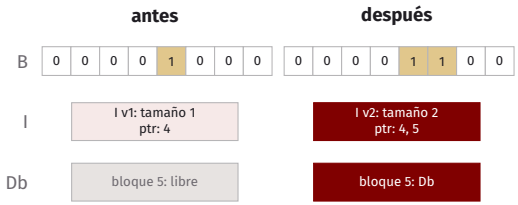
Pagar un costo fijo (el salto) por mucho trabajo útil. Como la transferencia mejora más rápido que la mecánica, los trozos deben crecer.

SECCIÓN 04

Consistencia ante fallos

04

Una escritura, tres bloques



- Agregar un bloque a un archivo cambia **tres** bloques: el bitmap B, el inodo I y el dato Db.
- El disco escribe un bloque a la vez. Con write buffering, los tres esperan juntos en la caché.
- Si se va la luz, o el kernel entra en pánico, a mitad de camino…¿en qué estado queda el disco?

El problema de la consistencia ante fallos ***(crash consistency)***

Pasar el disco de un estado consistente a otro de forma **atómica**, aunque el hardware solo garantice escrituras de un sector.

Los seis casos de un fallo

Llegó al disco	Qué ve el sistema después	¿Problema?
solo Db	nadie apunta al bloque; el bitmap lo da por libre	no: se perdió la escritura
solo I	I apunta a basura, y el bitmap da ese bloque por libre	sí: basura e inconsistencia
solo B	bloque marcado como usado, sin owner	sí: fuga de espacio
I y B	metadatos de acuerdo, pero el dato es basura	sí, y nadie lo nota
I y Db	dato correcto, pero el bitmap lo da por libre	sí: se reasignará
B y Db	bloque usado, sin inodo que lo apunte	sí: fuga de espacio

Lo que queremos: o llegan **las tres**, o **ninguna**. Dos respuestas históricas: dejar que pase y arreglar después (`fsck`), o escribir primero una nota (journaling).

`fsck`: revisar todo al arrancar

1. **Superbloque**: ¿tiene sentido?
2. **Bloques libres**: recorrer todos los inodos y sus punteros, reconstruir el bitmap. Los inodos mandan.
3. **Inodos**: tipo válido; si no, se liberan.
4. **`nlink`**: recorrer todo el árbol de directorios y contar nombres. Inodos sin nombre van a `lost+found`.
5. **Duplicados** y **punteros fuera de rango**.
6. **Directorios**: `.` y `..` correctos, inodos de las entradas asignados.

Dos problemas

Lento: lee todo el disco aunque el fallo tocó tres bloques. Con discos de varios TB son horas.

No ve todo: si llegaron I y B pero no Db, todo cuadra y el archivo tiene basura.

Analogía

Perder las llaves en el dormitorio y revisar toda la casa.

Journaling: anotar antes de escribir

Write-ahead logging

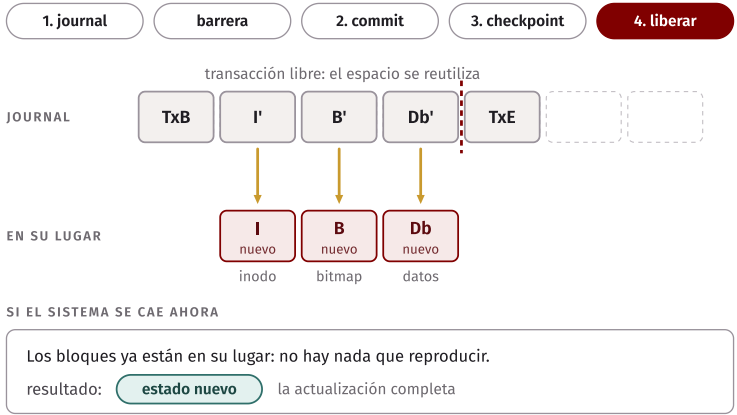
Antes de tocar las estructuras en su lugar, escribir en un área aparte, el **journal** (o log), una descripción de lo que se va a hacer. Si se cae a la mitad, se lee la nota y se termina el trabajo.

Viene de las bases de datos. En sistemas de archivos: ext3, ext4, XFS, JFS, NTFS… y xv6.



El journal suele ser **pequeño** (decenas o cientos de MB) y se reutiliza en círculo.

Journaling de datos, paso a paso



Journal, barrera, commit (TxE: desde ahí es durable), checkpoint y liberar.

¿Por qué TxE va por separado?

- Sería más rápido mandar los cinco bloques juntos. Pero el disco puede reordenar las escrituras.
- Si TxE llega antes que Db', la recuperación ve una transacción "completa" y copia basura a su lugar.
- Por eso: escribir todo menos TxE, **esperar** (barrera), y recién entonces TxE. El disco garantiza que una escritura de **512 B** es atómica: TxE cabe en un sector.
- ext4 lo evita de otro modo: un **checksum** del contenido de la transacción en TxB/TxE. Si no coincide, la transacción se descarta. Una espera menos.

bash

```
$ ./journal reorder
== sin barrera: TxE llega antes ==
en disco al fallar    tras recuperar
(nada)                ok: estado viejo
TxB                   ok: estado viejo
TxB TxE               CORRUPTO: replay de basura
TxB TxE j:I           CORRUPTO: replay de basura
TxB TxE j:I j:B       CORRUPTO: replay de basura
TxB TxE j:I j:B j:Db  ok: estado nuevo
```

`code/week-14/journal.c` simula un fallo después de cada escritura y luego la recuperación.

Un fallo en cada punto, y qué hace la recuperación



Fallo	Recuperación
antes de TxE	se ignora la transacción: estado viejo
después de TxE	replay *(redo)*: se copian I', B', Db' a su lugar
durante el checkpoint	replay de todo otra vez: es idempotente
después de liberar	nada que hacer

Journaling de metadatos (modo ordered)



- El journaling de datos escribe cada bloque de datos dos veces. Para escrituras grandes, eso es la mitad del ancho de banda.
- **Modo ordered** (ext3/ext4 por defecto): al journal solo van los **metadatos**; Db va directo a su lugar, antes del commit.
- ¿Por qué Db primero? Si TxE llegara antes que Db y se cae, el replay deja un inodo válido que apunta a **basura**. Regla: primero lo apuntado, después lo que apunta.
- Existe un modo **writeback** sin ese orden: más rápido, y con el riesgo de basura.

Revocación: cuando un bloque cambia de owner



ext3/ext4: al borrar se escribe un **registro de revocación** *(revoke record)*. En la recuperación, primero se buscan las revocaciones; esos bloques no se reproducen.

Otras soluciones

Técnica	Idea	Dónde
`fsck`	dejar que pase, revisar todo al arrancar	ext2, FAT
Journaling	anotar en el journal, confirmar, luego escribir	ext3/4, XFS, NTFS, xv6
Soft updates	ordenar *todas* las escrituras para que el disco nunca apunte a basura (el dato antes que el inodo)	FFS de FreeBSD
Copy-on-write	nunca sobrescribir: escribir versiones nuevas en lugares libres y al final cambiar un solo puntero raíz	ZFS, btrfs, APFS; LFS (semana 15)
Backpointers	cada bloque dice quién lo apunta; se verifica al leer, sin ordenar escrituras	investigación

Lo común

Todas buscan que una escritura de un sector, atómica en el hardware, haga visible de golpe una actualización de muchos bloques.

SECCIÓN 05

El log de xv6

05

En xv6: cada llamada es una transacción

uso típico

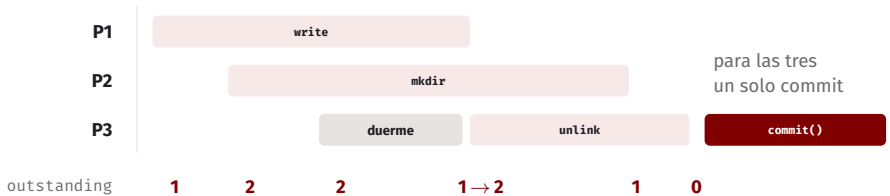
```
begin_op();
bp = bread(...);
// modificar bp->data[]
log_write(bp);
brelse(bp);
end_op();
```

kernel/file.c: filewrite() (extracto)

```
int max = ((MAXOPBLOCKS - 1 - 1 - 2) / 2) * BSIZE;
// ...
  begin_op();
  ilock(f->ip);
  if ((r = writei(f->ip, 1, addr + i, f->off, n1)) > 0)
    f->off += r;
  iunlock(f->ip);
  end_op();
```

- Toda llamada que modifica el disco (`write`, `mkdir`, `unlink`…) va entre `begin_op` y `end_op`.
- `log_write` reemplaza a `bwrite`, pero **no escribe**: anota el número de bloque y lo *fija (pin)* en la caché hasta el commit.
- El log tiene `LOGBLOCKS` = 30 bloques; una llamada escribe a lo más `MAXOPBLOCKS` = 10.
- Por eso `filewrite` parte un `write` grande en trozos de $\lfloor (10-4)/2 \rfloor \cdot 1024 = 3072$ B, cada uno en su transacción.

`begin_op` y `end_op`: group commit



La **última** en salir (`outstanding` = 0) hace el commit de las tres. Mientras tanto, cualquier `begin_op` nuevo duerme (`log.committing`).

`commit()`: cuatro pasos

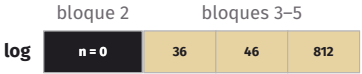
kernel/log.c

```
static void
commit()
{
  if (log.lh.n > 0) {
    write_log();      // Write modified blocks from cache to log
    write_head();     // Write header to disk -- the real commit
    install_trans(0); // Now install writes to home locations
    log.lh.n = 0;
    write_head(); // Erase the transaction from the log
  }
}

static void
recover_from_log(void)
{
  read_head();
  install_trans(1); // if committed, copy from log to disk
  log.lh.n = 0;
  write_head(); // clear the log
}
```

- `write_log`: copiar los bloques modificados de la caché al log (bloques 3–32).
- `write_head`: escribir la cabecera (bloque 2), un `struct logheader` con `n` y la lista de destinos. Este es el commit.
- `install_trans`: el checkpoint.
- `write_head` con `n = 0`: liberar el log.
- `recover_from_log`: al arrancar, desde `fsinit`.

¿Por qué el commit es escribir la cabecera?



en su lugar



Fallo antes de `write_head`: la cabecera dice $n = 0$ y se ignora el log. Fallo después: $n = 3$ y se copian los tres bloques, aunque algunos ya estuvieran (idempotente). El segundo `write_head` vacía el log antes de la próxima transacción. Una sola escritura de un bloque separa “nada” de “todo”: por eso es el commit.

xv6 frente a ext3

	xv6	ext3/ext4
Qué va al log	datos *y* metadatos: `writei` pasa los bloques de datos por `log_write`	ordered: solo metadatos
Marca de commit	la cabecera con $n > 0$ (124 B, cabe en un sector)	TxE en un sector, o checksum
El log	se vacía en cada commit	circular, con muchas transacciones
Revocación	no hace falta: el log se vacía antes de que un bloque cambie de owner	registros de revoke

Para P5

Un bloque doblemente indirecto también se modifica con `log_write`, y cada trozo de `filewrite` debe seguir cabiendo en `MAXOPBLOCKS`.

Resumen

- Un sistema de archivos reparte el disco en **superbloque**, **bitmaps**, **tabla de inodos** y **datos**. El número de inodo da su ubicación por aritmética.
- El inodo apunta a sus datos con punteros **directos**, **indirectos** y multinivel (o con **extents**). Con 4 KB, punteros de 4 B, 12 directos y un indirecto: 4144 KB.
- Abrir `/foo/bar` sin caché cuesta 5 lecturas; cada escritura que reserva un bloque, 5 E/S. La **caché** y el **write buffering** lo hacen viable.
- **FFS** agrupa lo relacionado en grupos de cilindros y reparte los archivos grandes en trozos que amortizan el posicionamiento.
- Un fallo a mitad de una actualización deja el disco inconsistente. `fsck` revisa todo; el **journaling** escribe primero en el journal, confirma con un bloque atómico, hace checkpoint y libera. En modo ordered, los datos van antes del commit.
- xv6 tiene un log de rehacer con datos y metadatos; su commit es escribir la cabecera, y `end_op` confirma en grupo.

Para la próxima clase

- **Laboratorio**: `vsfs.py` (preguntas 1–4 del cap. 40), `ffs.py` y `fsck.py`. Después: dibujar el formato del disco de xv6 y ubicar `bmap()` en `kernel/fs.c`.
- Prueben `code/week-14`: `make test` en el contenedor.
- Semana 15: OSTEP cap. 43 (LFS), 44 (SSD) y 45 (integridad de datos).

Entregas

P5 (archivos grandes y enlaces simbólicos) se entrega en la semana 16. El bono `cow` también.

Preguntas para pensar

¿Cuántas lecturas necesita `open("/a/b/c/d")` en VSFS sin caché? ¿Cuál es el tamaño máximo de un archivo con bloques de 1 KB, 12 directos y un indirecto? ¿Por qué el journaling de metadatos escribe los datos antes que la transacción? ¿Qué pasaría en xv6 si `commit` no hiciera el segundo `write_head`?