Capítulo 24 — Comunicación entre Procesos (IPC): Pipes, FIFOs, Memoria Compartida y Semáforos

> *Capítulo avanzado, pero **escrito para que se entienda de verdad**. Se apoya en file descriptors (`22-file-descriptors`), procesos paralelos (`23-procesos`), structs (`08`) y memoria dinámica (`09`).*
>
> *Este capítulo te lleva del aislamiento total al trabajo en equipo: dos o más procesos que necesitan hablar entre sí. Pipes anónimos, pipes con nombre (FIFOs), memoria compartida (`shm_open` / `mmap`), semáforos POSIX y protocolos de mensajes. Todo con syscalls crudas.*

Cómo leer este capítulo

Dos procesos aislados no se ven ni se hablan. Cada uno tiene su espacio de memoria propio, sus variables propias y su PID independiente. Pero hay momentos en que necesitás que dos procesos **compartan datos**: un productor genera información y un consumidor la procesa; un servidor responde peticiones de múltiples clientes; un sistema distribuye trabajo entre varios workers.

Linux ofrece **varios mecanismos IPC**, cada uno con sus fortalezas:

- **Pipes anónimos**: caños simples,unidireccionales, solo entre procesos relacionados (padre-hijo).
- **FIFOs (pipes con nombre)**: como pipes pero accesibles por cualquier proceso.
- **Memoria compartida (`shm_open` + `mmap`)**: el mecanismo más rápido — ambos procesos ven la misma memoria RAM.
- **Semáforos POSIX**: para sincronizar el acceso a recursos compartidos y evitar carreras.

Este capítulo va desde lo básico (un pipe, dos procesos) hasta lo complejo (protocolo length-prefixed, productor-consumidor con semáforos). Si entendés cada parte, vas a poder escribir programas distribuidos simples sin frameworks externos.

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	¿Por qué IPC?	Dos procesos aislados no se ven ni se hablan: si necesitan cooperar, Linux les da canales.
2	Pipes anónimos (`pipe()`)	Un caño de una sola dirección: el padre escribe, el hijo lee — los bytes fluyen sin tocar disco.
3	Pipes + fork: patrón clásico	El caño se crea antes del `fork`, así ambos procesos comparten los mismos fds y pueden hablar.
4	Named pipes (FIFOs)	Un caño con nombre en el sistema de archivos: procesos no relacionados se conectan al mismo canal.
5	`read` / `write` en pipes: comunicación pura sin stdio	Los pipes son fds: solo `read` y `write`. Cero `printf`, cero `FILE*`: es comunicación a nivel kernel.
6	Mensajes length-prefixed	Un pipe no sabe dónde termina un mensaje y empieza el otro: necesitás un mini-protocolo.
7	Memoria compartida (`shm_open` + `mmap`)	En vez de copiar bytes por un tubo, ambos procesos mapean el mismo segmento de memoria del kernel.
8	Semáforos POSIX (`sem_init` / `semop`)	Dos procesos escribiendo al mismo lugar es caos: un semáforo deja pasar de a uno.
9	Ejercicio jefe: chat multi-proceso completo	Pipe bidir + length-prefixed + shm para buffer de salida + semáforos + `dup2`.

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

PARTE 1 — ¿Por qué IPC?

Antes de ver los mecanismos, entendamos el problema que resolvemos.

1.1 Aislamiento y cooperación

Linux aísla procesos por defecto. Si el proceso A modifica su variable `x`, el proceso B no lo sabe ni le afecta. Esto es **bueno**: evita que un bug en un programa afecte a otro. Pero también significa:

> ***Dos procesos no pueden compartir datos directamente.*** *Necesitan un canal explícito para comunicarse, ya sea un pipe (bytes crudos), un archivo en disco, memoria compartida o una syscall de sincronización.*

1.2 Los mecanismos IPC de Linux

Mecanismo	Velocidad	Relación necesaria	Dirección	Persistencia
Pipe anónimo (`pipe()`)	Muy rápido	Sí (padre-hijo)	Unidireccional	Solo mientras existan los fds
FIFO (`mkfifo()`)	Rápido	No	Unidireccional	Persiste en disco hasta eliminarlo
Memoria compartida	Más rápido que pipe	No	Bidireccional	Hasta `shm_unlink()`
Semáforo POSIX	N/A (sincronización)	No	N/A (coordina acceso)	Hasta `sem_unlink()`

> ***Elegir el mecanismo correcto:*** *Si solo necesitás enviar datos de un proceso a otro, usá un pipe. Si necesitás comunicación bidireccional o mensajes grandes, memoria compartida es más eficiente. Si necesitás sincronizar acceso a un recurso compartido, semáforos.*

Práctica de la Parte 1

Ejercicio 24.1 — Demostración de aislamiento

Conceptos · `fork`, variables independientes, `printf` después de fork (para ver el efecto)

Contexto · Querés demostrar que dos procesos **no comparten memoria**. Escribí un programa que muestre cómo una variable modificada en el hijo no afecta al padre.

Consigna · Crea un proceso hijo, modifica una variable global en el hijo después de `fork()`, y verificá que el padre mantiene su valor original. Imprimí ambos valores con sus PIDs para diferenciarlos.

Criterios de aceptación

- ☐ Usa `fork()` para crear un hijo.
- ☐ Modifica una variable global en el hijo solo (después del `if/else`).
- ☐ El padre mantiene el valor original de la variable.
- ☐ Imprime los valores con PID para evidencia visual.

PARTE 2 — Pipes anónimos: `pipe()`

Un pipe es un **caño de una sola dirección** creado por el kernel. Tiene dos extremos: un fd para escribir (write-end) y un fd para leer (read-end). Los bytes que escribís en un extremo salen por el otro, sin tocar disco.

2.1 Crear un pipe

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

- `pipefd[0]` → **read end** (extremo de lectura).
- `pipefd[1]` → **write end** (extremo de escritura).

Devuelve `0` si éxito, `-1` en error.

```
int mypipe[2];
if (pipe(mypipe) == -1) {
    perror("pipe falló");
    return 1;
}

// Ahora:
// write(mypipe[1], "hola", 4)  →  read(mypipe[0], buf, 4)   ← lee "hola"
```

> ***Regla de oro:*** *los datos en un pipe son* ***transitorios****. No persisten al disco. Si el proceso lector no está corriendo o no leyó todo antes de que el writer cierre, los datos se pierden (o el write se bloquea si no hay espacio en el buffer del kernel).*

2.2 Escribir y leer con pipes simples

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

int main(void) {
    int mypipe[2];
    pipe(mypipe);

    // Escribir directamente al write-end
    write(mypipe[1], "Hola pipe\n", 10);

    // Leer desde el read-end
    char buf[256];
    ssize_t n = read(mypipe[0], buf, sizeof(buf));
    buf[n] = '\0';  // terminador manual para printf (read no pone \0)
    printf("Leí: %s", buf);  // "Leí: Hola pipe"

    close(mypipe[0]);
    close(mypipe[1]);

    return 0;
}
```

2.3 El orden importa: cerrar el extremo que no se usa

Si intentás leer de un pipe y **ningún proceso cerró el write-end**, `read()` esperará infinitamente porque cree que pueden llegar más datos. Por eso siempre hay que cerrar el extremo que no se usa:

```c
// Correcto: cerrar write-end del padre antes de leer
close(mypipe[1]);  // Padre ya no escribe → hijo puede ver EOF en read()

char buf[256];
ssize_t n = read(mypipe[0], buf, sizeof(buf));
if (n == 0) {
    printf("EOF — no hay más datos\n");
}
```

> ***Regla mnemotécnica:*** *para que* `read()` *devuelva* `0` *(EOF),* ***todos*** *los write-ends del pipe deben estar cerrados. Si incluso un proceso mantiene el write-end abierto,* `read()` *nunca devuelve 0.*

Práctica de la Parte 2

Ejercicio 24.2 — Conversión unidireccional simple

Conceptos · `pipe()`, `write()`, `read()`, EOF como retorno 0 de `read`

Contexto · Escribí un programa que cree un pipe, escriba datos en el write-end y los lea del read-end. Demuestra que los bytes fluyen en la dirección correcta sin tocar disco.

Consigna · Crea un pipe, escribe al menos 5 mensajes diferentes por `write()` al write-end, y después léelos todos con `read()` desde el read-end. Verificá que el orden de lectura coincide con el de escritura (pipes mantienen orden FIFO).

Criterios de aceptación

- [ ] Crea el pipe con `pipe()`.
- [ ] Usa `write(pipe[1], ...)` para escribir múltiples mensajes.
- [ ] Usa `read(pipe[0], ...)` para leer todos los mensajes.
- [ ] Verifica que el orden se mantiene (FIFO).
- [ ] Cierra ambos extremos del pipe al terminar.

Ejercicio 24.3 — Pipe bidireccional con doble pipe

Conceptos · Dos pipes, uno en cada dirección, `read`, `write`

Contexto · Un pipe es unidireccional. Para comunicación bidireccional necesitás **dos pipes**: uno para A→B y otro para B→A.

Consigna · Escribí un programa que cree dos pipes: uno de lectura (pipeA) y otro de escritura (pipeB). El programa escribe datos por pipeA y los lee desde pipeB, y viceversa. Muestra cómo dos procesos pueden comunicarse en ambas direcciones usando dos pipes simples.

Criterios de aceptación

- [ ] Crea dos pipes con `pipe()`.
- [ ] Escribe por un pipe y lee del otro.
- [ ] Verifica que la comunicación funciona en ambos sentidos.
- [ ] Cierra los extremos no usados correctamente (para evitar bloqueos infinitos).

PARTE 3 — Pipes + fork: el patrón clásico

Un pipe creado con `pipe()` solo existe mientras ambos extremos estén abiertos por **el mismo proceso**. La verdadera magia ocurre cuando hacés `fork()`: **ambos procesos heredan los mismos fds**, incluidos los del pipe, y pueden comunicarse a través de ellos.

3.1 El patrón padre-hijo con pipe

```c
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>

int main(void) {
    int mypipe[2];
    pipe(mypipe);

    pid_t pid = fork();

    if (pid == 0) {
        // HIJO: lee del read-end
        close(mypipe[1]);  // Cerramos write-end — ya no escribimos

        char buf[256];
        ssize_t n = read(mypipe[0], buf, sizeof(buf));
        if (n > 0) {
            buf[n] = '\0';
            printf("Hijo leyó: %s", buf);
        }

        close(mypipe[0]);
        return 0;
    }

    // PADRE: escribe al write-end
    close(mypipe[0]);  // Cerramos read-end — ya no leemos

    const char *msg = "Hola desde el padre\n";
    write(mypipe[1], msg, strlen(msg));

    close(mypipe[1]);
    wait(NULL);  // Esperar al hijo para limpiar zombie

    return 0;
}
```

> ***Regla del pipe con fork:***
>
> - *El padre cierra el extremo que no usa (read o write).*
> - *El hijo hace lo mismo.*
> - *Si el padre solo escribe y el hijo solo lee, cada uno cierra el otro extremo.*
> - *Si ambos necesitan leer Y escribir, necesitás **dos pipes** (uno en cada dirección).*

3.2 Cadena de procesos: pipe → fork → pipe → fork → ...

Podés encadenar múltiples procesos usando un pipe entre cada par:

```
// Proceso A | proceso B | proceso C
// pipe1    |   pipe2

pipe(pipe1);
pipe(pipe2);

pid_t pA = fork();
if (pA == 0) {
    // Proceso A: escribe en pipe1[1] y cierra
    write(pipe1[1], "mensaje", 8);
    close(pipe1[1]);
    close(pipe2[0]);  // No usa pipe2
}

pid_t pB = fork();
if (pB == 0) {
    // Proceso B: lee de pipe1[0] y escribe en pipe2[1]
    close(pipe1[1]);   // Cierra write-end de pipe1
    char buf[256];
    read(pipe1[0], buf, sizeof(buf));

    write(pipe2[1], "transformado", 12);

    close(pipe1[0]);
    close(pipe2[1]);
}

// Proceso C (padre o hijo de C): lee de pipe2[0]
close(pipe2[0]);
```

Este patrón se usa en la terminal para crear tuberías entre comandos (`cmd1 | cmd2 | cmd3`): cada pipe conecta la salida de un comando con la entrada del siguiente.

3.3 Cálculo distribuido: padre computa, hijo procesa

Escribí un programa donde el padre calcula números y los envía por un pipe al hijo, que los imprime con formato especial. Usa `fork()` + pipe para separar cálculo de presentación.

```
// Padre calcula números del 1 al 10 y los envía uno a uno.
for (int i = 1; i <= 10; i++) {
    char msg[64];
    int len = snprintf(msg, sizeof(msg), "NUM:%d\n", i);
    write(pipe_fd, msg, len);
}
// Cerrar write-end para que hijo vea EOF

// Hijo lee y muestra cada número formateado:
while ((n = read(pipe_fd, buf, sizeof(buf))) > 0) {
    // Procesar y mostrar
}
```

Práctica de la Parte 3

Ejercicio 24.4 — Calculadora distribuida

Conceptos · `pipe()`, `fork()`, `write()`, `read()`, EOF detection, `snprintf`

Contexto · Escribí un programa donde el **padre calcula** la suma de los números del 1 al N (N recibido por argumento) y envía el resultado por un pipe al **hijo**, que lo imprime formateado como `"Resultado: SUMA = X"`. El hijo espera a recibir el resultado, no lo calcula él mismo.

Consigna · Usa `fork()` + `pipe()`:

1. El padre recibe N por argumento (`argv[1]`).
2. Calcula la suma 1+2+3+...+N.
3. Envía el resultado como texto al hijo a través del pipe con `write()`.
4. El hijo lee del pipe, parsea el número y lo imprime en formato `"Resultado: SUMA = X"`.

Criterios de aceptación

- [ ] Crea el pipe antes de `fork()`.
- [ ] El padre calcula la suma correctamente.
- [ ] El hijo NO realiza ningún cálculo — solo lee del pipe y formatea.
- [ ] Usa `write()` / `read()` para comunicar (sin variables compartidas).
- [ ] Cierra ambos extremos correctamente para evitar bloqueos infinitos.

Ejercicio 24.5 — Cadena de procesadores

Conceptos · Doble pipe, triple fork, cada proceso hace una transformación distinta

Contexto · Escribí un programa que cree **tres procesos en cadena**: Padre → Hijo1 → Hijo2. Cada uno recibe texto del anterior y le aplica una transformación antes de enviarlo al siguiente.

- **Proceso A (padre):** escribe `"hello world"` por pipe1.
- **Proceso B (hijo1):** lee de pipe1, convierte a MAYÚSCULAS (`"HELLO WORLD"`), lo envía por pipe2.
- **Proceso C (hijo2):** lee de pipe2 e imprime el resultado final.

Consigna · Implementá la cadena de tres procesos con dos pipes:

1. Pipe 1 conecta A → B.
2. Pipe 2 conecta B → C.
3. Cada proceso cierra los extremos del pipe que no usa (para evitar bloqueos en `read()`).
4. El proceso final imprime `"Resultado final: HELLO WORLD"` .

Criterios de aceptación

- [ ] Usa dos pipes separados para la cadena.
- [ ] Proceso B convierte a mayúsculas usando solo syscalls (sin `stdio`).
- [ ] Cada proceso cierra correctamente los fds del pipe que no usa.
- [ ] El resultado final es correcto y el programa termina limpio.

PARTE 4 — Named pipes (FIFOs)

Los pipes anónimos solo funcionan entre procesos relacionados (padre-hijo, abuelo-nieto). Los **named pipes** o **FIFOs** son como pipes pero con un nombre en el sistema de archivos: cualquier proceso puede abrirlos y comunicarse a través de ellos.

4.1 Crear un FIFO

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

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

Crea un archivo especial en el sistema de archivos (tipo FIFO). Solo se puede crear **una vez** con el mismo nombre:

```c
// Crear el FIFO si no existe
mkfifo("/tmp/mi_fifo", 0666);
```

> ***El archivo "vive"*** *hasta que lo eliminás con* `unlink()` *. A diferencia del pipe anónimo, que desaparece cuando se cierran los últimos fds, el FIFO persiste en disco.*

4.2 Conectar dos procesos independientes al mismo FIFO

```c
// Proceso escritor (write_fifo.c):
int fd = open("/tmp/mi_fifo", O_WRONLY);  // Se bloquea hasta que alguien abra para leer
write(fd, "Hola desde el otro lado\n", 26);
close(fd);

// Proceso lector (read_fifo.c):
int fd = open("/tmp/mi_fifo", O_RDONLY);  // Se bloquea hasta que alguien abra para
escribir
char buf[256];
ssize_t n = read(fd, buf, sizeof(buf));
printf("Leí: %.*s", (int)n, buf);
close(fd);
```

> ***Bloqueo en*** `open()` ***:*** *cuando abres un FIFO con* `O_RDONLY` *, se bloquea hasta que alguien abre el mismo FIFO con* `O_WRONLY` *(y viceversa). Esto es para garantizar que siempre haya ambos extremos disponibles antes de empezar a comunicarse.*

4.3 Abrir en modo no bloqueante

Si no querés que `open()` se bloquee, usá `O_NONBLOCK`:

```
// Intentar abrir sin bloquear — devuelve -1 si nadie más abrió el otro extremo.
int fd = open("/tmp/mi_fifo", O_WRONLY | O_NONBLOCK);
if (fd < 0) {
    perror("No hay lector aún");
}
```

Práctica de la Parte 4

Ejercicio 24.6 — Emisor y receptor independiente

Conceptos · `mkfifo`, `open` (O_RDONLY / O_WRONLY), `write`, `read`, `unlink`

Contexto · Escribí dos programas independientes: uno que escribe mensajes al FIFO y otro que los lee. No son padre e hijo: corren por separado, en terminales diferentes.

Consigna · Programa 1 (`fifo_emisor.c`): crea el FIFO si no existe, escribe varios mensajes, y al recibir "QUIT" del usuario cierra todo.

Programa 2 (`fifo_receptor.c`): abre el FIFO para leer e imprime cada mensaje que recibe. Muestra el PID de quién escribió (si lo incluye en el mensaje) y al recibir "QUIT" termina limpiamente.

Criterios de aceptación

- [ ] `mkfifo()` se llama solo una vez (o verifica con `access()`).
- [ ] El emisor abre el FIFO con `O_WRONLY` y el receptor con `O_RDONLY`.
- [ ] Los mensajes llegan en orden del emisor al receptor.
- [ ] "QUIT" detiene ambos programas limpiamente.
- [ ] Limpia el FIFO con `unlink()` cuando ya no se necesita.

Ejercicio 24.7 — Chat básico entre dos terminales

Conceptos · Dos FIFOs (uno en cada dirección), bucle interactivo, EOF detection

Contexto · Escribí un chat de consola simple donde dos procesos se comunican en ambas direcciones usando **dos FIFOs**: uno para mensajes de A a B y otro para mensajes de B a A.

Consigna · Cada proceso:

1. Crea los dos FIFOs (`fifo_a_b` y `fifo_b_a`).
2. Abre `fifo_b_a` para lectura (lo que el otro escribe) y `fifo_a_b` para escritura.
3. En un bucle interactivo: lee del otro proceso Y escribe al otro por teclado.
4. Usa `select()` o `poll()` para monitorear stdin Y el pipe simultáneamente (no bloquearse esperando teclado).

Criterios de aceptación

- ☐ Crea dos FIFOs con `mkfifo()` .
- ☐ Cada proceso puede enviar Y recibir mensajes.
- ☐ No se bloquea esperando input: usa `poll()` o `select()` para stdin + pipe simultáneamente.
- ☐ Muestra los mensajes del otro con un indicador como `[OTRO]: mensaje` .

PARTE 5 — `read` / `write` en pipes: comunicación pura sin stdio

Los pipes son file descriptors. Para leer y escribir, usás **exclusivamente** `read()` y `write()` : cero `printf` , cero `FILE*` , cero `fgets` . Es comunicación a nivel kernel: cruda, directa y eficiente.

5.1 El patrón de lectura con pipes

```c
// Leer un pipe en un bucle hasta EOF
char buf[4096];
while (1) {
    ssize_t n = read(pipe_fd, buf, sizeof(buf));

    if (n == 0) {
        // EOF: todos los write-ends cerrados. No hay más datos.
        printf("[EOF recibido]\n");
        break;
    }

    if (n < 0) {
        perror("read falló");
        break;
    }

    // n > 0: leyó n bytes en buf[0..n-1]
    buf[n] = '\0';  // Terminador manual para imprimirlo como string
    printf("[LEIDO %ld bytes]: %s", (long)n, buf);
}
```

5.2 El patrón de escritura con pipes

```c
// Escribir un pipe — puede fallar si nadie está leyendo (SIGPIPE)
const char *msg = "Hola desde el otro lado\n";
ssize_t n = write(pipe_fd, msg, strlen(msg));
if (n < 0) {
    perror("write falló");
} else if ((size_t)n < strlen(msg)) {
    printf("Advertencia: escribí solo %ld de %lu bytes", (long)n, (unsigned
long)strlen(msg));
}
```

SIGPIPE: *si intentás escribir a un pipe cuyo read-end fue cerrado por todos los lectores, el kernel envía* `SIGPIPE` *al proceso. Por defecto, esto termina el proceso. Para evitarlo, capturá la señal:* `signal(SIGPIPE, SIG_IGN);`

5.3 Servidor de fecha: un proceso responde la hora

Escribí un programa servidor que abre un FIFO en modo lectura y, cada vez que alguien le escribe "cualquier cosa", responde con la hora actual escrita al otro FIFO.

```c
// Servidor:
while (1) {
    // Esperar dato entrante del cliente
    char buf[256];
    ssize_t n = read(client_pipe, buf, sizeof(buf));
    if (n == 0) break;  // Cliente se desconectó

    // Generar hora actual con localtime() y strftime()
    time_t now = time(NULL);
    struct tm *tm_info = localtime(&now);
    char time_buf[64];
    strftime(time_buf, sizeof(time_buf), "%Y-%m-%d %H:%M:%S", tm_info);

    // Escribir respuesta al cliente
    snprintf(response, sizeof(response), "Hora: %s\n", time_buf);
    write(client_pipe, response, strlen(response));
}
```

Práctica de la Parte 5

Ejercicio 24.8 — Línea de comando remota

Conceptos · Pipe bidireccional (dos pipes), `read`, `write`, ejecución de comandos del cliente

Contexto · Escribí un programa servidor que ejecuta comandos recibidos por un pipe y devuelve la salida al cliente. El cliente envía comandos como `"ls -la"` o `"date"` y recibe la respuesta exacta como si los hubiera ejecutado localmente.

Consigna · Implementá cliente + servidor con dos FIFOs:

1. **Servidor:** abre fifo_client_server para lectura, ejecuta el comando recibido (con `fork/exec`), captura la salida del proceso hijo y la envía por fifo_server_client.
2. **Cliente:** escribe un comando en fifo_client_server, lee la respuesta de fifo_server_client, y la imprime en pantalla.

> ***Nota:*** *usar* `dup2()` *para redirigir stdout del proceso hijo al write-end de un pipe (patrón clásico de captura de salida de comandos).*

Criterios de aceptación

- [ ] Usa dos FIFOs bidireccionales correctamente.
- [ ] El servidor ejecuta el comando recibido con `fork/exec`.
- [ ] La salida del comando llega intacta al cliente.
- [ ] El cliente imprime la respuesta completa sin corromper datos.

Ejercicio 24.9 — Servidor de fecha (implementado)

Conceptos · `read`, `write`, `time.h`, `strftime`, `mkfifo`

Contexto · Un servidor simple que, cada vez que un cliente le escribe "cuál es la hora?", responde con la hora actual escrita al pipe de respuesta del cliente. Usa solo syscalls crudas (cero `stdio`).

Consigna · Implementá el servidor y el cliente:

- **Servidor:** abre FIFO para lectura, espera mensajes, genera hora con `time()` / `localtime()` / `strftime()`, la envía al pipe de respuesta del cliente.
- **Cliente:** escribe "hora" por FIFO de petición, lee respuesta desde su propio FIFO.

> *Usa* `pipe()` *en vez de FIFO para simplificar (solo procesos locales). El servidor recibe N peticiones antes de terminar.*

Criterios de aceptación

- [ ] Servidor y cliente se comunican exclusivamente con `read` / `write`.
- [ ] La hora se genera correctamente con la API de `<time.h>`.
- [ ] Formato de salida: `"Hora: YYYY-MM-DD HH:MM:SS\n"`.
- [ ] El servidor maneja N peticiones y el cliente recibe cada respuesta completa.

PARTE 6 — Mensajes length-prefixed: protocolos en pipes

Un pipe es un **stream continuo de bytes**. No tiene estructura de mensajes: si escribís `"hola\nmundo\n"` y luego leés, podés recibir `"hol"` en una lectura y `"a\nmundo\n"` en la siguiente. Para enviar mensajes delimitados necesitás un protocolo. El más simple: **length-prefixed**.

6.1 El problema: ¿dónde termina un mensaje y empieza el otro?

```
// Escritor envía dos mensajes:
write(pipe_fd, "Hola", 4);    // Mensaje 1: 4 bytes
write(pipe_fd, "Mundo", 5);  // Mensaje 2: 5 bytes

// Lector lee de una vez:
read(pipe_fd, buf, 9);  // ¿Recibe "HolaMundo" (9 bytes)? Sí.
// Pero... ¿cómo sabe dónde termina el mensaje 1 y empieza el 2?
```

Sin protocolo, el lector no puede distinguir mensajes individuales. Solución: **cada mensaje comienza con un entero de 4 bytes que indica su longitud**.

6.2 Protocolo length-prefixed

```c
// Escritor envía un mensaje con prefijo de longitud:
void send_message(int fd, const char *msg) {
    uint32_t len = htonl((uint32_t)strlen(msg));  // Longitud en network byte order
    write(fd, &len, sizeof(len));                  // Primero: los 4 bytes de longitud
    write(fd, msg, strlen(msg));                   // Después: el contenido
}

// Lector recibe un mensaje con prefijo de longitud:
ssize_t recv_message(int fd, char *buf, size_t buf_size) {
    uint32_t len_net;

    // Leer los 4 bytes de longitud
    ssize_t n = read(fd, &len_net, sizeof(len_net));
    if (n != sizeof(len_net)) return -1;  // Error o EOF

    uint32_t len = ntohl(len_net);  // Convertir a host byte order

    if (len >= buf_size) {
        return -1;  // Mensaje demasiado largo para el buffer
    }

    // Leer exactamente `len` bytes de contenido
    ssize_t total = 0;
    while (total < (ssize_t)len) {
        n = read(fd, buf + total, len - total);
        if (n <= 0) return -1;  // Error o EOF inesperado
        total += n;
    }

    buf[total] = '\0';  // Terminar como string
    return total;         // Longitud del mensaje leído
}
```

`htonl()` / `ntohl()`: *convierten entre network byte order (big-endian) y host byte order. Es importante para compatibilidad si los procesos corren en arquitecturas con diferente endianness. En una máquina x86_64 local (little-endian), no estrictamente necesario, pero es buena práctica.*

6.3 Chat con mensajes estructurados

Usá el protocolo length-prefixed para enviar mensajes de tipo `"USUARIO: mensaje"` entre procesos:

```
// Lector en un bucle infinito:
while (1) {
    char msg[4096];
    ssize_t len = recv_message(pipe_fd, msg, sizeof(msg));
    if (len == 0) {
        printf("[EOF]\n");
        break;
    }
    if (len < 0) {
        perror("recv_message falló");
        break;
    }
    printf("[RECIBIDO]: %s", msg);  // msg ya tiene el \0
}
```

Práctica de la Parte 6

Ejercicio 24.10 — Protocolo length-prefixed básico

Conceptos · `htonl`, `ntohl`, `write`, `read`, protocolo de longitud prefijada

Contexto · Escribí un programa que demuestre el protocolo length-prefixed: envía varios mensajes de diferentes longitudes por un pipe y los lee en el otro extremo, verificando que cada mensaje se recibe intacto y separado.

Consigna · El programa:

1. Crea un pipe y fork().
2. El padre escribe 3 mensajes con longitudes diferentes (ej: `"h"`, `"hola mundo"`, `"abcde"`) usando el protocolo length-prefixed (4 bytes de longitud + contenido).
3. El hijo lee los 3 mensajes uno por uno, imprimiendo cada uno con su longitud.

Criterios de aceptación

- [ ] Usa `htonl()` para codificar la longitud del mensaje.
- [ ] Escribe primero 4 bytes de longitud, luego el contenido.
- [ ] El hijo lee primero los 4 bytes de longitud, luego exactamente esa cantidad de bytes.
- [ ] Verifica que cada mensaje se recibe completo y sin mezclar con otros.

Ejercicio 24.11 — Chat con mensajes estructurados

Conceptos · `fork`, pipe bidireccional, protocolo length-prefixed, lectura interactiva por stdin

Contexto · Escribí un programa de chat entre padre e hijo: el padre escribe desde teclado y el hijo imprime los mensajes recibidos en pantalla (y viceversa). Usa el protocolo length-prefixed para delimitar mensajes.

Consigna · Implementá un bucle interactivo:

1. El padre lee una línea de stdin (`fgets`), la envía al hijo con `send_message()` .
2. El hijo recibe con `recv_message()` e imprime `[HIJO] mensaje` .
3. Invertir el flujo: hijo responde, padre imprime `[PADRE] respuesta` .
4. Terminar con "exit" en cualquiera de los dos lados.

Criterios de aceptación

- [ ] Usa protocolo length-prefixed correctamente.
- [ ] Ambos extremos pueden enviar y recibir.
- [ ] Los mensajes llegan intactos (sin mezclas, sin pérdida).
- [ ] Termina limpiamente al escribir "exit".

PARTE 7 — Memoria compartida (`shm_open` + `mmap`)

Con pipes, cada mensaje se copia del espacio del escritor al del kernel y luego del kernel al del lector. Con **memoria compartida**, ambos procesos ven la misma región de memoria física: **cero copias**.

7.1 Crear una sección de memoria compartida

```c
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

// Crear o abrir una región compartida en el sistema de archivos /dev/shm/
int fd = shm_open("/mi_shm", O_CREAT | O_RDWR, 0666);
if (fd < 0) { perror("shm_open"); return 1; }

// Asignarle tamaño (en bytes) — esto la "reserva" en RAM del kernel.
ftruncate(fd, sizeof(int));  // Reserva 4 bytes para un solo entero compartido

// Mapear esa región en el espacio de memoria del proceso actual.
int *shared_counter = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
                           MAP_SHARED, fd, 0);
if (shared_counter == MAP_FAILED) { perror("mmap"); return 1; }

// Cerrar el fd — ya no lo necesitamos para acceder a la memoria.
close(fd);
```

7.2 Leer y escribir en memoria compartida

Una vez mapeada, la región se usa como cualquier puntero:

```c
// Proceso A escribe un valor:
*shared_counter = 42;

// Proceso B (que también hizo shm_open + mmap) lo lee:
printf("Valor en memoria compartida: %d\n", *shared_counter);  // Imprime 42
```

> ***La misma región física*** *aparece en el espacio de direcciones virtual de ambos procesos, pero en direcciones diferentes.* `shared_counter` *del proceso A apunta a una dirección distinta que la de B, pero ambas apuntan a la misma RAM.*

7.3 Limpiar y desmapear

```c
// Desmapear el segmento del espacio de este proceso.
munmap(shared_counter, sizeof(int));

// Eliminar la sección compartida del sistema (debe hacerlo uno de los procesos).
shm_unlink("/mi_shm");
```

> ***`shm_unlink()` no destruye la memoria inmediatamente.*** *Solo elimina el nombre del sistema. La región se libera cuando* ***el último proceso que la tiene mapeada*** *llama a* `munmap()` *. Si un proceso la desmpea pero otro aún la usa, los datos permanecen accesibles.*

Práctica de la Parte 7

Ejercicio 24.12 — Contador compartido

Conceptos · `shm_open` , `ftruncate` , `mmap` , `munmap` , `shm_unlink`

Contexto · Escribí un programa donde dos procesos comparten un entero con memoria compartida y cada uno lo incrementa 100 veces. Sin sincronización, el resultado final será menor a 200 debido a condiciones de carrera.

Consigna · El programa:

1. Crea una región compartida con `shm_open` / `ftruncate` para almacenar un `int` .
2. Fork() para crear un hijo.
3. Ambos procesos incrementan el contador 100 veces cada uno (usando un bucle simple).
4. Al final, imprimen el valor final del contador.

> *Sin sincronización (semáforos), verás que el resultado es menor a 200 porque ambos procesos leen-modifican-escriben el mismo entero simultáneamente. Esto demuestra* ***por qué se necesitan semáforos*** *(Parte 8).*

Criterios de aceptación

- [ ] Crea la región compartida con `shm_open` + `ftruncate`.
- [ ] Ambos procesos hacen `mmap` para acceder al entero.
- [ ] Cada proceso incrementa el contador exactamente 100 veces.
- [ ] Imprime el valor final (que será < 200 sin sincronización).

Ejercicio 24.13 — Buffer productor-consumidor básico

Conceptos · `shm_open`, `mmap`, estructura con datos, límites de buffer

Contexto · Escribí un programa productor-consumidor donde el productor escribe cadenas en un buffer compartido (estructura con `char data[1024]` y un campo `int used`) y el consumidor lee desde ahí.

Consigna · El programa:

1. Define una estructura `SharedBuffer` con un campo `char data[1024]` y otro `int used`.
2. Crea la región compartida con `shm_open` / `ftruncate` del tamaño de la estructura.
3. Fork() para crear hijo (consumidor).
4. El padre (productor) escribe cadenas en `data` mientras el hijo (consumidor) las lee.

Criterios de aceptación

- [ ] Define correctamente la estructura `SharedBuffer`.
- [ ] Usa `shm_open` / `ftruncate` con tamaño `sizeof(SharedBuffer)`.
- [ ] Ambos procesos acceden a la misma instancia mapeada.
- [ ] El productor escribe y el consumidor lee del buffer compartido.

PARTE 8 — Semáforos POSIX: sincronizar acceso a recursos compartidos

Dos procesos escribiendo al mismo lugar sin coordinación = datos corrompidos. Los semáforos son el mecanismo de Linux para **garantizar que solo un proceso acceda a un recurso a la vez**.

8.1 El problema: condiciones de carrera en escritura

```
// Proceso A y B hacen esto al mismo tiempo:
int x = *shared_counter;  // Paso 1: leer el valor actual
x++;                      // Paso 2: incrementarlo
*shared_counter = x;      // Paso 3: escribirlo de vuelta

// Si ambos ejecutan los tres pasos en paralelo, uno de los incrementos se pierde.
```

Este es un **read-modify-write race condition**: dos procesos leen el mismo valor, lo modifican por separado, y escriben — pero solo uno de los cambios sobrevive.

8.2 Semáforos binarios: mutex (mutual exclusion)

```
#include <semaphore.h>

// Crear un semáforo con valor inicial = 1 (lock disponible)
sem_t *mutex = sem_open("/mi_mutex", O_CREAT, 0666, 1);

// Adquirir el lock: si está disponible, lo toma. Si no, espera.
sem_wait(mutex);

// === SECCIÓN CRÍTICA: solo un proceso a la vez llega aquí ===
*shared_counter++;
// ============================================================

// Liberar el lock: otro proceso puede entrar ahora.
sem_post(mutex);
```

> *`sem_wait()` es como `wait()` en sistemas operativos clásicos: decrementa el semáforo y bloquea si llega a `-1`. `sem_post()` es como `signal()`: incrementa y despierta un proceso dormido.*

8.3 Semáforos con nombre vs anónimos

Tipo	Creación	Alcance	Ejemplo
Con nombre (`sem_open`)	`sem_open()`	Entre procesos no relacionados	`/mi_sem`
Anónimo (`sem_init`)	`sem_init()`	Solo entre hijos del mismo `fork()` (o misma memoria compartida)	En estructura `sem_t`

Para semáforos anónimos (usados en el mismo proceso padre e hijos):

```
sem_t sem;
sem_init(&sem, 0, 1);   // 0 = no compartido entre procesos no relacionados, 1 = valor inicial

// ... fork() ...

sem_wait(&sem);    // Adquirir
/* sección crítica */
sem_post(&sem);    // Liberar

sem_destroy(&sem);  // Al finalizar (no hay sem_unlink para anónimos)
```

8.4 Productor-consumidor con dos semáforos

El patrón clásico de productor-consumidor usa **dos semáforos**:

- `slots` : cuenta cuántos slots vacíos quedan en el buffer.
- `items` : cuenta cuántos items llenos hay en el buffer.

```
sem_t slots;  // Inicializado a N (tamaño del buffer)
sem_t items;  // Inicializado a 0 (no hay items al inicio)
sem_t mutex;  // Mutex para proteger acceso al buffer

// Productor:
sem_wait(&slots);    // Esperar slot vacío disponible
sem_wait(&mutex);    // Lock el buffer
/* escribir en buffer[write_pos++] */
sem_post(&mutex);    // Unlock buffer
sem_post(&items);    // Un item más disponible

// Consumidor:
```

```
sem_wait(&items);    // Esperar item disponible
sem_wait(&mutex);    // Lock el buffer
/* leer de buffer[read_pos++] */
sem_post(&mutex);    // Unlock buffer
sem_post(&slots);    // Un slot libre disponible
```

Práctica de la Parte 8

Ejercicio 24.14 — Productor-consumidor con mutex

Conceptos · `sem_init`, `sem_wait`, `sem_post`, sección crítica, read-modify-write race

Contexto · Repetí el ejercicio del contador compartido (24.12) pero ahora **agregando un semáforo mutex** para proteger el acceso al contador. Verificá que el resultado final es exactamente 200 (era menor sin sincronización).

Consigna · Modificá el programa de 24.12 para que:

1. Cada proceso adquiera el mutex antes de leer-modificar-escribir el contador compartido.
2. Libere el mutex después de escribir el valor actualizado.
3. Imprima el resultado final (debe ser exactamente 200).

Criterios de aceptación

- [ ] Usa `sem_init()` para crear un semáforo con valor inicial 1.
- [ ] Cada incremento del contador está protegido por `sem_wait` / `sem_post`.
- [ ] El resultado final es exactamente 200 (sin carrera).
- [ ] Limpia el semáforo con `sem_destroy()`.

Ejercicio 24.15 — Cola de trabajo múltiple

Conceptos · Dos semáforos (`slots` / `items`), productor-consumidor, buffer circular

Contexto · Escribí un programa con **un productor y N consumidores** (N=3) que comparten un buffer circular de tamaño 5. El productor genera números del 1 al 100; los consumidores los leen y los imprimen.

Consigna · Implementá el patrón productor-consumidor con:

- Buffer circular de 5 enteros en memoria compartida.
- Tres semáforos: `slots` (inicializado a 5), `items` (inicializado a 0), `mutex` (inicializado a 1).
- Un proceso productor y 3 procesos consumidores.
- Cada consumidor imprime `"Consumidor X tomó Y"` para cada valor que procesa.

Criterios de aceptación

- ☐ Usa tres semáforos correctamente (`slots` , `items` , `mutex`).
- ☐ El buffer circular tiene tamaño fijo de 5 elementos.
- ☐ Los 3 consumidores compiten por items sin duplicar trabajo.
- ☐ Cada valor es procesado exactamente una vez.
- ☐ El programa termina cuando el productor envía un marcador de fin (ej: -1).

PARTE 9 — Ejercicio jefe: chat multi-proceso completo

Llegó el momento de juntar **todo lo aprendido** en un solo programa: pipe bidireccional para comunicación, protocolo length-prefixed para mensajes delimitados, memoria compartida para buffer de salida, y semáforos para evitar que dos procesos impriman en pantalla simultáneamente.

9.1 El desafío completo

Escribí un **sistema de chat multi-proceso** que:

1. Cree un padre (emisor) y un hijo (receptor), comunicados por un **pipe bidireccional** (dos pipes anónimos, uno en cada dirección).
2. Use el protocolo **length-prefixed** para enviar mensajes delimitados sin ambigüedad.
3. Comparta un **buffer de salida** con memoria compartida (`shm_open` + `mmap`) donde cada proceso escribe su parte antes de imprimirla en pantalla.
4. Use un **semáforo POSIX** para garantizar que solo un proceso imprima a la vez (no se mezclen mensajes).
5. Redirija stdout y stderr al pipe del hijo mediante `dup2()` si se desea capturar toda la salida del hijo.

9.2 Estructura sugerida

```c
#include <semaphore.h>

// Estructura del buffer compartido para salida en pantalla.
typedef struct {
    char data[1024];  // Buffer de texto que puede ser escrito por cualquiera de los
procesos.
    int  len;         // Longitud actual del texto en `data`.
} OutBuffer;

int main(void) {
    /* 1. Crear pipes bidireccionales (pipe_a -> pipe_b y pipe_b -> pipe_a) */

    /* 2. Crear semáforo mutex para proteger acceso a pantalla/buffer compartido */
    sem_t *screen_mutex = sem_open("/chat_screen_mutex", O_CREAT, 0666, 1);

    /* 3. Crear región de memoria compartida para buffer de salida */
    int shm_fd = shm_open("/chat_outbuffer", O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, sizeof(OutBuffer));
    OutBuffer *outbuf = mmap(NULL, sizeof(OutBuffer), PROT_READ|PROT_WRITE,
                             MAP_SHARED, shm_fd, 0);

    /* 4. Fork() para crear el otro proceso (padre/receptor) */
    pid_t pid = fork();

    if (pid == 0) {
        // HIJO: bucle interactivo — lee del pipe_A y envía por pipe_B
        handle_child(pipe_read, pipe_write, outbuf, screen_mutex);
        _exit(0);
    }

    // PADRE: bucle interactivo — escribe al pipe_B y lee del pipe_A
    handle_parent(pipe_write, pipe_read, outbuf, screen_mutex);

    /* 5. Limpieza */
    sem_close(screen_mutex);
    sem_unlink("/chat_screen_mutex");
    munmap(outbuf, sizeof(OutBuffer));
    shm_unlink("/chat_outbuffer");
    close(shm_fd);

    return 0;
}

// Cada extremo del chat tiene un bucle similar:
void handle_parent(int read_pipe, int write_pipe, OutBuffer *outbuf, sem_t *mutex) {
    char line[256];
```

```
    while (fgets(line, sizeof(line), stdin)) {
        // Remover el \n final.

        // Si es "QUIT", enviar mensaje de salida y romper el bucle.

        // Enviar mensaje al hijo con protocolo length-prefixed.
        send_message(write_pipe, line);

        // Recibir respuesta del hijo (si la hay).
        char response[4096];
        ssize_t len = recv_message(read_pipe, response, sizeof(response));
        if (len > 0) {
            // Escribir en buffer compartido y tomar pantalla.
            sem_wait(mutex);
            printf("%s", response);
            fflush(stdout);
            sem_post(mutex);
        }
    }
}

void handle_child(int read_pipe, int write_pipe, OutBuffer *outbuf, sem_t *mutex) {
    // Leer del pipe, responder al padre.
}
```

9.3 Puntos clave de implementación

Protocolo length-prefixed en pipes bidireccionales

Cada proceso tiene dos fd: uno para leer (pipe_A[0] o pipe_B[0]) y otro para escribir (pipe_A[1] o pipe_B[1]). El protocolo length-prefixed se usa tanto en un sentido como en el otro.

Buffer de salida compartido

Ambos procesos pueden escribir en `outbuf` al mismo tiempo. Para evitar que las escrituras se mezclen:

- Cada proceso escribe su mensaje completo en la zona local del buffer (o directamente a `printf` con el mutex).
- El mutex protege tanto el buffer como la escritura a stdout.

Semáforo para pantalla

Usá `sem_wait(mutex)` antes de imprimir y `sem_post(mutex)` después:

```
sem_wait(screen_mutex);
printf("[PADRE]: %s\n", mensaje);  // Imprime en pantalla.
fflush(stdout);                    // Asegura que se escriba inmediatamente (sin
buffering).
sem_post(screen_mutex);
```

Redirigir stdout del hijo con `dup2()`

Si querés capturar toda la salida del hijo hacia el pipe:

```
// En el hijo, antes de empezar a leer/escribir:
int log_fd = open("child_stdout.log", O_WRONLY | O_CREAT | O_TRUNC, 0644);
dup2(log_fd, STDOUT_FILENO);  // stdout ahora apunta al archivo.
close(log_fd);

// Ahora todo lo que el hijo imprima por printf/stderr va al archivo.
```

9.4 Ejercicio: implementación completa

Ejercicio 24.16 — JEFE FINAL: Chat multi-proceso completo

Conceptos · `pipe()`, `fork()`, protocolo length-prefixed, `shm_open`, `mmap`, semáforos POSIX (`sem_open` / `sem_wait` / `sem_post`), `dup2` para redirección, manejo asíncrono de stdin

Contexto · Un sistema de chat entre dos procesos que usa pipes bidireccionales para comunicación, protocolo length-prefixed para mensajes delimitados, memoria compartida para buffer de salida compartido, y semáforos POSIX para sincronizar la impresión en pantalla. Es un programa real de IPC que combina todos los conceptos del capítulo.

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

Requisitos funcionales:

- Comunicación bidireccional con dos pipes anónimos (uno en cada dirección).
- Protocolo length-prefixed para delimitar mensajes enviados por los pipes.
- Región de memoria compartida (`shm_open` + `mmap`) para buffer de salida compartido.
- Semáforo POSIX para garantizar exclusión mutua al imprimir en pantalla.
- Soporte para mensaje especial `"QUIT"` que cierra limpiamente ambos procesos.
- Limpieza completa al terminar: desmpeado, `munmap` , `sem_close` , `shm_unlink` .

Entrada · El programa se ejecuta en la consola. Cada proceso escribe mensajes por stdin.

Salida · Mensajes del otro lado aparecen en pantalla con indicador `[OTRO]: mensaje` .

Criterios de aceptación

- ☐ Usa dos pipes anónimos para comunicación bidireccional entre padre e hijo.
- ☐ Implementa protocolo length-prefixed correctamente (`send_message` / `recv_message`).
- ☐ Crea memoria compartida con `shm_open` + `mmap` .
- ☐ Usa semáforo POSIX (`sem_open` , `sem_wait` , `sem_post`) para proteger impresión en pantalla.
- ☐ Soporta mensaje `"QUIT"` que termina limpiamente ambos procesos.
- ☐ Limpia todos los recursos al salir: `munmap` , `sem_close` , `shm_unlink` .
- ☐ No hay mensajes mezclados en pantalla (el mutex protege la sección crítica correctamente).
- ☐ Maneja EOF en pipes limpiamente (sin bucles infinitos ni crashes).

Bonus:

- Soportar más de dos procesos (padre + N hijos) con un único pipe bidireccional central.
- Redirigir stdout/stderr del hijo al archivo `child.log` con `dup2()` .
- Guardar historial de conversación en memoria compartida y restaurarlo al reiniciar.