Capítulo 22 — File Descriptors: del Kernel a Sistemas Concurrentes de Alto Rendimiento

> *Capítulo avanzado, pero **escrito para que se entienda de verdad**. Se apoya en memoria dinámica (`09`), structs (`08`) y archivos (`15`). También asume que ya compilaste programas con* `gcc` .
>
> *Este capítulo te saca de la comodidad de* `FILE*` *y te mete directo al kernel:* `open` , `read` , `write` , `lseek` , `flock` , `O_NONBLOCK` , `select` , `poll` , `epoll` , `dup` / `dup2` . *Aquí mandás vos. El kernel obedece.*

Cómo leer este capítulo

Llegó el momento en que `FILE*` deja de ser suficiente. `fopen` , `fgets` , `printf` son comodísimos, pero esconden lo que realmente importa: **los file descriptors**. Son números enteros pequeños que le decís al kernel "accedé a este recurso" y el kernel **obedece sin preguntar**.

Este capítulo va de cero (qué es un fd) hasta alto rendimiento (E/S no bloqueante, multiplexación, concurrencia con bloqueos). Si entendés cada parte, vas a poder escribir programas que leen, escriben y comunican datos **sin intermediarios**, directamente con el sistema operativo.

Está dividido en **nueve partes**. Después de cada concepto hay una o más prácticas —uno o varios ejercicios— para que los uses ya mismo. **No las saltees**: son la mitad del capítulo.

Parte	De qué habla	La idea en una frase
1	¿Qué es un fd?	Un file descriptor es solo un número entero que le dice al kernel "a qué recurso quiero acceder".
2	`open`, `close`, `read`, `write`	Las cuatro funciones primitivas: abrir, leer, escribir y cerrar sin intermediarios de `stdio`.
3	Flags y permisos (`O_CREAT`, `O_TRUNC`, `O_APPEND`, `mode_t`)	El contrato con el kernel define creación atómica, truncamiento y seguridad de acceso.
4	`lseek`: manipulación directa del cursor	El puntero de posición del kernel es tuyo: saltá, medí y posicionate a voluntad sin búferes.
5	Concurrencia y bloqueos (`fcntl` / `flock`)	Dos procesos escribiendo al mismo archivo destruyen los datos si no usan bloqueos advisory.
6	E/S no bloqueante (`O_NONBLOCK`) y multiplexación (`select`, `poll`, `epoll`)	No te quedes colgado esperando E/S: el kernel te avisa mediante descriptores listos para operar.
7	`/proc/self/fd/`: el mapa interno del kernel	Linux te expone el estado de tus descriptores en tiempo real como un sistema de archivos virtual.
8	`dup` y `dup2`: clonación y redirección	Dos números apuntando al mismo recurso: la clave absoluta de cómo los shells redirigen stdout/stderr.
9	Ejercicio jefe: demonio de logs concurrente, rotativo y multicanal	Unifica todo: llamadas crudas, bloqueos contra procesos concurrentes, multiplexación y rotación atómica.

> ***Regla de oro del libro:*** *el capítulo explica y muestra* ***cómo funciona*** *cada system call, pero* ***los ejercicios no traen la solución****. El enunciado te da el problema; la respuesta la escribís vos.*

PARTE 1 — ¿Qué es un file descriptor?

Antes de tocar código, entendamos qué estás pidiéndole al kernel cuando decís "abrí este archivo". Porque la respuesta no es lo que pensás.

1.1 El fd: solo un número entero positivo

Cuando llamás a `fopen()`, te devuelve un puntero (`FILE *`). Por debajo, ese puntero apunta a una estructura en tu espacio de memoria que contiene **un número entero**: el file descriptor (fd).

Un file descriptor es **simple**. Demasiado simple para lo poderoso que es:

> ***Es solo un `int` mayor o igual a cero.*** *El kernel te asigna el próximo fd libre cuando llamás a `open()`, y te lo devuelve. Vos lo usás como un "handle" para decirle al kernel "quería leer de este recurso". Listo. Nada de punteros, nada de estructuras: solo un número.*

```
fopen("datos.txt", "r") → FILE* → { fd = 3, ... }
open("datos.txt", O_RDONLY) → directamente el número: 3
```

Por defecto, **todo proceso tiene tres fds abiertos al arrancar**:

Número	Nombre	Uso por defecto
`0`	`stdin`	Entrada estándar (teclado / redirección)
`1`	`stdout`	Salida estándar (pantalla / redirección)
`2`	`stderr`	Mensajes de error (siempre pantalla, a menos que lo desvíes)

El próximo `open()` te va a devolver `3`. El siguiente, `4`. Y así. El kernel siempre asigna el **fd más bajo disponible**.

> ***¿Por qué empiezan en 0?*** *Porque el estándar C los reservó para `stdin`, `stdout` y `stderr` mucho antes de que existiera Linux. Cuando Linux llegó, simplemente respetó esa convención: cero, uno y dos ya estaban "ocupados" semánticamente. Si empezaban desde cero, `open()` empieza en tres.*

1.2 ¿Qué recursos tienen fd?

Un file descriptor **no es solo para archivos**. Es un handle genérico del kernel para **cualquier recurso I/O**:

- Un archivo en disco (`/etc/passwd` , `/home/usuario/documento.txt`)
- Una tubería (pipe) entre procesos
- Un socket de red (TCP, UDP, Unix domain)
- Un dispositivo (`/dev/null` , `/dev/tty`)
- Un pipe named (FIFO)
- Un descriptor de memoria compartida

En Linux, todo es un archivo. Esa es la filosofía del sistema. Cuando abris un socket con `socket()` , el kernel te devuelve un fd igual que si hubieras abierto un archivo con `open()` . La diferencia está en la **semántica** de lo que leás/escribas por ese fd, no en cómo lo accedés.

```
open("archivo.txt", O_RDONLY)    → fd = 3     (disco)
socket(AF_INET, SOCK_STREAM, 0)  → fd = 4     (red)
pipe(fd_array)                   → fd[0]=5, fd[1]=6  (IPC entre procesos)
```

1.3 El contrato con el kernel

El trato es simple y brutalmente eficiente:

1. **Vos llamás a** `open()` con una ruta y flags. El kernel verifica permisos, busca el inode y te devuelve un fd. Costo de la llamada: algunas decenas de ciclos de CPU + acceso a disco si hace falta buscar en el sistema de archivos.
2. **Usás el fd** para `read()` o `write()` . El kernel mantiene una tabla interna por proceso que mapea cada fd al recurso subyacente (estructura de archivo del kernel).
3. **Llamás a** `close()` cuando terminás. El kernel libera la entrada en tu tabla y decrementa el contador referenciado del recurso.

No hay punteros, no hay estructuras que manejes vos, no hay "contexto" que lleves entre llamadas. Solo un número. **Un número entero.** Y eso es lo que hace a las syscalls de Linux tan rápidas y tan elegantes: el kernel solo necesita ese `int` para saber exactamente qué recurso estás tocando.

La tabla de descriptores por proceso

Cada proceso tiene su propia tabla de fds. El fd `3` del proceso A puede apuntar a un archivo diferente que el fd `3` del proceso B. Esto es porque **el fd no es un identificador global**: solo tiene sentido dentro del contexto de un proceso.

```
Proceso A:     Proceso B:

fd 0 → stdin   fd 0 → stdin
fd 1 → stdout  fd 1 → stdout
fd 2 → stderr  fd 2 → stderr
fd 3 → /tmp/a.txt    fd 3 → /var/log/syslog
fd 4 → pipe[5→6]     fd 4 → socket{192.168.1.1:80}
```

Dos procesos diferentes pueden tener el mismo número de fd apuntando a recursos completamente distintos. El kernel los mantiene aislados gracias a que cada uno tiene **su propia tabla**.

Práctica de la Parte 1

Ejercicio 22.1 — ¿Qué número soy?

Conceptos · file descriptor, `/proc/self/fd/`, `open`, `close`

Contexto · Para entender qué es un fd y cómo el kernel lo asigna, nada mejor que **mirar los propios fds de tu proceso**. Linux expone esa información como un sistema de archivos virtual en `/proc/self/fd/`. Cada entrada es un enlace simbólico que apunta al recurso real.

Consigna · Escribí un programa que:

1. Imprima el valor de `stdin` (`0`), `stdout` (`1`) y `stderr` (`2`).
2. Abra un archivo de prueba con `open()` y muestre qué fd le asignó el kernel.
3. Muestre los enlaces simbólicos de `/proc/self/fd/` para que el usuario vea a dónde apunta cada uno.

Entrada · Ninguna: el programa no lee nada. **Salida** · Los tres fds por defecto y el fd del archivo abierto, junto con la salida de `ls -l /proc/self/fd/`.

Escenario de prueba

```
$ gcc 22_01.c -o inspect_fds && ./inspect_fds
stdin  = 0
stdout = 1
stderr = 2

Archivo abierto → fd = 3

Enlaces en /proc/self/fd/:
lrwx------ 1 user user 64 Nov  5 10:30 0 -> /dev/pts/0
lrwx------ 1 user user 64 Nov  5 10:30 1 -> /dev/pts/0
lrwx------ 1 user user 64 Nov  5 10:30 2 -> /dev/pts/0
lrwx------ 1 user user 64 Nov  5 10:30 3 -> ./archivo_prueba.txt
```

> *El fd del archivo (`3` en este caso) apunta al archivo que abriste con `open()`. Los fds `0`, `1` y `2` apuntan a tu terminal (`/dev/pts/0` o similar).*

Criterios de aceptación

- [ ] Imprime correctamente los valores de `stdin`, `stdout` y `stderr`.
- [ ] Usa `open()` para abrir un archivo y muestra el fd devuelto.
- [ ] Muestra los enlaces simbólicos de `/proc/self/fd/` (puede usar `system("ls -l /proc/self/fd/")` o leer con `readdir`).
- [ ] Cierra el archivo con `close()` antes de terminar.

PARTE 2 — Las cuatro funciones primitivas: `open`, `close`, `read`, `write`

Con `FILE*` usás `fopen`, `fclose`, `fread` y `fwrite`. Con file descriptors directamente, las funciones son **cinco letras**: `open`, `close`, `read`, `write`. Pero tienen un poder que `stdio` no tiene: **control total**.

2.1 `open` : abrir sin intermediarios

```c
#include <fcntl.h>
#include <unistd.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- **`pathname`** : la ruta del archivo.
- **`flags`** : uno o más de `O_RDONLY` , `O_WRONLY` , `O_RDWR` . Opcionalmente combinados con `\|` con `O_CREAT` , `O_TRUNC` , `O_APPEND` , `O_EXCL` , etc.
- **`mode`** (solo si usás `O_CREAT`): los permisos del archivo nuevo (`0644` , `0600` , etc.).
- **Devuelve**: un fd entero positivo o `-1` en caso de error.

A diferencia de `fopen` , que recibe un string como `"r"` o `"w"` , `open` usa **banderas binarias**. Esto es importante: los flags son bits que combinás con OR, no strings arbitrarios. Cada combinación tiene un significado preciso definido por POSIX. No hay ambigüedad como `"a+"` vs `"r+"` en `stdio` .

```c
int fd = open("datos.txt", O_RDONLY);          // solo lectura
int fd = open("datos.txt", O_WRONLY);           // solo escritura
int fd = open("datos.txt", O_RDWR);             // lectura y escritura
int fd = open("nuevo.txt", O_WRONLY | O_CREAT, 0644); // crear si no existe
```

Ojo con `O_CREAT` sin `mode` . *Si llamás a `open()` con `O_CREAT` pero* ***sin*** *el tercer argumento (`mode`), el resultado es* ***comportamiento indefinido****. El compilador no te puede avisar porque `open()` tiene un número variable de argumentos (`...`): si pasás menos de lo esperado, se rompe.* ***Siempre*** *pasá el modo cuando uses `O_CREAT` .*

2.2 `close` : liberar el descriptor

```c
int close(int fd);
```

- Recibe el fd a cerrar.
- Devuelve `0` si salió bien, `-1` en caso de error (fd inválido, por ejemplo).

Cuando cerrás un fd, el kernel libera la entrada en tu tabla de descriptores y decrementa el contador referenciado del recurso subyacente (estructura de archivo, socket, etc.). Si es el último fd apuntando al recurso, se libera completamente.

Si no cierras un fd: el proceso no muere, pero consumís uno de los fds disponibles (normalmente 1024 por defecto en Linux). Cuando se agotan, `open()` falla con `EMFILE`.

```
if (close(fd) == -1) {
    perror("close");  // "close: Bad file descriptor"
}
```

2.3 `read`: sacar datos del fd

```
ssize_t read(int fd, void *buf, size_t count);
```

- Recibe el fd de donde leer, un buffer destino y la cantidad máxima de bytes a leer.
- Devuelve **la cantidad de bytes leídos**. Si devuelve `0`, llegaste al fin del archivo (`EOF`). Si devuelve `-1`, hubo un error (ver `errno`).
- **No garantiza** que lea `count` bytes: puede devolver menos si el recurso no tiene más datos disponibles todavía (es el caso de pipes, sockets y dispositivos).

```
char buffer[4096];
ssize_t n = read(fd, buffer, sizeof(buffer));
if (n == -1) {
    perror("read");
} else if (n == 0) {
    // fin del archivo / fin del pipe cerrado
} else {
    // leí n bytes en buffer[0..n-1]
}
```

El patrón de lectura completa

Para leer todo un archivo, el bucle clásico es:

```c
char buf[4096];
ssize_t total = 0;
ssize_t n;
while ((n = read(fd, buf, sizeof(buf))) > 0) {
    total += n;
}
if (n < 0) {
    perror("read");
}
```

Este bucle lee bloques de 4096 bytes hasta que `read` devuelve `0` (EOF) o `-1` (error). Cada llamada puede devolver menos de 4096 si el recurso no tiene más datos disponibles todavía. Es **normal** en pipes, sockets y TTYs: el kernel entrega lo que hay disponible y te avisa que podés volver a leer cuando haya más datos.

2.4 `write` : poner datos en el fd

```c
ssize_t write(int fd, const void *buf, size_t count);
```

- Recibe el fd destino, un buffer fuente y la cantidad de bytes a escribir.
- Devuelve **la cantidad de bytes escritos** (normalmente igual a `count`). Puede devolver menos en algunos recursos (pipes, sockets), en cuyo caso debés reintentar con los bytes restantes.
- Devuelve `-1` en caso de error.

```c
const char *msg = "Hola desde syscalls\n";
ssize_t n = write(fd, msg, strlen(msg));
if (n < 0) {
    perror("write");
} else if ((size_t)n != strlen(msg)) {
    // escribiste menos de lo que querías — reintentá con los bytes restantes
}
```

El patrón de escritura completa

```
ssize_t total = 0;
while (total < len) {
    ssize_t n = write(fd, buf + total, len - total);
    if (n < 0) {
        perror("write");
        break;
    }
    total += n;
}
```

Este patrón garantiza que se escriban todos los bytes incluso si `write` no los escribe de una sola vez.

2.5 stdio vs syscalls crudas: comparación directa

Operación	Con `stdio` (`FILE*`)	Sin intermediarios (fd)
Abrir	`fopen("f", "r")` → `FILE*`	`open("f", O_RDONLY)` → `int`
Leer	`fread(buf, 1, n, fp)`	`read(fd, buf, n)`
Escribir	`fwrite(buf, 1, n, fp)`	`write(fd, buf, n)`
Cerrar	`fclose(fp)`	`close(fd)`
Búfer	Automático (típicamente 8 KB)	**No hay búfer** — va directo al kernel
Error	`ferror(fp)` , `feof(fp)`	Retorna `-1` , revisá `errno`

La diferencia fundamental: **`stdio` tiene un búfer en tu espacio de usuario**. Cuando llamás a `fread()` , puede que `stdio` haya pedido 8 KB al kernel de una vez y te esté dando trozos de ahí. Con `read()` directo, cada llamada es **otra syscall** al kernel: más control pero potencialmente más llamadas de sistema.

> ***¿Cuándo usar uno u otro?*** *Si solo trabajás con archivos locales,* `stdio` *es rápido y cómodo porque el búfer reduce syscalls. Si necesitás control fino (no bloqueante, multiplexación, fd compartidos entre procesos), las syscalls crudas son tu única opción.*

Práctica de la Parte 2

Ejercicio 22.2 — Copiadora cruda

Conceptos · `open`, `close`, `read`, `write`, bucle de lectura/escritura, `errno`

Contexto · Querés copiar un archivo sin usar ni una sola función de `stdio.h`. Solo syscalls crudas y un búfer en tu pila. Este es el patrón fundamental que vas a repetir por todo este capítulo.

Consigna · Escribí un programa que reciba dos argumentos por línea de comandos: archivo origen y archivo destino. Copiá el contenido del origen al destino usando `open`, `read` y `write`. Usa un búfer fijo de 4096 bytes en la pila.

Entrada · Dos rutas por argumento: `./copiadora original.txt copia.txt`

Salida · Un archivo nuevo con el mismo contenido exacto que el origen. Imprimí `"Copiado: N bytes"` al final.

Escenario de prueba

```
$ echo "hola mundo desde syscalls" > original.txt
$ gcc 22_02.c -o copiadora && ./copiadora original.txt copia.txt
Copiado: 31 bytes
$ diff original.txt copia.txt
(no output — los archivos son idénticos)
```

Criterios de aceptación

- [ ] Abre el archivo origen con `open()` y verifica que no devuelve `-1`.
- [ ] Crea el archivo destino con `open(O_WRONLY | O_CREAT | O_TRUNC, 0644)`.
- [ ] Lee en bloques de hasta 4096 bytes con un bucle `while ((n = read(...)) > 0)`.
- [ ] Escribe cada bloque leído con `write()` al destino.
- [ ] Imprime la cantidad total de bytes copiados al finalizar.
- [ ] Cierra ambos fds con `close()`.

Ejercicio 22.3 — Lector por bloques fijos

Conceptos · `read`, manejo de retornos parciales, EOF como `0`

Contexto · A veces querés leer **exactamente N bytes**, incluso si el recurso no los tiene todos de una vez. Esto es común con sockets y pipes: el kernel puede entregar 1 byte de los 10 que pediste.

Consigna · Escribí una función `read_exact(int fd, void *buf, size_t count)` que lea **exactamente** `count` **bytes** del fd, reintentando con `read()` hasta completar el buffer o llegar a EOF/error.

Entrada · Un archivo de texto (podés usar el creado en 22.2). **Salida** · El contenido completo leído en un buffer.

Criterios de aceptación

- ☐ La función reintentá `read()` hasta completar los bytes pedidos o encontrar EOF/error.
- ☐ Retorna la cantidad total de bytes leídos (o `-1` si hubo error).
- ☐ Maneja correctamente el caso donde `count` es `0`.
- ☐ No hace más syscalls de las necesarias.

PARTE 3 — Flags y permisos: el contrato con el kernel

`open()` no solo abre archivos: puede crearlos, truncarlos, agregarles al final o fallar si ya existen. Todo eso se controla con **flags** y **permisos**. Es un contrato binario con el kernel: le decís exactamente qué querés hacer y el kernel lo hace **o falla**.

3.1 Los flags de acceso: cómo abrí el recurso

Los tres flags principales son mutuamente excluyentes: elegí **uno solo**:

Flag	Significado	Usá cuando...
`O_RDONLY`	Solo lectura	Querés leer, no escribir
`O_WRONLY`	Solo escritura	Querés escribir, no leer
`O_RDWR`	Lectura y escritura	Necesitás ambos

Combinados con OR (`|`) podés agregar comportamientos adicionales:

Flag	Efecto	Nota
`O_CREAT`	Crea el archivo si no existe	Requiere tercer argumento `mode`
`O_TRUNC`	Trunca (borra) el contenido existente	Solo con `O_WRONLY` o `O_RDWR`
`O_APPEND`	Escribe al final siempre	Cada `write()` se va al final automáticamente
`O_EXCL`	Falla si el archivo ya existe	Solo tiene sentido con `O_CREAT`; evita condiciones de carrera en la creación
`O_NONBLOCK`	Abre sin bloquear (ver Parte 6)	Para E/S no bloqueante

La combinación más común es:

```c
// Abrir existente para lectura
int fd = open("datos.txt", O_RDONLY);

// Crear o truncar para escritura
int fd = open("nuevo.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

// Agregar al final (no truncá)
int fd = open("log.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
```

> ***Trampa N.º 1:*** *`O_TRUNC` borra **todo** el contenido del archivo antes de abrirlo. Si abrís con `O_RDWR | O_TRUNC`, no solo vacías el archivo sino que podés corromper datos si otro proceso lo tiene abierto. Siempre pensá dos veces antes de combinar `O_TRUNC` con `O_RDWR`.*

3.2 Permisos: `mode_t` y los octales de Linux

Cuando creás un archivo con `O_CREAT`, necesitás especificar **permisos** en el tercer argumento. Los permisos de Linux se representan como una máscara de 9 bits (más los setuid/setgid/sticky bits):

```
0755 = rw-r--r--  (propietario lee+escribe, grupo y otros solo leen)
0644 = rw-r--r--  (el permiso más común para archivos normales)
0600 = rw-------  (solo el propietario puede leer/escribir)
```

Cada dígito octal representa tres bits:

Dígito	Significado binario	Permisos
`0`	`000`	Sin permisos
`1`	`--x`	Ejecución sola
`2`	`-w-`	Escritura sola
`3`	`-wx`	Escritura + ejecución
`4`	`r--`	Lectura sola
`5`	`r-x`	Lectura + ejecución
`6`	`rw-`	Lectura + escritura
`7`	`rwx`	Lectura + escritura + ejecución

El orden es: **propietario**, **grupo**, **otros**. Por ejemplo, `0644`:

```
Propietario: 6 = rw-   (puede leer y escribir)
Grupo:       4 = r--   (solo puede leer)
Otros:       4 = r--   (solo pueden leer)
```

> ***Ojo con los ceros iniciales.*** *En C, un número que empieza con `0` se interpreta como **octal**, no decimal. `0644` es el valor correcto. Si escribís `644`, estás pidiendo permisos binarios `1010000100` (muy diferente a lo que querés).*

3.3 `O_APPEND` : atomicidad de escritura al final del archivo

Cuando usás `O_APPEND`, el kernel mueve automáticamente el puntero de posición **al final del archivo** antes de cada `write()`. Esto garantiza que **cada write se va al final sin pisar nada**, incluso si múltiples procesos escriben al mismo tiempo.

> ***¿Por qué importa?*** *Sin `O_APPEND`, dos procesos podrían hacer `write()` y los bytes se entrelazarían o pisan datos del otro. Con `O_APPEND`, el kernel garantiza atomicidad a nivel de cada syscall: cada escritura va al final sin interferir con otras.*

3.4 `O_EXCL` : creación atómica, sin condiciones de carrera

Si dos procesos llaman a `open("archivo", O_CREAT | O_WRONLY)` al mismo tiempo y el archivo **no existe**, ambos podrían crearlo y uno pisaría al otro. Con `O_EXCL`, la combinación `O_CREAT | O_EXCL` **falla si el archivo ya existe**, garantizando que solo un proceso logra crearlo:

```c
int fd = open("mi_archivo.txt", O_WRONLY | O_CREAT | O_EXCL, 0644);
if (fd == -1 && errno == EEXIST) {
    // otro proceso ya creó el archivo — manejá el error
}
```

Esto es la base de los **candados por archivo** en sistemas distribuidos: "yo gano si `open(O_EXCL)` funciona, el otro pierde si devuelve `EEXIST`".

Práctica de la Parte 3

Ejercicio 22.4 — Creador con permisos estrictos

Conceptos · `O_CREAT`, `mode_t`, `O_EXCL`, `errno`, `perror` / `strerror`

Contexto · Querés crear un archivo que **solo el propietario pueda leer**, pero si ya existe querés fallar en vez de abrirlo para escritura. Este es el patrón típico de configuración segura: archivos de credenciales o claves deben ser inaccesibles para otros usuarios.

Consigna · Escribí un programa que reciba dos argumentos por línea de comandos: ruta del archivo y una cadena a escribir. El programa debe:

1. Crear el archivo con `O_WRONLY | O_CREAT | O_EXCL` y permisos `0600`.
2. Si el archivo ya existe, mostrar un error claro.
3. Escribir la cadena recibida en el archivo.
4. Mostrar los permisos reales del archivo creado (usando `ls -l` vía `system()` o leyendo `/proc/self/fd/`).

Entrada · Ruta y cadena: `./creador_seguro ./secret.txt "contraseña123"`

Salida · El archivo creado con permisos `0600`, la cadena escrita y confirmación.

Escenario de prueba

```
$ gcc 22_04.c -o creador_seguro && ./creador_seguro ./secret.txt "contraseña123"
Archivo ./secret.txt creado con éxito (permisos: 0600)
$ ls -l ./secret.txt
-rw------- 1 user user 15 Nov  5 10:30 ./secret.txt
$ cat ./secret.txt
contraseña123
$ ./creador_seguro ./secret.txt "otra cosa"
Error: no se pudo crear ./secret.txt: File exists (El archivo ya existe)
```

Criterios de aceptación

- [ ] Usa `O_CREAT | O_EXCL` para creación atómica.
- [ ] Permisos `0600` (`rw-------`).
- [ ] Detecta y reporta correctamente el error `EEXIST`.
- [ ] Verifica que el retorno de cada syscall no sea `-1`.

Ejercicio 22.5 — Comportamiento atómico de `O_APPEND`

Conceptos · `O_APPEND`, escritura concurrente, condiciones de carrera sin bloqueo

Contexto · Sin `O_APPEND`, dos procesos escribiendo al mismo archivo pueden entrelazar sus bytes. Querés **demostrar** que ocurre el caos y luego **mostrar** que `O_APPEND` lo resuelve.

Consigna · Escribí un programa que reciba una ruta de archivo como argumento y:

1. Abra el archivo en modo `O_WRONLY | O_CREAT` (**sin** `O_APPEND`).
2. Escriba un mensaje corto con su PID incluido (ejemplo: `"Hijo-42\n"`).
3. Repita esto 5 veces con una pequeña pausa (`usleep(100)`) entre escrituras.

Luego, corré este programa **dos veces en paralelo** sobre el mismo archivo y observá cómo se entrelazan los mensajes. Después repetí con `O_APPEND` y observá que cada escritura queda al final del todo.

Entrada · Ruta de un archivo (ejemplo: `./append_test log.txt`). **Salida** · El contenido del archivo tras correr el programa dos veces en paralelo.

Criterios de aceptación

- ☐ Primero demostra la condición de carrera **sin** `O_APPEND`.
- ☐ Luego demostrá que con `O_APPEND` los mensajes no se entrelazan (cada PID aparece completo y al final).
- ☐ Usa `getpid()` para identificar cada proceso en el mensaje.

PARTE 4 — `lseek`: manipulación directa del cursor en el kernel

Con `stdio`, si querés saber dónde estás parados en un archivo usás `ftell()`. Con syscalls crudas, el control es aún más directo: **movés el puntero del kernel** donde quieras.

4.1 ¿Por qué mover el puntero?

Imaginá que tenés un archivo grande y solo querés leer los últimos 64 bytes sin leer todo lo anterior. Con `stdio`, tendrías que hacer `fseek(fp, -64, SEEK_END)` — que por debajo hace exactamente esto: `lseek(fd, -64, SEEK_END)`.

`lseek` te permite:

1. **Saltar** a cualquier posición sin leer los bytes intermedios.
2. **Medir** el tamaño de un archivo moviéndote al final.
3. **Preposicionar** el cursor para escrituras secuenciales (ejemplo: escribir al inicio sin truncar).
4. **Multiplexar** con `select` / `poll` (ver Parte 6): saber si hay datos disponibles en una posición concreta.

4.2 La función

```c
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

- `fd` : el file descriptor del archivo.
- `offset` : desplazamiento en bytes (puede ser negativo).
- `whence` : punto de referencia para el desplazamiento:
  - `SEEK_SET` : desde el inicio del archivo (`offset` bytes desde el byte 0)
  - `SEEK_CUR` : desde la posición actual (`posición + offset`)
  - `SEEK_END` : desde el final del archivo (`tamaño + offset`)

Devuelve **la nueva posición** del cursor, o `-1` en caso de error.

```c
// Ir al inicio para leer desde el principio
lseek(fd, 0, SEEK_SET);

// Ir 10 bytes antes del final para leer los últimos 10
lseek(fd, -10, SEEK_END);

// Posicionarse 5 bytes después de donde estoy parado.
lseek(fd, 5, SEEK_CUR);
```

> ***Importante:*** *`lseek` con `offset = 0` y `whence = SEEK_END` devuelve el* ***tamaño del archivo****. Es el truco universal para averiguar cuánto mide un archivo sin leerlo: vas al final, preguntás dónde llegaste y volvés atrás.*

4.3 El patrón de obtener el tamaño de un archivo

```c
// Ir al final
lseek(fd, 0, SEEK_END);
// Preguntar dónde estamos (eso es el tamaño)
long size = lseek(fd, 0, SEEK_CUR);
// Volver al inicio para leer desde el principio
lseek(fd, 0, SEEK_SET);
```

Este patrón funciona con **cualquier** fd que soporte posicionamiento: archivos en disco, pero no así pipes, sockets o TTYs. En esos recursos, `lseek` devuelve `-1` con `errno == ESPIPE`.

> ***¿Por qué `SEEK_END` + `SEEK_CUR` y no directamente `ftell`?*** *Porque estamos en el mundo de las syscalls crudas. No hay `ftell`. `lseek(fd, 0, SEEK_CUR)` es el equivalente directo: preguntar al kernel "¿dónde estoy parado ahora?" sin moverme.*

4.4 Escribir sin truncar: preposicionar con `lseek`

Si abres un archivo con `O_WRONLY` (sin `O_TRUNC`), podés usar `lseek` para escribir en cualquier posición sin borrar lo que había antes:

```c
// Abrir existente para escritura (sin truncar)
int fd = open("datos.txt", O_WRONLY);

// Ir al byte 100 del archivo
lseek(fd, 100, SEEK_SET);

// Escribir a partir de ahí — los primeros 100 bytes quedan intactos.
const char *data = "reemplazo\n";
write(fd, data, strlen(data));
```

Esto es útil para actualizar registros de tamaño fijo en un archivo: buscás el registro con `lseek`, lo sobreescribís con `write` y listo. No tocás nada más.

Práctica de la Parte 4

Ejercicio 22.6 — Buscador de ocurrencias binarias

Conceptos · `lseek`, `SEEK_SET`, `read`, búsqueda en archivo binario

Contexto · Querés encontrar todas las apariciones de una secuencia de bytes dentro de un archivo binario grande. Para hacerlo eficiente, leés bloques y usás `lseek` para reposicionar sin volver al inicio cada vez.

Consigna · Escribí un programa que reciba como argumentos: ruta del archivo, patrón a buscar (hex) y muestre la posición (offset en bytes) de cada ocurrencia del patrón dentro del archivo.

Entrada · Ruta del archivo + patrón hexadecimal (ejemplo: `./buscador datos.bin DEADBEEF`)

Salida · Lista de offsets donde se encontró el patrón.

Criterios de aceptación

- ☐ Lee el archivo en bloques usando `read()`.
- ☐ Usa `lseek` para reposicionar correctamente entre bloques (manejando superposición del patrón).
- ☐ Compara bytes crudos contra el patrón.
- ☐ Imprime cada offset encontrado.

Ejercicio 22.7 — Cálculo de tamaño con `lseek` y `SEEK_END`

Conceptos · `lseek`, `SEEK_END`, `SEEK_CUR`, cálculo de tamaño

Contexto · No hay función estándar para obtener el tamaño de un fd sin abrirlo como `FILE*`. El patrón universal es moverse al final con `SEEK_END` y preguntar la posición con `SEEK_CUR`.

Consigna · Escribí una función `get_file_size(int fd)` que devuelva el tamaño en bytes del archivo al que apunta `fd`, asumiendo que el fd ya fue abierto. El fd debe quedar reposicionado al final después de la llamada (no hay problema).

Criterios de aceptación

- [ ] Usa `lseek(fd, 0, SEEK_END)` seguido de `lseek(fd, 0, SEEK_CUR)`.
- [ ] Devuelve el tamaño como `long` o `off_t`.
- [ ] Maneja correctamente archivos vacíos (tamaño 0).

PARTE 5 — Concurrencia y bloqueos (`fcntl` / `flock`)

Dos procesos escribiendo al mismo archivo sin coordinación = datos corrompidos. Linux ofrece mecanismos de bloqueo advisory que resuelven este problema.

5.1 El problema: condiciones de carrera en escritura

Imaginá dos procesos ejecutando esto simultáneamente:

```
// Proceso A                    // Proceso B
open("shared.log", O_WRONLY | O_APPEND);   open("shared.log", O_WRONLY | O_APPEND);
write(fd, "A\n", 2);             write(fd, "B\n", 2);
close(fd);                       close(fd);
```

Sin `O_APPEND`, el resultado en el archivo podría ser `"AB\n"` (mezclado) o `"BA\n"`. Con `O_APPEND` el kernel garantiza atomicidad por syscall, pero si necesitás leer-modificar-escribir de forma atómica, necesitás bloqueos.

5.2 `flock`: bloques simples y prácticos

```c
#include <sys/file.h>

int flock(int fd, int operation);
```

- `LOCK_SH` : bloqueo compartido (varios procesos pueden tenerlo). Solo lectura.
- `LOCK_EX` : bloqueo exclusivo (solo un proceso puede tenerlo). Lectura/escritura.
- `LOCK_UN` : libera el bloqueo.
- `LOCK_NB` : bandera no bloqueante — si no se puede adquirir el bloqueo, devuelve `-1` con `EWOULDBLOCK` en vez de esperar.

```
// Bloqueo exclusivo: solo yo puedo escribir ahora.
flock(fd, LOCK_EX);
write(fd, "mi dato\n", 8);
flock(fd, LOCK_UN);

// Bloqueo compartido: cualquier lector puede tenerlo a la vez.
flock(fd, LOCK_SH);
read(fd, buf, sizeof(buf));
flock(fd, LOCK_UN);
```

Advisory vs mandatory. *Los bloqueos `flock` son **advisory**: el kernel solo los respeta si todos los procesos que acceden al archivo saben y usan `flock`. Si un proceso abre el archivo sin usar bloqueos, puede escribir lo que quiera. No hay coerción a nivel de sistema de archivos.*

5.3 `fcntl` : locking más granular

```
#include <fcntl.h>

struct flock {
    short l_type;     // F_RDLCK, F_WRLCK, F_UNLCK
    short l_whence;   // SEEK_SET, SEEK_CUR, SEEK_END
    off_t l_start;    // offset del inicio del lock
    off_t l_len;      // longitud del lock (0 = hasta EOF)
    pid_t l_pid;      // PID que tiene el bloqueo (solo al leer con F_GETLK)
};

int fcntl(int fd, int cmd, struct flock *lock);
```

`fcntl` permite bloquear **regiones específicas** de un archivo, no solo el archivo completo. Esto es útil cuando tenés archivos con múltiples registros y querés que los procesos accedan a diferentes registros sin interferirse.

```
struct flock lock;
lock.l_type   = F_WRLCK;
lock.l_whence = SEEK_SET;
lock.l_start  = 0;
lock.l_len    = 1024;       // bloqueo exclusivo de los primeros 1024 bytes

fcntl(fd, F_SETLK, &lock);  // intentar bloqueo (no bloquea si no puede)
// o bien:
fcntl(fd, F_SETLKW, &lock); // intentar bloqueo (bloquea hasta que pueda)
```

> ***`F_SETLK` vs `F_SETLKW`.*** *El sufijo `W` significa "wait" (esperar). `F_SETLK` intenta y falla inmediatamente si el bloqueo no está disponible. `F_SETLKW` bloquea hasta que consiga el lock.*

5.4 Buenas prácticas con bloqueos

1. **Bloqueá el menor tiempo posible.** El tiempo que mantengás un lock es tiempo que otros procesos quedan esperando. Hacé solo las operaciones necesarias dentro del lock y soltalo rápido.
2. **Siempre liberá en todos los caminos.** Si tenés salidas anticipadas (`return` , `break`) dentro de la sección crítica, asegurate de liberar el lock antes de cada una.
3. **El lock se libera al cerrar el fd o terminar el proceso.** Pero no dependas de eso: liberá explícitamente.

Práctica de la Parte 5

Ejercicio 22.8 — Escritor sincronizado con `flock`

Conceptos · `flock` , `LOCK_EX` , escritura atómica, condiciones de carrera

Contexto · Escribí un programa que cree N procesos hijos, cada uno escribiendo líneas en el mismo archivo. Sin bloqueos, las líneas se mezclan. Con `flock` , cada escritura queda intacta.

Consigna · Escribí un programa que:

1. Reciba por argumento la cantidad de writers y el archivo destino.
2. Cree N procesos hijos con `fork()`.
3. Cada hijo escribe 10 líneas en formato `"PID-XXXX\n"` (donde XXXX es un número único) al archivo, usando `flock(LOCK_EX)` antes de cada escritura y `flock(LOCK_UN)` después.
4. El padre espera a todos los hijos con `wait()`.

Criterios de aceptación

- ☐ Usa `fork()` para crear writers paralelos.
- ☐ Cada escritura individual está protegida por `flock(LOCK_EX)` / `flock(LOCK_UN)`.
- ☐ Las líneas en el archivo final no están mezcladas (cada línea es completa).
- ☐ El padre espera a todos los hijos con `wait()`.

Ejercicio 22.9 — Detector de condiciones de carrera

Conceptos · `flock`, `LOCK_EX`, `fcntl`, demostración de race condition

Contexto · Este ejercicio te pide que **demuestres** el problema sin bloqueos y su solución con ellos. Es un programa que podés correr de dos maneras: una corrupta, una correcta.

Consigna · Escribí un programa que reciba dos argumentos: `modo` (`sin-lock` o `con-lock`) y ruta de archivo. El programa abre el archivo, hace `flock(LOCK_EX)` si el modo es `con-lock`, escribe 3 veces con pausa intermedia de 10ms, y cierra.

Corré **cinco instancias en paralelo** del mismo programa (tres sin-lock, tres con-lock) y compará los resultados para ver la diferencia.

Criterios de aceptación

- ☐ Funciona tanto en modo `sin-lock` como `con-lock`.
- ☐ En modo `con-lock`, las escrituras no se superponen.
- ☐ En modo `sin-lock`, podés observar líneas mezcladas o truncadas.
- ☐ El programa informa qué modo está usando al arrancar.

PARTE 6 — E/S no bloqueante (`O_NONBLOCK`) y multiplexación (`select` , `poll` , `epoll`)

Imaginá que estás leyendo del teclado y el usuario todavía no escribió nada. Con `read()` normal, tu programa **se queda esperando** indefinidamente. No podés hacer otra cosa hasta que escriba.

La solución: **E/S no bloqueante**. Le decís al kernel "si no hay datos, avisame pero no me pares".

6.1 Activar `O_NONBLOCK`

Hay dos formas de activarlo:

```
// Opción 1: en open() (no funciona con todos los tipos de fd)
int fd = open("datos.txt", O_RDONLY | O_NONBLOCK);

// Opción 2: con fcntl() (funciona con todo tipo de fds)
int flags = fcntl(fd, F_GETFL);           // obtiene los flags actuales
fcntl(fd, F_SETFL, flags | O_NONBLOCK);   // agrega O_NONBLOCK
```

Cuando el fd está en modo no bloqueante, si llamás `read()` y **no hay datos**, devuelve `-1` con `errno == EAGAIN` o `EWOULDBLOCK` (son lo mismo en Linux). No se para: te avisa "ahora no hay nada".

```
char buf[256];
ssize_t n = read(fd, buf, sizeof(buf));
if (n > 0) {
    // hay datos — procesalos
} else if (n == -1 && (errno == EAGAIN || errno == EWOULDBLOCK)) {
    // no hay datos ahora. puedo hacer otra cosa y volver a intentarlo después.
} else if (n < 0) {
    // error real
    perror("read");
}
```

6.2 `select()` : la abuela de la multiplexación

`select()` te permite preguntar al kernel: "de estos fds, **¿cuáles están listos para leer/escribir?**" Le pasás un conjunto de fds y el kernel te devuelve cuáles cambiaron.

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

int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds, struct timeval
*timeout);
```

- **`nfds`** : el mayor fd + 1 (para saber cuántos fds chequear).
- **`readfds`** : fds que querés monitorear para lectura.
- **`writefds`** : fds que querés monitorear para escritura.
- **`errorfds`** : fds que querés monitorear para errores.
- **`timeout`** : cuánto esperar (o `NULL` = esperar indefinidamente, o `0` = no esperar nada).

Macros de manejo:

Macro	Qué hace
`FD_ZERO(fd_set *)`	Limpia el set
`FD_SET(int fd, fd_set *)`	Agrega un fd al set
`FD_CLR(int fd, fd_set *)`	Quita un fd del set
`FD_ISSET(int fd, fd_set *)`	Verifica si un fd está en el set (devuelve 1 sí)

```
fd_set reads;
FD_ZERO(&reads);
FD_SET(STDIN_FILENO, &reads);      // monitorear stdin
FD_SET(socket_fd, &reads);         // monitorear socket

struct timeval tv;
tv.tv_sec  = 5;    // 5 segundos
tv.tv_usec = 0;    // 0 microsegundos

int ready = select(max_fd + 1, &reads, NULL, NULL, &tv);

if (ready < 0) {
    perror("select");
} else if (ready == 0) {
```

```
    printf("timeout — nadie escribió en 5 segundos\n");
} else {
    // Hay fds listos. Verificar cuáles:
    if (FD_ISSET(STDIN_FILENO, &reads)) {
        read(STDIN_FILENO, buf, sizeof(buf));
    }
    if (FD_ISSET(socket_fd, &reads)) {
        read(socket_fd, buf, sizeof(buf));
    }
}
```

> ***Limitación de `select`:*** *el límite de fds que podés monitorear es limitado (típicamente 1024 en Linux). Si necesitás manejar miles de conexiones simultáneas, usá `poll` o `epoll`.*

6.3 `poll()`: sin límite de fds

`poll()` funciona como `select` pero usa una **struct** por fd en lugar de bitmaps fijos:

```
#include <poll.h>

struct pollfd {
    int   fd;       // file descriptor a monitorear
    short events;   // eventos que me importan
    short revents;  // eventos que ocurrieron (devuelve el kernel)
};

int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

Eventos comunes:

Evento	Significado
`POLLIN`	Hay datos para leer
`POLLOUT`	Podés escribir sin bloquear
`POLLERR`	Error en el fd
`POLLHUP`	Colgado (hung up)
`POLLNVAL`	fd no válido

```c
struct pollfd fds[2];
fds[0].fd   = STDIN_FILENO;
fds[0].events = POLLIN;
fds[1].fd   = socket_fd;
fds[1].events = POLLIN;

int ready = poll(fds, 2, -1);  // -1 = esperar indefinidamente

if (ready < 0) { perror("poll"); }
else if (ready > 0) {
    if (fds[0].revents & POLLIN) {
        read(STDIN_FILENO, buf, sizeof(buf));
    }
    if (fds[1].revents & POLLIN) {
        read(socket_fd, buf, sizeof(buf));
    }
}
```

6.4 `epoll()` : la evolución de Linux (bonus)

`select` y `poll` recopilan información sobre **todos** los fds que les pasás en cada llamada. Si tenés 10.000 conexiones activas y solo 3 tienen datos, desperdiciaste tiempo chequeando las otras 9.997.

`epoll` (Linux-specific) resuelve esto con un **árbol interno**: registrás los fds una vez y el kernel te avisa **solo** cuando cambian. No hay límite de fds, es O(1) por evento.

```c
int epfd = epoll_create1(0);              // crear instancia epoll
struct epoll_event ev;
ev.events  = EPOLLIN;                      // monitorear lectura
ev.data.fd = socket_fd;
epoll_ctl(epfd, EPOLL_CTL_ADD, socket_fd, &ev);  // registrar fd

struct epoll_event events[10];
int n = epoll_wait(epfd, events, 10, -1);  // esperar eventos

for (int i = 0; i < n; i++) {
    read(events[i].data.fd, buf, sizeof(buf));
}
```

> ***Nota:*** `epoll` *es específico de Linux. No está disponible en macOS o Windows. Si necesitás portabilidad entre sistemas, quedate con* `poll` *.*

Práctica de la Parte 6

Ejercicio 22.10 — Lector no bloqueante con `O_NONBLOCK`

Conceptos · `O_NONBLOCK` , `fcntl` , `EAGAIN` / `EWOULDBLOCK` , `errno`

Contexto · Escribí un programa que lea de stdin en modo no bloqueante y, mientras no hay datos, imprima un "." cada segundo como indicador de actividad.

Consigna · Configurar stdin como no bloqueante con `fcntl()` . En un bucle infinito: intentar leer; si no hay datos (`EAGAIN`), esperar 1 segundo e imprimir "."; si hay datos, procesarlos normalmente. Salir al detectar EOF o error.

Criterios de aceptación

- ☐ Usa `fcntl(fd, F_SETFL, flags | O_NONBLOCK)` para configurar stdin como no bloqueante.
- ☐ Cuando no hay datos, el programa no se para: espera 1 segundo e imprime ".".
- ☐ Cuando hay datos en stdin, los procesa normalmente (`read()` devuelve bytes reales).
- ☐ Termina limpiamente al detectar EOF o error real.

Ejercicio 22.11 — Monitor multicanal con `select` / `poll`

Conceptos · `select` , `poll` , `FD_SET` , `POLLIN` , multiplexación de descriptores

Contexto · Escribí un programa que monitoree stdin Y un pipe (entre procesos) al mismo tiempo. Cuando haya datos en cualquiera de los dos, leer y procesar del canal disponible. Si ningún canal tiene datos en 10 segundos, imprimir timeout.

Consigna · El programa crea un pipe con `pipe()` , luego se fork(). El padre escribe fechas/hora en el pipe cada segundo. El hijo usa `select()` (o `poll()`) para monitorear stdin Y el read-end del pipe. Cuando hay datos en stdin, los imprime como "Entrada usuario: ...". Cuando hay datos en el pipe, los imprime como "Reloj: ...". Si ninguno tiene datos en 10 segundos, imprime "timeout".

Criterios de aceptación

- [ ] Crea un pipe y fork() correctamente.
- [ ] El hijo usa `select()` o `poll()` para monitorear stdin + pipe-read-end simultáneamente.
- [ ] Distingue claramente entre datos de stdin vs datos del pipe en la salida.
- [ ] Maneja correctamente el timeout (10 segundos sin actividad).
- [ ] Bonus: usar ambos (`select` y `poll`) en secciones separadas para comparar.

PARTE 7 — `/proc/self/fd/` : el mapa interno del kernel

Linux te muestra sus propios recursos como si fueran archivos normales. `/proc/self/fd/` es un directorio virtual donde cada entrada es un **enlace simbólico** que apunta al recurso real de cada fd.

7.1 ¿Qué es `/proc/self/fd/` ?

```
$ ls -l /proc/self/fd/
lrwx------ 1 user user 64 Nov  5 10:30 0 -> /dev/pts/0
lrwx------ 1 user user 64 Nov  5 10:30 1 -> /dev/pts/0
lrwx------ 1 user user 64 Nov  5 10:30 2 -> /dev/pts/0
lrwx------ 1 user user 64 Nov  5 10:30 3 -> /home/user/archivo.txt
```

Cada entrada (`0` , `1` , `2` , `3` ...) es el fd. El `->` muestra a qué recurso apunta. Esto te permite **inspeccionar en tiempo real** los archivos que tu programa tiene abiertos, sin herramientas externas.

7.2 Leer `/proc/self/fd/` desde C

Podés abrir `/proc/self/fd/` como un directorio normal con `opendir()` / `readdir()` e inspeccionar cada symlink con `readlink()`:

```c
#include <dirent.h>
#include <limits.h>
#include <unistd.h>

DIR *dir = opendir("/proc/self/fd");
if (dir != NULL) {
    struct dirent *entry;
    while ((entry = readdir(dir)) != NULL) {
        // Solo entries numéricos (0, 1, 2, 3...)
        if (entry->d_name[0] >= '0' && entry->d_name[0] <= '9') {
            char path[PATH_MAX];
            snprintf(path, sizeof path, "/proc/self/fd/%s", entry->d_name);

            char target[PATH_MAX];
            ssize_t len = readlink(path, target, sizeof(target) - 1);
            if (len != -1) {
                target[len] = '\0';
                printf("fd %s -> %s\n", entry->d_name, target);
            }
        }
    }
    closedir(dir);
}
```

Este patrón es la base de herramientas de diagnóstico como `lsof` y `strace`. Te permite ver **qué archivos tiene abiertos tu proceso** en tiempo real.

7.3 Uso práctico: auditoría de recursos

¿Tu programa se queda sin fds? ¿Alguien dejó un archivo abierto por error? Con `/proc/self/fd/` podés:

- Contar cuántos fds tiene abiertos el proceso.
- Identificar fd's que apuntan a archivos temporales olvidados.
- Detectar leaks de memoria (archivos que deberían haberse cerrado).

Práctica de la Parte 7

Ejercicio 22.12 — Auditor y explorador de fds activos

Conceptos · `/proc/self/fd/` , `opendir` , `readdir` , `readlink`

Contexto · Escribí un programa que liste todos los fds abiertos por el propio proceso, mostrando el número de fd, su tipo (archivo, terminal, pipe) y el recurso al que apunta.

Consigna · El programa debe:

1. Abrir `/proc/self/fd/` como directorio con `opendir()` .
2. Recorrer cada entrada, ignorando "." y "..".
3. Para cada fd numérico, usar `readlink()` para obtener el recurso al que apunta.
4. Mostrar un reporte en formato legible: número de fd, tipo (archivo/terminal/pipe/socket) y ruta.

Criterios de aceptación

- ☐ Usa `opendir("/proc/self/fd/")` para listar los fds.
- ☐ Usa `readlink()` para obtener el recurso real de cada fd.
- ☐ Ignora entradas no numéricas.
- ☐ Muestra un reporte claro y ordenado por número de fd.

PARTE 8 — `dup` y `dup2` : clonación y redirección

Dos números apuntando al mismo recurso: eso es lo que hacen `dup()` y `dup2()` . Son la base absoluta de cómo los shells redirigen `stdout` a un archivo o conectan programas con tuberías.

8.1 `dup` : clonar un fd

```c
#include <unistd.h>

int dup(int oldfd);
```

Crea una **copia** del fd pasado como argumento. El nuevo fd apunta al **mismo recurso** que el original:

```
// Clonar stdout (fd 1) en fd 5
int fd5 = dup(1);  // ahora fd 1 y fd 5 apuntan al mismo archivo/terminal

// Escribir a cualquiera de los dos escribe en el mismo lugar.
write(fd5, "hola\n", 5);   // aparece en stdout porque fd5 == clon de stdout
write(1,       "mundo\n", 6);  // también aparece en stdout (mismo recurso)

close(fd5);  // liberar la copia cuando no la necesitemas.
```

> ***No es una copia del archivo.*** *Es una copia de la* ***entrada en la tabla de descriptores****. Ambos fds comparten el mismo puntero de posición, los mismos flags (incluido* `O_APPEND` *), y cualquier escritura por uno afecta lo que el otro ve.*

8.2 `dup2` : clonar y elegir el destino

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

Igual que `dup()` , pero vos elegís **cuál** será el nuevo fd:

- Si `newfd` ya estaba abierto, se cierra primero.
- Si `newfd == oldfd`, devuelve `oldfd` sin hacer nada (no hay error).
- Devuelve el nuevo fd o `-1` en error.

```
int save_stdout = dup(1);  // guardar una copia de stdout actual

// Redirigir stdout a un archivo
int log_fd = open("salida.log", O_WRONLY | O_CREAT | O_TRUNC, 0644);
dup2(log_fd, 1);           // ahora fd 1 (stdout) apunta al archivo

printf("Esto va al archivo, no a pantalla\n");  // stdout fue reemplazado por dup2!

dup2(save_stdout, 1);      // restaurar stdout original
close(save_stdout);
close(log_fd);
```

Este es el patrón de redirección de shells. *Cuando escribís* `comando > archivo` *en la terminal, lo que hace el shell internamente es exactamente esto: abre el archivo, usa* `dup2()` *para reemplazar fd 1 con el nuevo fd, y ejecuta el comando. El programa no sabe que stdout fue redirigido.*

8.3 Redirigir stderr también

El mismo patrón funciona para `stderr` (fd 2):

```
int err_fd = open("errores.log", O_WRONLY | O_CREAT | O_TRUNC, 0644);
dup2(err_fd, 2);   // stderr ahora apunta al archivo
perror("este error va a errores.log");
close(err_fd);
```

Y para redirigir stdout y stderr al mismo destino:

```
int log_fd = open("todo.log", O_WRONLY | O_CREAT | O_TRUNC, 0644);
dup2(log_fd, 1);   // stdout → archivo
dup2(log_fd, 2);   // stderr → mismo archivo
close(log_fd);      // el fd original ya no lo necesitamos (dos duplo comparten recurso)
```

Práctica de la Parte 8

Ejercicio 22.13 — Redirección manual de stdout a archivo

Conceptos · `dup` , `dup2` , redirección, `close`

Contexto · Querés que **todo** lo que tu programa escribe por `stdout` vaya a un archivo en vez de la pantalla. Usás `dup()` para guardar el stdout original y `dup2()` para reemplazarlo temporalmente.

Consigna · Escribí un programa que:

1. Guarde una copia del fd actual de stdout con `dup(1)` .
2. Abra un archivo `captura.txt` en modo escritura.
3. Use `dup2(log_fd, 1)` para redirigir stdout al archivo.
4. Escriba varias líneas por `printf` (que ahora van al archivo).
5. Restaure el stdout original con `dup2(saved_stdout, 1)` .
6. Imprima un mensaje de confirmación en pantalla.

Criterios de aceptación

- ☐ Usa `dup(1)` para guardar una copia del stdout original.
- ☐ Abre el archivo destino correctamente con `open()` .
- ☐ Usa `dup2()` para reemplazar fd 1 con el fd del archivo.
- ☐ Verifica que los `printf` van al archivo (no a pantalla).
- ☐ Restaura stdout con `dup2(saved, 1)` antes de terminar.

Ejercicio 22.14 — Canal invertido (fd a stdin)

Conceptos · `dup` , `dup2` , redirección inversa, pipe write-end → fdo

Contexto · Escribí un programa que cree una tubería con `pipe()` , escriba datos en el write-end y luego use `dup2()` para hacer que **stdin (fd 0)** apunte a ese mismo pipe. De esta forma, cuando el programa lee de stdin (con `read(0, ...)` o `getchar()`), está leyendo del pipe.

Consigna · El programa debe:

1. Crear un pipe con `pipe()`.
2. Escribir datos en el write-end del pipe.
3. Cerrar ambos ends originales del pipe.
4. Usar `dup2(pipe[0], 0)` para clonar el read-end sobre fd 0 (stdin).
5. Leer de stdin normalmente y mostrar lo que se leyó.

> *Esto es esencialmente cómo los shells implementan tuberías (`|`): conectan stdout de un proceso al stdin del siguiente usando `dup2()`.*

Criterios de aceptación

- [ ] Crea el pipe con `pipe()`.
- [ ] Escribe datos en el write-end antes de clonar.
- [ ] Usa `dup2(pipe[0], 0)` correctamente.
- [ ] Lee stdin y obtiene los datos del pipe.
- [ ] Cierra todos los fds originales del pipe después del dup.

PARTE 9 — Ejercicio jefe: demonio de logs concurrente, rotativo y multicanal

Llegó el momento de juntar todo lo que aprendiste en un solo programa que unifica llamadas crudas, bloqueos contra procesos concurrentes, multiplexación y rotación atómica.

9.1 El desafío completo

Escribí un **demonio de logs** que:

1. Escriba registros a un archivo usando **exclusivamente syscalls crudas** (`open` , `write` , `close`), con **cero uso de** `stdio.h` . Cada registro tiene el formato: `[TIMESTAMP] PID Mensaje\n` .
2. Implemente sincronización entre procesos usando `flock()` para evitar condiciones de carrera cuando múltiples instancias escriben simultáneamente al mismo archivo de log.
3. Soporte **rotación automática** cuando el archivo supera N KB (por ejemplo, 5 KB): renombra `app.log` → `app.log.1` y recrea el archivo de forma atómica (`rename()` + nuevo `open`).
4. Gestione entradas de comandos o señales de forma asíncrona mediante **E/S no bloqueante** (`O_NONBLOCK`) y supervisión de descriptores con `select()` o `poll()` . Los comandos válidos son:
   - `FLUSH` → imprime estadísticas (total de bytes escritos, cantidad de registros)
   - `STATUS` → muestra el estado actual del daemon (archivo activo, tamaño, rotaciones realizadas)
   - `QUIT` → cierra limpiamente todos los recursos y termina
5. Redireccione **stdout y stderr** al archivo de log mediante `dup2()` para capturar la salida de cualquier subproceso heredado.

9.2 Estructura sugerida

```c
// Estructura principal del daemon
typedef struct {
    int  log_fd;         // fd del archivo de log actual
    char log_path[256];  // ruta del archivo (ej: /var/log/app.log)
    long total_bytes;    // total de bytes escritos en toda la vida del daemon
    long total_records;  // cantidad de registros escritos
    int  rotations;      // cantidad de rotaciones realizadas
} LogDaemon;

// Funciones principales:
void log_init(LogDaemon *daemon, const char *path);
void log_write(LogDaemon *daemon, const char *msg, size_t len);
void log_rotate(LogDaemon *daemon);
void log_flush(LogDaemon *daemon);
void log_quit(LogDaemon *daemon);
void handle_command(LogDaemon *daemon, const char *cmd);

// main(): bucle principal que monitorea stdin con select/poll
int main(void) {
    // 1. Inicializar daemon
    // 2. Redirigir stdout/stderr al log con dup2()
    // 3. Configurar stdin como no bloqueante con O_NONBLOCK
    // 4. Bucle principal:
    //    - select/poll espera datos en stdin
    //    - si hay datos, leer comando y procesar
    //    - verificar tamaño del log y rotar si es necesario
    // 5. Al salir (QUIT o señal): liberar recursos, restaurar fds, cerrar todo
}
```

9.3 Puntos clave de implementación

Rotación atómica

Usá `rename()` para mover el archivo actual a `.log.N` y abrir uno nuevo:

```
void log_rotate(LogDaemon *daemon) {
    // Cerrar el archivo actual si está abierto
    if (daemon->log_fd >= 0) close(daemon->log_fd);

    // Renombrar con incremento en el número
    char new_path[256];
    snprintf(new_path, sizeof new_path, "%s.%d", daemon->log_path, daemon->rotations);
    rename(daemon->log_path, new_path);  // atómico en POSIX

    // Crear nuevo archivo
    daemon->log_fd = open(daemon->log_path, O_WRONLY | O_CREAT | O_APPEND, 0644);
    daemon->rotations++;
}
```

Verificación de tamaño

Usá `lseek(fd, 0, SEEK_END)` para obtener el tamaño actual del archivo sin leerlo.

Comandos asíncronos con poll

Configurá stdin no bloqueante y usá `poll()` para detectar cuándo hay datos disponibles:

```
struct pollfd pfd;
pfd.fd   = STDIN_FILENO;
pfd.events = POLLIN;

while (1) {
    int ret = poll(&pfd, 1, -1);  // espera indefinida

    if (ret > 0 && (pfd.revents & POLLIN)) {
        char cmd[256];
        ssize_t n = read(STDIN_FILENO, cmd, sizeof(cmd) - 1);
        if (n > 0) {
            cmd[n] = '\0';
            // Remover el \n final
            if (n > 0 && cmd[n-1] == '\n') cmd[n-1] = '\0';
            handle_command(daemon, cmd);
        }
    }
}
```

> ***Recuerda:*** *`handle_command()` debe llamar a `log_flush()` para `FLUSH`, imprimir el estado del daemon para `STATUS`, y llamar a `log_quit()` para salir limpiamente.*

Restaurar stdout/stderr al terminar

Cuando el daemon reciba `QUIT`, restaurá los fds originales antes de salir:

```
void log_quit(LogDaemon *daemon) {
    // Restaurar stdout/stderr desde las copias guardadas
    dup2(daemon->saved_stdout, 1);
    dup2(daemon->saved_stderr, 2);

    if (daemon->log_fd >= 0) close(daemon->log_fd);

    // Cerrar las copias de seguridad
    close(daemon->saved_stdout);
    close(daemon->saved_stderr);
}
```

9.4 Ejercicio: implementación completa

Ejercicio 22.15 — JEFE FINAL: Rotador avanzado de logs tolerante a concurrencia y E/S no bloqueante

Conceptos · `open`, `write`, `close`, `lseek`, `flock`, `dup2`, `O_NONBLOCK`, `poll` / `select`, rotación con `rename()`, redirección de streams, manejo asíncrono de comandos

Contexto · Un daemon de logs de producción real necesita escribir a disco sin bloquear el hilo principal, sincronizar escrituras entre múltiples procesos, rotar archivos automáticamente y aceptar comandos interactivos. Este ejercicio implementa todo eso con syscalls crudas.

Consigna · Implementá el demonio de logs descrito en la sección 9.1 siguiendo la estructura sugerida en 9.2 y los puntos clave de 9.3. El programa debe ser robusto, limpio y funcional.

Requisitos funcionales:

- Escritura de logs con syscalls crudas (cero `stdio.h`)
- Sincronización con `flock()` para escrituras concurrentes
- Rotación automática del archivo cuando supera 5 KB
- Comandos asíncronos por stdin: `FLUSH`, `STATUS`, `QUIT`
- Redirección de stdout y stderr al archivo de log
- Restauración limpia de fds al terminar

Entrada · El programa se ejecuta en la consola. Escribe comandos por stdin mientras el daemon corre.

Salida · Archivos de log rotados (`app.log` , `app.log.1` , `app.log.2` ...), estadísticas impresas en stdout/stderr cuando corresponde, y terminación limpia con `QUIT` .

Criterios de aceptación

- ☐ No usa **ninguna función de** `<stdio.h>` para escribir logs. Usa `open()` / `write()` / `close()` .
- ☐ Usa `flock(LOCK_EX)` para proteger cada escritura al archivo de log.
- ☐ Rota el archivo automáticamente cuando supera 5 KB usando `rename()` y recreación.
- ☐ Acepta comandos `FLUSH` , `STATUS` y `QUIT` de forma no bloqueante (stdin + `poll` / `select`).
- ☐ Redirige stdout y stderr al archivo de log con `dup2()` .
- ☐ Restaurá los fds originales (`stdout` / `stderr`) al recibir `QUIT` antes de terminar.
- ☐ Cierra todos los recursos limpiamente al salir (log_fd, saved_stdout, saved_stderr).
- ☐ Maneja correctamente el caso donde se reciben múltiples comandos en una misma línea.

Bonus:

- Soportar múltiples archivos rotados (`app.log.1` , `app.log.2` , ... `app.log.N`).
- Recibir señal SIGUSR1 para forzar rotación sin necesidad de comando.
- Guardar el PID del daemon en un archivo `.pid` para permitir señales externas.