

UNIVERSIDAD NACIONAL DE INGENIERÍA

Sistemas Operativos

RAID y la interfaz de archivos

Muchos discos como uno, y cómo ven los archivos los programas

Gerson Garrido

Semana 13

Hoy

Sesión A · RAID

1. La interfaz y el modelo de fallos
2. RAID 0, 1, 4 y 5; la paridad XOR
3. El problema de la escritura pequeña
4. Comparación con números; RAID 6 y RAID por software

Sesión B · Archivos y directorios

5. `open`, `read`, `write`, `lseek`
6. La tabla de archivos abiertos: `fork` y `dup`
7. `fsync`, `rename`, `stat`
8. Directorios, enlaces, permisos, `mkfs` y `mount`

> **Lectura**
>
> OSTEP cap. 38 *Redundant Arrays of Inexpensive Disks (RAIDs)* y cap. 39 *Interlude: Files and Directories*. Código: `code/week-13`.

SECCIÓN 01

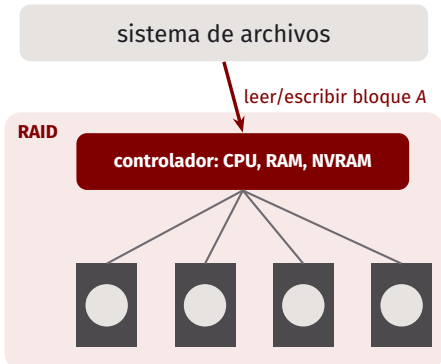
RAID

01



Un disco no alcanza

- Es **lento**: el disco de la semana 12 hace unas 76 lecturas aleatorias por segundo.
- Es **pequeño**: una base de datos no cabe en uno.
- **Falla**: y con él se van todos los datos.
- Idea de Patterson, Gibson y Katz (Berkeley, 1988): juntar muchos discos baratos *(Redundant Array of Inexpensive Disks)* y presentarlos como uno solo, más rápido, más grande y más confiable.
- Es **transparente**: el sistema operativo y los programas no cambian. Por eso se adoptó tan rápido.



por dentro: varios discos, cálculo y memoria

Modelo de fallos y cómo vamos a comparar

Modelo fail-stop

Un disco está **funcionando** (se lee y escribe bien) o **falló** por completo. Y cuando falla, el RAID se entera de inmediato.

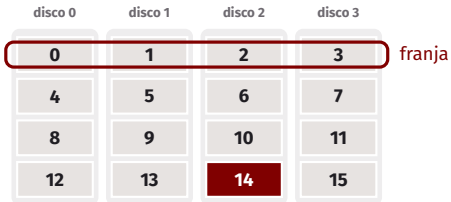
- Deja fuera la corrupción silenciosa y los sectores dañados sueltos: eso es más difícil (cap. 45).
- Tres ejes para comparar: **capacidad** útil, **confiabilidad** (cuántas fallas aguanta) y **rendimiento**.

Dos cargas de trabajo por disco, con el disco de escritorio de la semana 12 (7 200 RPM, 9 ms de búsqueda, 105 MB/s):

	Pedido	**Tasa**
S secuencial	100 MB en 965,5 ms	**103**,**6** MB/s
R aleatorio	4 KB en 13,20 ms	**0**,**303** MB/s

$S/R \approx 340$. En aleatorio, cada disco da unas 76 E/S por segundo. N discos, de B bloques cada uno; T = latencia de un pedido a un disco.

RAID 0: striping



- Repartir los bloques entre los discos en ronda *(round-robin)*. Una fila es una franja *(stripe)*.
- El RAID traduce el bloque lógico A:

$$\text{disco} = A \bmod N \qquad \text{desplazamiento} = A \operatorname{div} N$$

- Bloque 14 con $N = 4$: disco $14 \bmod 4 = \mathbf{2}$, fila $14 \operatorname{div} 4 = \mathbf{3}$.

RAID 0 en números

Capacidad $N \cdot B$ (perfecta). Confiabilidad 0: si falla un disco, se pierde el arreglo. Rendimiento $N \cdot S$ y $N \cdot R$: la cota superior.

El tamaño del chunk



chunk = 2 bloques

- El **chunk** es cuánto se pone en un disco antes de pasar al siguiente. Arriba, $C = 2$ bloques de 4 KB: chunk de 8 KB, franja de 32 KB.
- Bloque 13: disco $(13 \operatorname{div} C) \bmod N = 6 \bmod 4 = \mathbf{2}$.
- **Chunk chico**: un archivo se reparte en muchos discos (más paralelismo para él), pero el pedido espera al disco más lento en posicionarse.
- **Chunk grande**: menos paralelismo por archivo; hace falta tener muchos pedidos a la vez. Lo usual: 64 KB a 512 KB.

RAID 1: espejo



escribir 5: las dos copias

- Cada bloque se guarda **dos veces**, en discos distintos. Aquí los pares de espejos se reparten con striping: RAID 10.
- Leer: de cualquiera de las dos copias.
- Escribir: en las dos, en paralelo; la latencia es la del más lento.

RAID 1 en números

Capacidad $N \cdot B/2$. Aguanta 1 falla seguro, hasta $N/2$ con suerte. Aleatorio: lectura $N \cdot R$, escritura $\frac{N}{2} \cdot R$. Secuencial: $\frac{N}{2} \cdot S$ en ambos.

¿Por qué la lectura secuencial no da $N \cdot S$? Cada disco recibe un bloque sí y uno no: mientras pasa el bloque que se salta, no entrega datos útiles.

RAID 4: un disco de paridad



$P1 = 4 \oplus 5 \oplus 6 \oplus 7$

- El espejo gasta la mitad del espacio. ¿Se puede proteger con menos?
- Por cada franja, un bloque de **paridad** en un disco aparte.
- La paridad es el **XOR** ($\oplus$) bit a bit de los bloques de datos de la franja.
- XOR de varios bits da 1 si hay un número *impar* de unos. Invariante: en cada posición, contando la paridad, el número de unos es par.

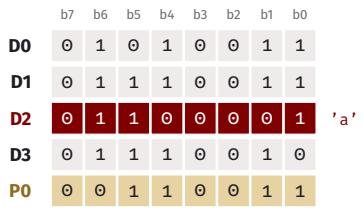
La paridad con bits reales

	b7	b6	b5	b4	b3	b2	b1	b0	
D0	0	1	0	1	0	0	1	1	`'S'`
D1	0	1	1	1	0	0	1	1	`'s'`
D2	0	1	1	0	0	0	0	1	`'a'`
D3	0	1	1	1	0	0	1	0	`'r'`
P0	0	0	1	1	0	0	1	1	`0x33`

Primer byte de cada bloque de la franja 0 (salida real de `raid5`).

- Bit 6: cuatro unos, número par ⇒ paridad 0.
- Bit 5: tres unos, impar ⇒ paridad 1.
- Así con los ocho bits, y con los 4 096 bytes del bloque.
- El controlador hace esto con una instrucción XOR por palabra: es barato.

Falla un disco: reconstruir con XOR



- Falla el disco 2. Fail-stop: el RAID sabe *cuál* falló.
- Bit 5: los demás tienen 0, 1, 1 y 1 (tres unos). Para que el total sea par, el que falta es 1.
- Es lo mismo que calcular la paridad:

$$D2 = D0 \oplus D1 \oplus D3 \oplus P0$$

- Reconstruir un disco entero obliga a **leer todos los demás**. Con paridad se aguanta una falla, no dos.

El problema de la escritura pequeña



4 E/S en total. Los discos 0, 2 y 3 no se tocan.

RAID 4: el disco de paridad es el cuello de botella



- Dos escrituras pequeñas a la vez: bloques 4 y 13. Sus datos están en discos distintos: podrían ir en paralelo.
- Pero las dos leen y escriben la paridad en el disco 4: se **serializan**.
- El disco de paridad hace 2 E/S por cada escritura lógica. Todo el arreglo queda en **R/2**, y no mejora al agregar discos.
- Lo demás va bien: capacidad $(N-1)\cdot B$, secuencial $(N-1)\cdot S$ (con escrituras de franja completa), lectura aleatoria $(N-1)\cdot R$.

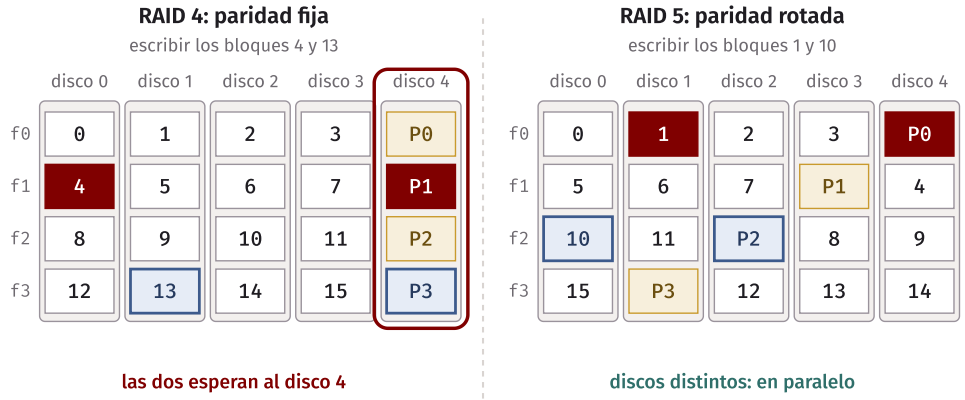
RAID 5: rotar la paridad



La distribución de `code/week-13/raid5`.

- La paridad de cada franja va en un disco **distinto**: la franja s la pone en el disco $N - 1 - (s \bmod N)$.
- Escribir el bloque 1 usa los discos 1 y 4.
- Escribir el bloque 10 usa los discos 0 y 2: en paralelo con la anterior.
- Con muchos pedidos, todos los discos trabajan. Cada escritura sigue costando 4 E/S: escritura aleatoria $\frac{N}{4} \cdot \mathbf{R}$. Lectura aleatoria $N \cdot R$.
- Por eso RAID 5 reemplazó casi por completo a RAID 4.

RAID 4 frente a RAID 5



Cada color es una escritura pequeña: su bloque de datos y su bloque de paridad.

Comparación

	RAID 0	RAID 1	RAID 4	RAID 5
Capacidad	$N \cdot B$	$N \cdot B/2$	$(N-1) \cdot B$	$(N-1) \cdot B$
Fallas que aguanta	0	1 (hasta $N/2$ con suerte)	1	1
Lectura secuencial	$N \cdot S$	$\frac{N}{2} \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Escritura secuencial	$N \cdot S$	$\frac{N}{2} \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Lectura aleatoria	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Escritura aleatoria	$N \cdot R$	$\frac{N}{2} \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} \cdot R$
Latencia de lectura	T	T	T	T
Latencia de escritura	T	T	$2T$	$2T$

Modelo simplificado de OSTEP (fig. 38.8): ignora, por ejemplo, que esperar a dos discos cuesta un poco más que esperar a uno.

Ejemplo resuelto: 8 discos de 4 TB

$N = 8$, $B = 4$ TB, $S = 103{,}6$ MB/s, $R = 0{,}303$ MB/s (76 E/S por segundo por disco).

	Capacidad	Secuencial	Lect. aleatoria	Escr. aleatoria
RAID 0	32 TB	829 MB/s	606 E/S/s	606 E/S/s
RAID 1	16 TB	414 MB/s	606 E/S/s	303 E/S/s
RAID 4	28 TB	725 MB/s	530 E/S/s	38 E/S/s
RAID 5	28 TB	725 MB/s	606 E/S/s	151 E/S/s

Cómo elegir

Solo rendimiento y nada de confiabilidad: RAID 0. Escrituras aleatorias y rendimiento con confiabilidad: RAID 1 (paga con la mitad del espacio). Capacidad con confiabilidad: RAID 5, sabiendo que las escrituras pequeñas son caras.

RAID 5 en código: `raid5.c`

code/week-13/raid5.c (extracto)

```c
// XOR de los N-1 sobrevivientes
for (int d = 0; d < N; d++) {
    if (d == fail) continue;
    read_blk(d, s, tmp);
    xor_into(rec, tmp);
}
...
int pd = parity_disk(s);
read_blk(d, s, old);      // dato viejo
read_blk(pd, s, pold);    // paridad vieja
// P_nueva = P_vieja ^ D_viejo ^ D_nuevo
xor_into(pold, old);
xor_into(pold, newb);
write_blk(d, s, newb);    // dato nuevo
write_blk(pd, s, pold);   // paridad nueva
```

code/week-13

```
$ ./raid5 2
...
Falla el disco 2. Reconstrucción:
  f0: "a, la pa"  == original
  f1: "s: basta"  == original
  f2: P2 = 59 4a 08 0a 01 00 5b 4e  == original
  f3: " la fran"  == original
  4/4 bloques recuperados, 16 lecturas

Escritura pequeña al bloque 6:
  dato en disco 1, paridad en disco 3
  2 lecturas + 2 escrituras = 4 E/S
  paridad sustractiva == recalculada
```

Cinco discos en memoria, bloques de 8 bytes. En la franja 2, el disco 2 guardaba la paridad: se reconstruye con el mismo XOR.

RAID hoy: discos grandes y RAID 6

- Reconstruir un disco exige leer todos los demás y escribir el nuevo completo.
- Cota inferior: tamaño / velocidad. Un disco de 20 TB a 250 MB/s tarda $\approx$ **22** horas, y más si el arreglo sigue atendiendo pedidos.
- Durante ese tiempo un RAID 5 no tiene redundancia. Los discos son del mismo lote y la misma edad, y la reconstrucción los exige al máximo.



Controladora y 4 bandejas de discos. Redline, CC BY-SA 3.0, Wikimedia Commons.

RAID 6

Dos bloques de redundancia por franja: P (XOR) y Q (código de Reed–Solomon), rotados. Aguanta 2 fallas. Capacidad $(N-2)\cdot B$. Una escritura pequeña cuesta 6 E/S: leer y escribir dato, P y Q.

RAID no es un backup

Si alguien borra un archivo, el RAID borra todas las copias.

RAID por hardware o por software

- **Hardware**: una tarjeta controladora con su CPU y RAM respaldada por batería (NVRAM).
- **Software**: el kernel hace el trabajo. En Linux, el driver `md`, que se administra con `mdadm`:

```
mdadm --create /dev/md0 --level=5 \
  --raid-devices=4 /dev/sd[b-e]
```

- **ZFS** junta el sistema de archivos y el RAID: RAID-Z1, Z2 y Z3 (1, 2 o 3 discos de paridad).

El hueco de escritura *(write hole)*

Si se va la luz entre escribir el dato y la paridad, la franja queda **inconsistente** sin que nadie lo note. Igual con dos copias de un espejo.

- Hardware: anotar la escritura en la NVRAM antes de hacerla (un log).
- `md`: un bitmap de intención de escritura (resincroniza solo las zonas sucias) y, para RAID 5, un journal o un log de paridad parcial (PPL).
- ZFS: nunca sobrescribe en el lugar (copy-on-write) y guarda checksums que detectan corrupción silenciosa, que el modelo fail-stop no contempla.

SECCIÓN 02

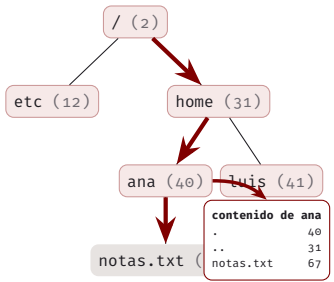
Archivos y descriptores

02



Dos abstracciones: archivo y directorio

- Un **archivo** es un arreglo de bytes. Al sistema de archivos no le importa qué hay adentro.
- Su nombre de bajo nivel es el **número de inodo** *(inode number)*. El inodo guarda sus metadatos y dónde están sus bloques.
- Un **directorio** también tiene inodo, y su contenido es una lista de pares (nombre, número de inodo).
- Directorios dentro de directorios forman el **árbol**; una ruta como `/home/ana/notas.txt` se recorre desde la raíz.



Abrir, leer y escribir

crear un archivo

```
int fd = open("foo",
              O_CREAT | O_WRONLY | O_TRUNC,
              S_IRUSR | S_IWUSR);
```

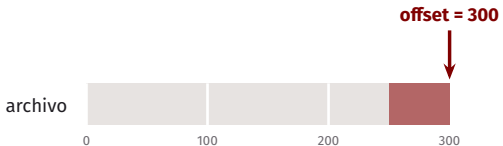
- `O_CREAT`: crearlo si no existe; `O_TRUNC`: vaciarlo si existe.
- `open` devuelve un **descriptor de archivo**: un entero chico, privado de cada proceso. Es una capability: con él se puede leer o escribir.
- Los descriptores 0, 1 y 2 ya vienen abiertos (entrada, salida y error): el primer `open` devuelve 3, el número libre más bajo.
- `read` devuelve cuántos bytes leyó; 0 significa fin de archivo.

strace cat foo (resumido)

```
$ echo hola > foo
$ strace -e trace=openat,read,write,close cat foo
...
openat(AT_FDCWD, "foo", O_RDONLY)       = 3
read(3, "hola\n", 131072)               = 5
write(1, "hola\n", 5hola
)                       = 5
read(3, "", 131072)                     = 0
close(3)                                = 0
close(1)                                = 0
close(2)                                = 0
+++ exited with 0 +++
```

`strace` muestra cada llamada al sistema. Se omiten las primeras líneas (cargar bibliotecas).

El offset y `lseek`



Llamada	Retorna	offset
`fd = open("f", O_RDONLY)`	3	0
`read(fd, buf, 100)`	100	100
`read(fd, buf, 100)`	100	200
`lseek(fd, 250, SEEK_SET)`	250	250
`read(fd, buf, 100)`	50	300
`read(fd, buf, 100)`	0	300

- Cada archivo abierto tiene un **offset**: dónde empieza la próxima lectura o escritura. `read` y `write` lo avanzan solos.
- `lseek(fd, off, whence)` lo cambia: `SEEK_SET` (desde el inicio), `SEEK_CUR` (desde donde está) o `SEEK_END` (desde el final).

`lseek` no mueve el disco

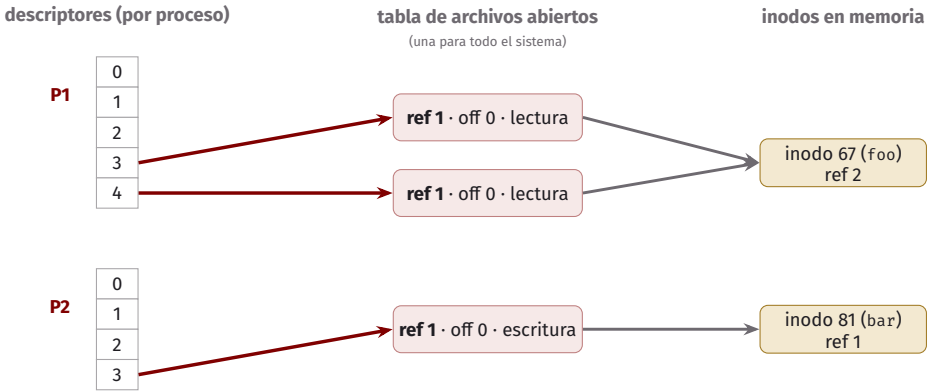
Solo cambia un número en la memoria del kernel. El brazo se mueve después, si una lectura necesita un bloque que no está en caché.

SECCIÓN 03

La tabla de archivos abiertos

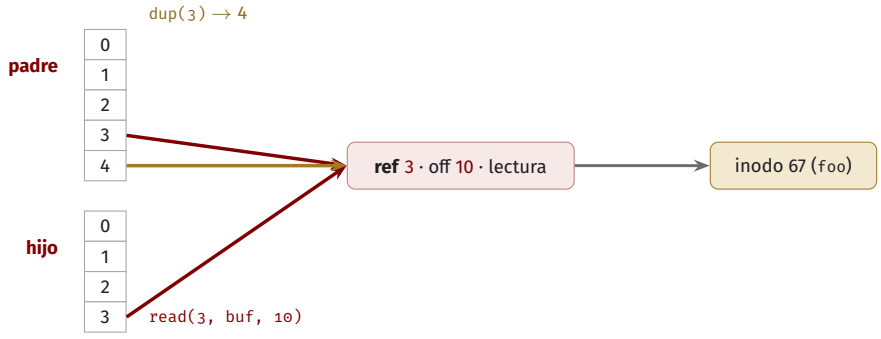
03

Tres niveles: descriptor, archivo abierto, inodo



P2 abre `bar`: otra entrada. Cada `open` crea una entrada nueva; dos descriptores la comparten solo si uno se copió del otro.

`fork` y `dup`: entradas compartidas



`dup(3)` da otro descriptor del mismo proceso que apunta a la misma entrada. La shell hace `cmd > out 2>&1` con `dup2(1, 2)`: la salida y el error comparten offset y no se pisan.

Comprobarlo: `shareoff.c`

code/week-13/shareoff.c (extracto)

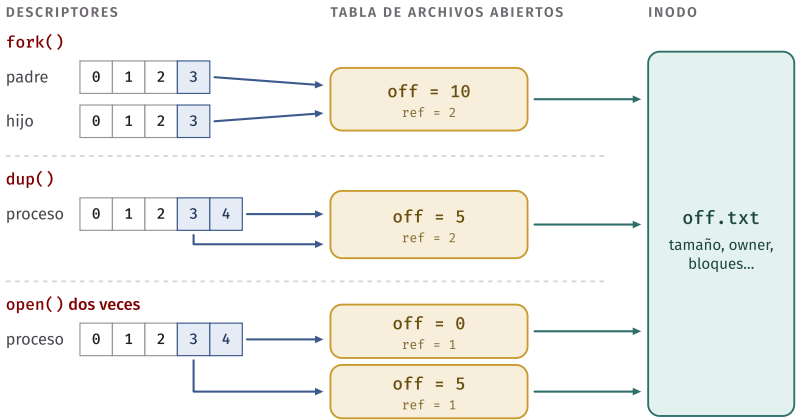
```c
fd = open(argv[1], O_RDONLY);
fflush(stdout);             // no duplicar el buffer
pid_t pid = fork();
if (pid == 0) {             // hijo: lee 10 bytes
    read_n(fd, 10, "hijo ");
    exit(0);
}
waitpid(pid, NULL, 0);      // padre: espera al hijo
printf("  offset del padre: %ld\n", where(fd));
read_n(fd, 5, "padre");     // sigue donde quedó
...
int a = open(argv[1], O_RDONLY);
int b = open(argv[1], O_RDONLY);
read_n(a, 5, "a    ");
```

contenedor del curso

```
$ ./shareoff /tmp/w13/off.txt
1) fork():
  hijo  lee 10 bytes: "abcdefghij"
  offset del padre: 10
  padre lee 5 bytes: "klmno"
2) dup():
  fd    lee 5 bytes: "abcde"
  offset de fd=3: 5, de fd2=4: 5
3) open() dos veces:
  a     lee 5 bytes: "abcde"
  offset de a=3: 5, de b=4: 0
  b     lee 5 bytes: "abcde"
```

Padre e hijo: sí comparten. Dos `open`: no.
(`where(fd)` es `lseek(fd, 0, SEEK_CUR)`.)

Lo que mostró `shareoff`



El offset vive en la entrada de la tabla de archivos abiertos: ni en el descriptor ni en el inodo.

xv6: `struct file` y la tabla `ftable`

kernel/file.h

```
struct file {
  enum { FD_NONE, FD_PIPE, FD_INODE, FD_DEVICE } type;
  int ref; // reference count
  char readable;
  char writable;
  struct pipe *pipe; // FD_PIPE
  struct inode *ip;  // FD_INODE and FD_DEVICE
  uint off;          // FD_INODE
  short major;       // FD_DEVICE
};
```

kernel/file.c

```
struct {
  struct spinlock lock;
  struct file file[NFILE];
} ftable;
```

- En `struct proc` (`kernel/proc.h`): `struct file *ofile[NOFILE];` es la tabla de descriptores: el descriptor es el índice. `NOFILE` = 16.
- `ftable` es la tabla de archivos abiertos de todo el sistema: `NFILE` = 100 entradas; libre si `ref == 0`.
- `sys_open` siempre pide una entrada nueva con `filealloc()` y le pone `off = 0`.

xv6: `filedup` y `fileclose`

kernel/proc.c, en kfork()

```
for (i = 0; i < NOFILE; i++)
  if (p->ofile[i])
    np->ofile[i] = filedup(p->ofile[i]);
```

kernel/file.c: filedup()

```
acquire(&ftable.lock);
if (f->ref < 1)
  panic("filedup");
f->ref++;
release(&ftable.lock);
return f;
```

kernel/file.c: fileclose()

```
acquire(&ftable.lock);
if (f->ref < 1)
  panic("fileclose");
if (--f->ref > 0) {
  release(&ftable.lock);
  return;
}
ff = *f;
f->ref = 0;
f->type = FD_NONE;
release(&ftable.lock);
...  iput(ff.ip);  ...
```

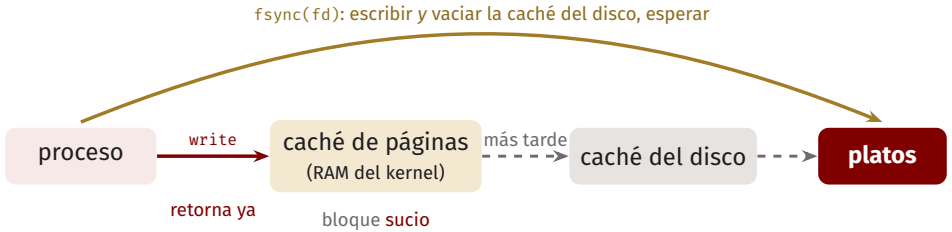
`kfork` y `sys_dup` copian el puntero: la entrada (y su `off`) es la misma. `fileclose` solo la libera cuando `ref` llega a 0. xv6 no tiene `lseek`: el offset solo avanza.

SECCIÓN 04

Durabilidad y metadatos

04

`write` no es durable: `fsync`



- `write` copia los datos a la memoria del kernel y retorna. Así se pueden juntar y reordenar escrituras (semana 12).
- En Linux, por defecto, cada 5 s se escriben los bloques sucios que llevan más de 30 s.
- `fsync(fd)` fuerza al disco los datos *y* metadatos de ese archivo, y espera. `fdatasync` omite metadatos que no hacen falta para leerlo.
- Un archivo **nuevo** vive también en su directorio: hay que hacer `fsync` del directorio, o el nombre puede no estar después de un fallo.

`rename` es atómico: el patrón temporal + `rename`

atomic.c, sin manejo de errores

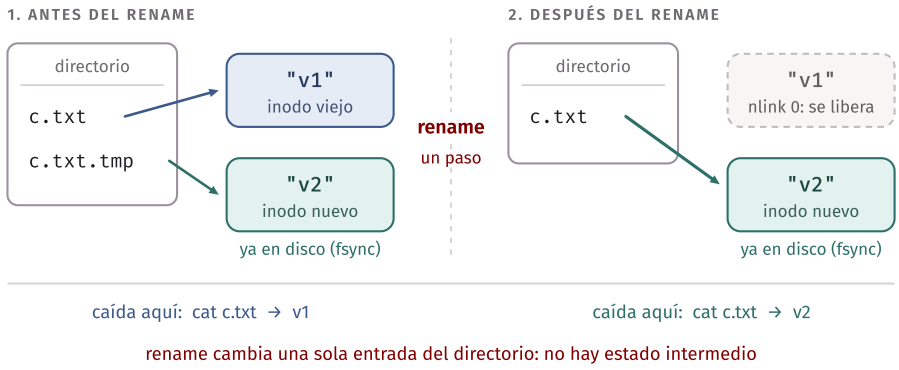
```c
// 1. la versión nueva completa, en un temporal
int fd = open(tmp,
              O_WRONLY | O_CREAT | O_TRUNC, 0644);
write(fd, argv[2], n);
// 2. al disco antes de cambiar el nombre
fsync(fd);
close(fd);
// 3. atómico: el nombre apunta al inodo
//    viejo o al nuevo, nunca a una mezcla
rename(tmp, argv[1]);
// 4. el cambio vive en el directorio
int dfd = open(dirname(dir),
               O_RDONLY | O_DIRECTORY);
fsync(dfd);
```

- Reescribir un archivo en su lugar es peligroso: un fallo a la mitad deja una mezcla o un archivo vacío.
- `rename(viejo, nuevo)` cambia la entrada del directorio en un solo paso, aun ante una caída. Así guardan los editores.

en /tmp/w13

```
$ /work/atomic c.txt v1
escrito y sincronizado: c.txt.tmp
rename c.txt.tmp -> c.txt listo
$ /work/atomic c.txt v2 crash
escrito y sincronizado: c.txt.tmp
*** caída simulada antes de rename ***
$ cat c.txt
v1
```

`rename`: antes y después



rename cambia una sola entrada del directorio: no hay estado intermedio

Por eso el temporal se sincroniza antes del `rename`, y el directorio después.

`stat`: los metadatos del inodo

struct stat (campos principales)

```
struct stat {
  dev_t     st_dev;     // dispositivo
  ino_t     st_ino;     // número de inodo
  mode_t    st_mode;    // tipo y permisos
  nlink_t   st_nlink;   // enlaces duros
  uid_t     st_uid;     // owner
  gid_t     st_gid;     // grupo
  off_t     st_size;    // tamaño en bytes
  blkcnt_t  st_blocks;  // bloques de 512 B
  struct timespec st_atim, st_mtim,
                  st_ctim;   // tiempos
};
```

- `stat` sigue los enlaces simbólicos; `lstat` no; `fstat` recibe un descriptor.

code/week-13/mystat

```
$ ./mystat /tmp/w13/docs/a.txt
/tmp/w13/docs/a.txt
  tipo      archivo regular
  inodo     35442243  (dispositivo 0:108)
  enlaces   2
  tamaño    5 bytes, 8 bloques de 512
  permisos  0644 rw-r--r--  owner root  grupo root
  acceso    2026-11-04 14:35:41
  modif.    2026-11-04 14:35:41
  cambio    2026-11-04 14:35:41
```

- El nombre no está en el inodo: está en el directorio.
- 5 bytes ocupan 8 bloques de 512 (4 KB). *modif.*: el contenido; *cambio*: el inodo.

SECCIÓN 05

Directorios y enlaces

05



Directorios: leer sin escribir

leer un directorio

```c
DIR *dp = opendir(".");
struct dirent *d;
while ((d = readdir(dp)) != NULL)
    puts(d->d_name);
closedir(dp);
```

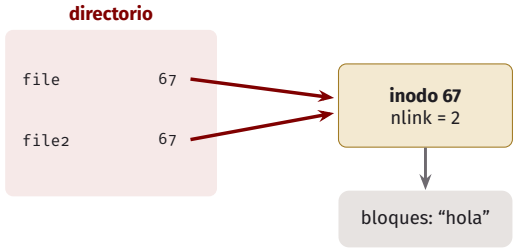
- `mkdir` crea uno con `.` (sí mismo) y `..` (el padre); por eso tiene $2 +$ subdirectorios enlaces. `rmdir` solo borra uno vacío.
- Nadie escribe un directorio con `write`: solo el kernel, vía `open`, `link`, `unlink`, `rename`…
- Cada `struct dirent` trae nombre y número de inodo (`d_ino`); lo demás sale de un `lstat`. Así funciona `ls -l`.

code/week-13/myls

```
$ ./myls -l /tmp/w13/docs
35442243 -rw-r--r--  2 root  root       5 Nov  4 14:35 a.txt
35442243 -rw-r--r--  2 root  root       5 Nov  4 14:35 b.txt
35442240 drwxr-xr-x  2 root  root    4096 Nov  4 14:35 notas
35442245 lrwxrwxrwx  1 root  root       8 Nov  4 14:35 roto -> nada.txt
35442244 lrwxrwxrwx  1 root  root       5 Nov  4 14:35 sim -> a.txt
```

Primera columna: el número de inodo. `a.txt` y `b.txt` tienen el mismo. ¿Cómo?

Enlaces duros y el contador de referencias



- `echo hola > file`: un inodo y un nombre.
- `ln file file2`: otra entrada con el mismo número de inodo. No se copia nada; `nlink` sube a 2.
- Los dos nombres valen lo mismo: ninguno es el “original”.
- `rm file` borra la entrada y baja `nlink` a 1. El archivo sigue ahí con el otro nombre.
- `rm file2`: `nlink` llega a 0 y, si nadie lo tiene abierto, el inodo y sus bloques se liberan.

¿Por qué se llama `unlink`?

- Porque eso hace: quita un **enlace** (un nombre) al inodo. No existe una llamada “borrar archivo”.
- El archivo desaparece cuando se cumplen dos cosas: `nlink == 0` y ningún proceso lo tiene abierto.
- Uso real: crear un temporal, abrirlo y hacerle `unlink` de inmediato. Nadie más lo ve, y el kernel lo borra cuando el proceso lo cierra.

contenedor del curso

```
$ echo hola > file
$ ln file file2
$ ls -li file file2
34297533 -rw-r--r-- 2 root root 5 Oct 29 09:22 file
34297533 -rw-r--r-- 2 root root 5 Oct 29 09:22 file2
$ rm file
$ ls -li file2
34297533 -rw-r--r-- 1 root root 5 Oct 29 09:22 file2
```

abierto pero sin nombre

```
$ exec 3< file2      # la shell lo abre en el fd 3
$ rm file2           # nlink = 0
$ ls
$ cat <&3            # todavía se puede leer
hola
```

xv6: `nlink`, `sys_unlink` e `iput`

kernel/sysfile.c, en sys_unlink()

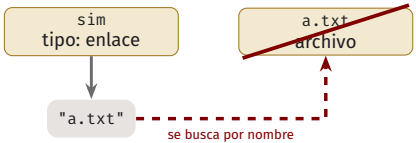
```
memset(&de, 0, sizeof(de));
if (writei(dp, 0, (uint64)&de, off, sizeof(de)) != sizeof(de))
  panic("unlink: writei");
...
ip->nlink--;
iupdate(ip);
```

kernel/fs.c, en iput()

```
int last = (ip->ref == 1 && ip->valid && ip->nlink == 0);
...
if (last) {
  ...
  itrunc(ip);
...
if (last)
  ifree(dev, inum);
```

- `nlink` está en `struct dinode` (`kernel/fs.h`): en el disco.
- `sys_unlink` borra la entrada y resta 1; `sys_link` suma 1.
- `nlink` cuenta nombres; `ref`, referencias abiertas. Con los dos en cero, `iput` libera todo.

Enlaces simbólicos



- `ln -s a.txt sim` crea un archivo nuevo, de otro tipo, cuyo contenido es la **ruta** destino. Tamaño 5 = `strlen("a.txt")`.
- No suma al `nlink` del destino. Puede apuntar a un directorio o a otro sistema de archivos; un enlace duro no.
- Si se borra el destino, queda **colgando** *(dangling)*.

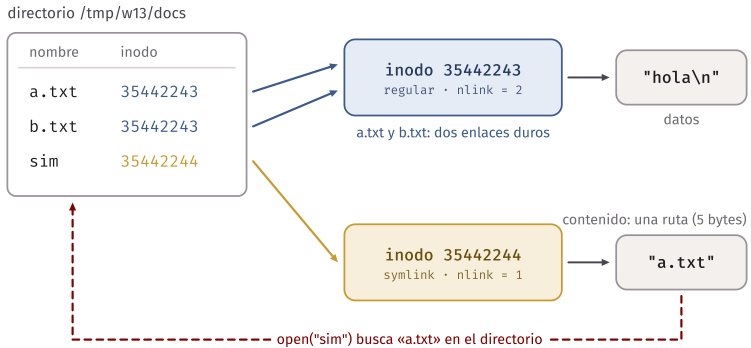
contenedor del curso

```
$ echo hola > a.txt
$ ln -s a.txt sim
$ ls -l sim
lrwxrwxrwx 1 root root 5 Oct 29 09:22 sim -> a.txt
$ cat sim
hola
```

enlace colgando

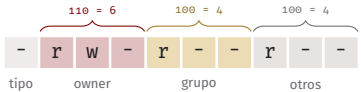
```
$ rm a.txt
$ cat sim
cat: sim: No such file or directory
$ ls -l sim
lrwxrwxrwx 1 root root 5 Oct 29 09:22 sim -> a.txt
```

Enlaces duros y simbólicos, juntos



Duro: otro nombre para el mismo inodo. Simbólico: otro inodo que guarda una ruta.

Permisos y ACL



- Nueve bits en `st_mode`: leer, escribir y ejecutar para owner, grupo y otros. `-rw-r--r--` es `0644`; `chmod 600 f` deja solo al owner.
- En un directorio: `r` = listar, `w` = crear y borrar entradas, `x` = atravesarlo para llegar a lo de adentro.
- `root` pasa por encima de todo esto.
- Una **ACL** da permisos a usuarios y grupos con nombre, no solo a tres clases. El `+` de `ls -l` avisa que hay una.

contenedor del curso

```
$ touch f
$ setfacl -m u:nobody:r f
$ getfacl f
# file: f
# owner: root
# group: root
user::rw-
user:nobody:r--
group::r--
mask::r--
other::r--

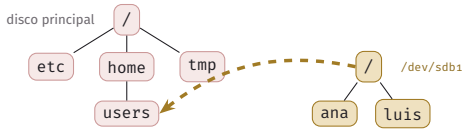
$ ls -l f
-rw-r--r--+ 1 root root 0 Oct 29 09:22 f
```

`mkfs` y `mount`: armar el árbol

contenedor del curso

```
$ truncate -s 64M disk.img
$ mkfs.ext4 -E nodiscard disk.img
mke2fs 1.47.0 (5-Feb-2023)
Creating filesystem with 16384 4k blocks and 16384 inodes
Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
```

- `mkfs` escribe un sistema de archivos vacío (solo la raíz) en un dispositivo o, aquí, en un archivo imagen.
- `mount -t ext4 /dev/sdb1 /home/users` lo pega en un punto del árbol. Pide privilegios: en el contenedor no se puede.



Un solo árbol, muchos sistemas de archivos (ext4, tmpfs, proc…). `mount` sin argumentos los lista.

Herramientas propias: `mytail` y `myfind`

code/week-13/mytail.c (extracto)

```c
off_t pos = end > 0 ? end - 1 : 0;
while (pos > 0 && !found) {
    off_t from = pos >= CHUNK ? pos - CHUNK : 0;
    lseek(fd, from, SEEK_SET);   // saltar atrás
    ssize_t n = read(fd, buf, pos - from);
    if (n <= 0) break;
    scanned += n;
    // recorrer el bloque al revés
    for (ssize_t i = n - 1; i >= 0; i--) {
        if (buf[i] == '\n' && ++seen == want) {
            start = from + i + 1;
            found = 1;
            break;
        }
    }
    pos = from;
}
```

contenedor del curso

```
$ ./mytail -3 /tmp/w13/log.txt
línea 1998
línea 1999
línea 2000
[mytail: 4096 de 22893 bytes examinados]
$ ./myfind /tmp/w13 -name '*.c'
/tmp/w13/src/main.c
/tmp/w13/src/util.c
$ ./myfind /tmp/w13 -type l
/tmp/w13/docs/roto
/tmp/w13/docs/sim
```

- `mytail` lee solo el último bloque de 4 KB, no los 22 KB: para eso sirve `lseek`.
- `myfind` baja con `scandir` y `lstat`: no sigue enlaces simbólicos, así que no cae en ciclos.

En xv6, y lo que viene en P5

Lo que xv6 tiene

- `open`, `read`, `write`, `close`, `dup`, `fstat`, `link`, `unlink`, `mkdir`, `chdir`.
- **No** tiene `lseek`, `rename`, `fsync` ni enlaces simbólicos. Tiene un `sync` que espera a que se confirme el log.
- Sus enlaces duros funcionan como vimos: prueben `ln` y `rm` en su shell.

P5: lab `fs` del MIT (se publica hoy)

1. **Archivos grandes**: un bloque doblemente indirecto en el inodo (semana 14).
2. **Enlaces simbólicos**: la llamada `symlink(target, path)`, el tipo `T_SYMLINK` y la opción `O_NOFOLLOW`. `open` sigue el enlace, aunque apunte a otro enlace, con un límite de profundidad contra los ciclos.

Entrega: semana 16.

Resumen

- **RAID** presenta muchos discos como uno. Modelo fail-stop; se compara en capacidad, confiabilidad y rendimiento (N, B, S, R, T).
- RAID 0 reparte (rápido, sin protección); RAID 1 copia (la mitad del espacio); RAID 4/5 guardan la **paridad XOR**, que reconstruye cualquier disco que falle.
- Una **escritura pequeña** con paridad cuesta 4 E/S. En RAID 4 el disco de paridad la limita a $R/2$; RAID 5 rota la paridad y llega a $\frac{N}{4} \cdot R$.
- Un **descriptor** indexa la tabla del proceso, que apunta a una entrada de la **tabla de archivos abiertos** (offset, ref), que apunta al **inodo**. `fork` y `dup` comparten la entrada; dos `open`, no.
- `write` no es durable: `fsync`. Reemplazo atómico: temporal, `fsync`, `rename`.
- Un directorio guarda pares (nombre, inodo). `link` agrega un nombre; `unlink` lo quita; el inodo se libera con `nlink` = 0 y sin referencias abiertas. Los enlaces simbólicos guardan una ruta y pueden quedar colgando.

Para la próxima clase

- **Laboratorio**: `raid.py`, preguntas 1–5 del cap. 38. Tarea de código del cap. 39: `mystat`, `myls -l`, `mytail -n` y una búsqueda recursiva tipo `find`.
- Semana 14: OSTEP cap. 40 *File System Implementation*; sesión B: cap. 41 (FFS) y 42 (*Crash Consistency*).

Entregas

Esta semana se entrega P4 (`pzip`). Se publica P5: archivos grandes y enlaces simbólicos en xv6, para la semana 16. El bono `cow` sigue abierto hasta la semana 16.

Preguntas para pensar

¿Cuántas E/S cuesta una escritura pequeña en RAID 4, y por qué RAID 5 escala mejor con escrituras aleatorias? ¿Por qué se llama `unlink` y no `delete`? Si dos procesos abren el mismo archivo por separado, ¿comparten el offset? ¿Y un padre y su hijo después de `fork`?