Capítulo 23 — Procesos en Paralelo: Fork, Exec y la Vida de un Proceso

> *Capítulo avanzado, pero **escrito para que se entienda de verdad**. Se apoya en file descriptors (`22-file-descriptors`), memoria dinámica (`09`) y structs (`08`). También asume que ya escribiste programas con syscalls crudas.*
>
> *Este capítulo te saca del mundo monocoral y te mete dentro del kernel: cada programa que corre tiene su propio espacio de memoria, sus propias variables y su propio identificador. `fork` , `exec` , `wait` , zombies, pipes, señales... acá aprendés a controlar la vida y muerte de otros procesos desde C.*

Cómo leer este capítulo

Llegó el momento en que tu programa deja de ser **el único que corre**. Linux ejecuta miles de procesos al mismo tiempo: servidores, shells, editors, browsers. Cada uno con su espacio de memoria aislado, sus archivos abiertos y su propio identificador numérico.

Este capítulo va desde lo básico (`getpid` , `fork()`) hasta aplicaciones avanzadas (mini-shell paralelo, copiador recursivo concurrente). Si entendés cada parte, vas a poder escribir programas que **crean otros programas**, **coordinan entre ellos** y **monitorean su progreso**.

Está dividido en **ocho 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 proceso?	Cada programa tiene su propio espacio de memoria y su ID: el kernel los aísla unos de otros.
2	`fork()`: clonarte a ti mismo	Crear un hijo es detenerte en el tiempo y crear un gemelo idéntico que sigue por su lado.
3	`exec*`: reemplazarte completamente	Lanzar otro programa es cambiar la cara del proceso sin perder el PID ni los fds abiertos.
4	`wait`, `waitpid`: esperar a que terminen	El hijo zombie muere solo cuando el padre lo recoge: `wait` es la tarea pendiente que nadie salta.
5	Estado de salida: exit code y señales	Tu proceso puede "morir" bien, con código de error, o ser asesinado por una señal del sistema.
6	Variables de entorno y `argv`	Cada proceso vive con su conjunto de variables; podés verlas, filtrarlas y pasarlas al lanzar otro.
7	UID, GID, umask: tu identidad en Linux	No solo tenés un PID: tenés usuario real/efectivo, grupo, y el umask define qué permisos heredás.
8	Ejercicio jefe: copiador recursivo paralelo	Spawnea un hijo por archivo + `waitpid(WNOHANG)` + estadísticas en tiempo real de copia.

> ***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 — ¿Qué es un proceso?

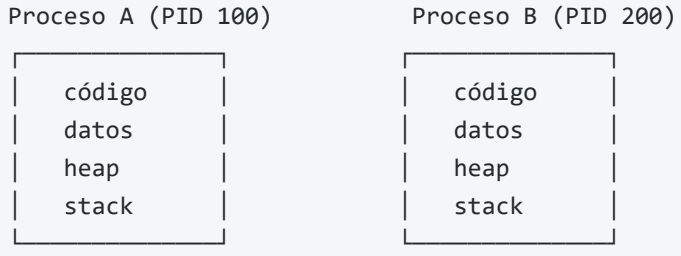
Antes de hablar de cómo crear procesos, entendamos qué es uno y por qué importa.

1.1 Un espacio de memoria aislado

Cada programa que ejecutás en Linux tiene su propio **espacio de direcciones virtuales**. Eso significa:

- Tus variables **no existen** en el proceso vecino.
- Tu heap, tu stack, tu código binario —todo— está en una zona de memoria que **solo vos** podés tocar.
- Si hacés `*pointer = 42` y ese puntero pertenece a otro proceso, el kernel te mata con `SIGSEGV`.

Esta aislamiento es lo que hace posible que Linux sea un sistema multi-tarea seguro: ningún programa puede corromper la memoria de otro.



1.2 El PID: tu número de identidad

Cada proceso tiene un **PID (Process ID)** que lo identifica de forma única en el sistema:

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

int main(void) {
    printf("Mi PID: %d\n", getpid());        // Mi propio PID
    printf("Mi padre (PPID): %d\n", getppid()); // PID de mi padre

    return 0;
}
```

- `getpid()` : devuelve el PID del proceso que lo llama.
- `getppid()` : devuelve el PID del proceso padre que creó este proceso.
- `getuid()` : devuelve el UID real del usuario que ejecuta el proceso.
- `geteuid()` : devuelve el UID efectivo (puede ser distinto por setuid).
- `getgid()` / `getegid()` : mismas funciones para grupos.

> ***El PID 1.*** *Cuando Linux arranca, el primer proceso que inicia se llama* `init` *(o* `systemd` *). Tiene PID 1 y es el ancestro de todos los demás. Nadie puede tener padre con PID 0 excepto procesos del kernel mismo. Cuando un proceso muere sin ser reclamado por su padre, se convierte en* ***zombie*** *y queda como PID huérfano hasta que* `init` *lo recoja.*

1.3 El árbol de procesos

Los procesos forman un **árbol jerárquico**. Cada proceso (excepto PID 1) tiene exactamente un padre:

```
PID 1 (systemd/init)
├── PID 42 (bash shell)
│   ├── PID 101 (gcc -o prog prog.c)
│   └── PID 102 (./prog)
├── PID 55 (sshd: server)
│   └── PID 200 (user@pts/0)
│       └── PID 300 (vim archivo.c)
```

Cuando `PID 101` muere, el shell (`PID 42`) lo reclama con `wait()` . Cuando el shell sale, `init` reclama sus hijos huérfanos y los limpia.

Práctica de la Parte 1

Ejercicio 23.1 — ¿Quién soy?

Conceptos · `getpid` , `getppid` , `getuid` , `geteuid` , `getgid`

Contexto · Querés entender tu identidad como proceso. Linux te da varias identidades: PID, PPID, UID real, UID efectivo, GID y más. Cada una tiene un significado distinto.

Consigna · Escribí un programa que imprima toda la información de identidad del proceso actual: PID, PPID, UID real, UID efectivo, GID real, GID efectivo, y nombre de usuario/grupo si es posible.

Entrada · Ninguna. **Salida** · Lista completa de identidades del proceso.

Escenario de prueba

```
$ gcc 23_01.c -o whoami && ./whoami
Proceso: ¿Quién soy?
====================
PID:            4512
PPID:           4510 (bash)
UID real:       1000
UID efectivo:   1000
GID real:       1000
GID efectivo:   1000
```

Criterios de aceptación

- [ ] Usa `getpid()` y `getppid()` correctamente.
- [ ] Usa `getuid()`, `geteuid()`, `getgid()`, `getegid()`.
- [ ] Muestra la información en un formato legible y ordenado.
- [ ] Incluye al menos 6 campos de identidad.

PARTE 2 — `fork()`: clonarte a ti mismo

`fork()` es la syscall más poderosa de Linux. Crea una **copia exacta** de tu proceso: mismas variables, mismos archivos abiertos, mismo código. Solo que ahora hay dos procesos corriendo simultáneamente.

2.1 La magia en una línea

```c
#include <unistd.h>

pid_t fork(void);
```

- Devuelve **el PID del hijo** al proceso padre.
- Devuelve `0` al proceso hijo.
- Devuelve `-1` si no se pudo crear el proceso (sin memoria, sin recursos).

La misma línea de código se ejecuta **dos veces**: una en el padre, otra en el hijo. Cada uno con su propio espacio de memoria:

```c
int valor = 42;
pid_t pid = fork();

if (pid == -1) {
    perror("fork falló");
    return 1;
} else if (pid > 0) {
    // Esto SOLO lo ejecuta el PADRE. pid es el PID del hijo.
    printf("Soy el padre (PID %d). Mi hijo es el PID %d.\n", getpid(), pid);
    valor = 100;  // El hijo NO VE este cambio: tiene su propia copia de `valor`
} else {
    // Esto SOLO lo ejecuta el HIJO. pid es 0.
    printf("Soy el hijo (PID %d). Mi padre es PID %d.\n", getpid(), getppid());
    valor = 999;  // El padre NO VE este cambio
}

// Aquí ambos llegan, pero por caminos diferentes:
printf("Valor final: %d\n", valor);  // Padre imprime 100, hijo imprime 999
```

> ***`fork` es copy-on-write.*** *El kernel no copia físicamente la memoria del proceso al hacer* `fork()` *. Marca las páginas como "solo lectura" y solo las copia cuando uno de los dos procesos intenta escribir. Esto hace que* `fork()` *sea sorprendentemente rápido incluso para procesos con gigabytes de memoria.*

2.2 El patrón fork-then-exec: crear otro programa

El uso más común de `fork()` es: clonarte a ti mismo y luego **reemplazar** la copia con otro programa usando `exec*`. Así el hijo deja de ser tu proceso y se convierte en lo que le digas:

```c
pid_t pid = fork();

if (pid == 0) {
    // Hijo: reemplazate por "ls -la"
    execl("/bin/ls", "ls", "-la", NULL);

    // Si exec llega acá, falló. Manejá el error.
    perror("execl falló");
    _exit(1);  // Salir SIN fflush ni exit(): solo syscalls
}

// Padre: seguís corriendo. Esperás al hijo con wait().
```

> ***`_exit()` vs `exit()`.*** *En el hijo, después de un `exec` fallido, usá `_exit(1)` (syscall directa) no `exit(1)`. `exit()` hace cleanup de stdio (fflush, handlers) que puede corromper si el padre y el hijo comparten buffers. `_exit()` es más limpio: cierra fds, libera memoria, termina.*

2.3 Variables independientes: el gemelo que se diferencia

Después de `fork()`, padre e hijo comparten **copia inicial** de todo. Pero a partir de ahí son completamente independientes:

```c
int counter = 0;
fork();
counter++;  // Padre incrementa su counter a 1, hijo al 1 — pero son variables distintas.
printf("%d\n", counter);  // Imprime "1" dos veces, no "2"
```

Este comportamiento explica por qué algunos programas imprimen cosas dos veces después de hacer `fork()`: porque **dos procesos ejecutan el mismo código**. Si tenés un `printf` después de `fork()` y no lo separás con `if (pid == 0) ... else if (pid > 0) ...`, ambos procesos lo ejecutarán.

Práctica de la Parte 2

Ejercicio 23.2 — Gemelos digitales

Conceptos · `fork`, `getpid`, `getppid`, variables independientes, `if/else` post-fork

Contexto · Querés crear un proceso hijo que calcule algo por su cuenta sin interferir con el padre. Cada uno opera sobre sus propias variables.

Consigna · Escribí un programa que:

1. Cree un proceso hijo con `fork()`.
2. El padre sumará números del 1 al 1000 y mostrará el resultado.
3. El hijo multiplicará los mismos números y mostrará el resultado.
4. Cada uno imprimirá su PID para diferenciarse.

Entrada · Ninguna. **Salida** · Resultado de la suma (padre) y del producto (hijo), junto con sus PIDs.

Criterios de aceptación

- ☐ Usa `fork()` para crear el hijo.
- ☐ El padre realiza la suma, el hijo el producto.
- ☐ Cada proceso imprime su PID con `getpid()`.
- ☐ Las variables son independientes (no hay interferencia).

Ejercicio 23.3 — Árbol de procesos

Conceptos · `fork`, estructura de árbol, profundidad variable

Contexto · Creá un programa que cree un **árbol de procesos**: cada proceso puede tener hasta N hijos. Es una forma visual de entender la jerarquía padre-hijo.

Consigna · Escribí un programa que reciba dos argumentos: `profundidad` y `ramificacion`. En cada nivel del árbol, el proceso actual crea `ramificacion` hijos. Cada proceso imprime su PID, su PPID y el nivel en el árbol antes de terminar (o esperar a sus hijos).

Criterios de aceptación

- [ ] Crea correctamente un árbol de profundidad N con ramificación M.
- [ ] Cada proceso imprime su PID y PPID.
- [ ] Los padres esperan a todos sus hijos antes de imprimir "Nivel X completado".
- [ ] Maneja correctamente casos base (profundidad 0).

PARTE 3 — `exec*` : reemplazarte completamente

Después de `fork()`, el hijo sigue siendo tu proceso. Pero si llamás a `exec*`, **dejas de ser vos**. Todo tu código, tus variables, tu stack —todo— se reemplaza por otro programa. El PID permanece igual, pero el contenido es completamente distinto.

3.1 Las variantes de `exec`

Hay siete funciones `exec`. Se diferencian en cómo pasás los argumentos y el path:

Función	Path absoluto	Args como...	Terminado con...
`execl`	Sí (`/bin/ls`)	Separados	`NULL`
`execle`	Sí	Separados	`NULL` + envp[]
`execlp`	No (busca en PATH)	Separados	`NULL`
`execv`	Sí (`/bin/ls`)	`char *argv[]`	—
`execvp`	No (busca en PATH)	`char *argv[]`	—
`execvpe`	No (busca en PATH)	`char *argv[]` + envp[]	—

Las más útiles son:

`execlp`: path relativo (busca en PATH), args separados, NULL al final:

```
// Ejecuta "ls -la /tmp"
execlp("ls", "ls", "-la", "/tmp", NULL);
```

Nota el primer argumento es **el nombre del programa** (como aparece en `ps`), después los argumentos.

`execvp`: path relativo, args como arreglo:

```
char *args[] = {"ls", "-la", "/tmp", NULL};
execvp("ls", args);
```

> *`p` significa "PATH". `execlp` y `execvp` buscan el executable en las rutas de `PATH`. `execl` y `execv` necesitan la ruta completa: `/bin/ls`, `/usr/bin/python3`, etc.*

3.2 El patrón fork-exec completo

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

pid_t pid = fork();

if (pid < 0) {
    perror("fork falló");
    exit(1);
} else if (pid == 0) {
    // HIJO: reemplazate por "wc -l archivo.txt"
    char *argv[] = {"wc", "-l", "archivo.txt", NULL};
    execvp(argv[0], argv);

    // Si llegamos acá, exec FALLÓ (el programa no se ejecutó)
    perror("exec falló");
    _exit(1);  // Importante: usar _exit, no exit()
}

// PADRE: seguís corriendo. Esperar al hijo.
int status;
waitpid(pid, &status, 0);
printf("Hijo terminó con status %d\n", WEXITSTATUS(status));
```

3.3 Herencia de file descriptors

Cuando hacés `fork()`, el hijo hereda **todos los fds abiertos** del padre: stdin (0), stdout (1), stderr (2), y cualquier fd que hayas abierto con `open()` o `pipe()`. Esto es útil para redirección:

```c
// El hijo hereda el archivo de log abierto con dup2(stdout)
int log_fd = open("app.log", O_WRONLY | O_CREAT | O_APPEND, 0644);
dup2(log_fd, 1);  // stdout ahora apunta al archivo

pid_t pid = fork();
if (pid == 0) {
    // Hijo: todo lo que imprima por printf va al archivo de log
    execlp("mi_programa", "mi_programa", NULL);
}
```

Práctica de la Parte 3

Ejercicio 23.4 — Lanzador de comandos

Conceptos · `fork`, `execvp`, manejo de errores, `_exit`

Contexto · Escribí un programa que reciba una línea de comando por stdin y la ejecute usando `fork` + `execvp`. Debe soportar cualquier comando que el usuario escriba.

Consigna · El programa lee líneas de stdin con `fgets()`, las separa en argumentos (por espacios), y ejecuta cada una como un comando con `execvp()`. Si el comando no se encuentra, imprime error.

Entrada · Comandos por línea: `ls -la`, `grep foo *.txt`, `date`

Salida · Salida del comando ejecutado.

Criterios de aceptación

- [ ] Usa `fgets()` para leer líneas de stdin.
- [ ] Separa la línea en argumentos (splitea por espacio).
- [ ] Usa `fork()` y `execvp()` correctamente.
- [ ] Maneja errores de exec con `perror()` y `_exit(1)` en el hijo.

Ejercicio 23.5 — "Mi propio shell"

Conceptos · `fork`, `execvp`, `waitpid`, bucle interactivo, parsing de comandos

Contexto · Escribí un shell mínimo que ejecute comandos del usuario, con soporte para: comandos simples, argumentos y redirección básica (`>`).

Consigna · Un bucle infinito que:

1. Imprima el prompt `$`
2. Lea un comando con `fgets()`
3. Parsee el comando (separar en tokens)
4. Detecte si hay redirección (`> archivo`) y abra/prepare el archivo destino
5. Ejecute con `fork()` + `execvp()`

Criterios de aceptación

- [ ] Bucle interactivo que termina con `exit` o EOF.
- [ ] Soporta comandos simples con argumentos.
- [ ] Soporta redirección básica con `>`.
- [ ] Espera cada comando antes de seguir (`waitpid`).

PARTE 4 — `wait`, `waitpid`: esperar a que terminen

Cuando creás un hijo con `fork()`, el proceso sigue corriendo después del fork. Si querés saber **cuándo terminó** y **cómo terminó**, necesitás `wait()` o `waitpid()`.

4.1 `wait()`: la forma simple de esperar

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

pid_t wait(int *status);
```

- Bloquea el padre hasta que **cualquier hijo** termine.
- Devuelve el PID del hijo que terminó.
- Escribe el estado de salida en `*status` (macros para interpretar).
- Si no hay hijos, devuelve `-1`.

```
pid_t pid = fork();
if (pid == 0) {
    _exit(42);  // Hijo termina con código 42
}

int status;
pid_t child = wait(&status);
// child == pid
// WEXITSTATUS(status) == 42
printf("Hijo %d terminó con código %d\n", child, WEXITSTATUS(status));
```

4.2 `waitpid()` : el control fino

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

pid_t waitpid(pid_t pid, int *status, int options);
```

- `pid > 0` : espera solo al proceso con ese PID.
- `pid == -1` : espera a cualquier hijo (igual que `wait()`).
- `pid == 0` : espera a cualquier hijo del mismo grupo de procesos.
- `pid < -1` : espera a cualquier hijo del grupo `-pid` .

Options:

Option	Significado
`0`	Bloquea hasta que el hijo termine (comportamiento por defecto)
`WNOHANG`	No bloquea: devuelve inmediatamente si el hijo no terminó
`WUNTRACED`	También devuelve si el hijo fue detenido (para debugging)

```
// No bloqueante: revisar sin esperar
pid_t result = waitpid(-1, &status, WNOHANG);
```

```c
if (result == 0) {
    // El hijo aún está corriendo. Puedo hacer otra cosa.
} else if (result > 0) {
    // Hijo terminó con este PID. Revisar status.
} else {
    // Error o no hay hijos.
}
```

> ***WNOHANG*** *es clave para el ejercicio jefe de este capítulo: permite monitorear múltiples hijos sin bloquearse mientras unos terminan y otros siguen trabajando.*

4.3 Interpretar `status` : macros esenciales

El `int status` que devuelve `wait()` / `waitpid()` es un valor comprimido. Usá las macros para leerlo:

Macro	Qué devuelve
`WIFEXITED(status)`	Verdadero si el hijo terminó normalmente (`_exit` , `exit` , o retorno de main)
`WEXITSTATUS(status)`	Código de salida (0-255). **Solo válido si** `WIFEXITED` es verdadero
`WIFSIGNALED(status)`	Verdadero si el hijo fue asesinado por una señal (segfault, SIGKILL...)
`WTERMSIG(status)`	Número de la señal que mató al hijo. **Solo válido si** `WIFSIGNALED` es verdadero

```c
if (WIFEXITED(status)) {
    printf("Terminó normalmente con código %d\n", WEXITSTATUS(status));
} else if (WIFSIGNALED(status)) {
    printf("Asesinado por señal %d%s\n", WTERMSIG(status),
           WTERMSIG(status) == SIGSEGV ? " (segfault)" : "");
}
```

4.4 El zombie: el hijo que no muere

Cuando un hijo termina, su espacio de memoria se libera, pero **queda una entrada en la tabla de procesos** con su estado de salida hasta que el padre lo reclame con `wait()`. A esto se llama **zombie**:

```
PID   STATUS    COMMAND
450   S+        bash
451   Z+        mi_proceso <defunct>  ← zombie! esperando wait()
```

El zombie no consume CPU ni memoria significativa, pero sí una entrada en la tabla de procesos. Si el padre nunca llama a `wait()`, los zombies se acumulan y eventualmente el sistema no puede crear nuevos procesos (`EAGAIN`).

```
// Si el padre muere antes que el hijo, el hijo queda huérfano y pasa a ser hijo de init
(PID 1).
// Init llama a wait() periódicamente para limpiar todos los zombies.
// Por eso nunca ves un zombie con PID huérfano: init lo limpia automáticamente.
```

Práctica de la Parte 4

Ejercicio 23.6 — Contador de procesos terminados

Conceptos · `wait`, `waitpid`, `WEXITSTATUS`, `WIFEXITED`

Contexto · Creá N procesos hijos y contá cuántos terminaron con éxito (código 0) y cuántos fallaron (código distinto de 0). Usá `waitpid()` para controlar qué hijo esperás.

Consigna · El programa recibe como argumento la cantidad de hijos a crear. Cada hijo elige aleatoriamente si termina con éxito (exit code 0) o falla (exit code 1-255). El padre espera a todos, cuenta los éxitos y fracasos, e imprime un resumen.

Criterios de aceptación

- ☐ Crea N procesos hijos con `fork()`.
- ☐ Cada hijo elige aleatoriamente exit code (éxito o fracaso).
- ☐ El padre usa `waitpid()` para esperar a cada uno.
- ☐ Cuenta y reporta éxitos vs fracasos correctamente.

Ejercicio 23.7 — Zombie factory: crear y limpiar zombies

Conceptos · `waitpid`, zombies, señales de limpieza

Contexto · Este ejercicio es **solo diagnóstico**: creá un programa que produzca zombies intencionalmente para luego entender por qué aparecen y cómo limpiarlos.

Consigna · Escribí dos programas:

1. **zombie_factory.c**: Crea 3 hijos con `fork()`, pero el padre **nunca llama a** `wait()`. Los hijos terminan inmediatamente. Observá el comportamiento con `ps`.
2. **zombie_cleaner.c**: Igual que factory, pero el padre llama a `waitpid(-1, NULL, WNOHANG)` en un bucle para limpiar zombies sin bloquearse.

Criterios de aceptación

- ☐ zombie_factory: los 3 hijos aparecen como `<defunct>` hasta que el proceso termina.
- ☐ zombie_cleaner: ninguno aparece como zombie; `waitpid` con WNOHANG los limpia correctamente.
- ☐ Ambos programas terminan limpiamente al final.

PARTE 5 — Estado de salida: exit code y señales

Cuando un proceso "muere", lo hace de una de dos formas: **terminó normalmente** (con código de error) o fue **asesinado por una señal** (segfault, SIGKILL, etc.).

5.1 Código de salida: el último número que imprimes

```c
int main(void) {
    return 42;  // exit code 42
}
// O equivalentemente:
int main(void) {
    exit(42);   // lo mismo
}
// O con syscall directa:
int main(void) {
    _exit(42);  // sin cleanup de stdio
}
```

En la línea de comandos, Linux guarda el código de salida en `$?` :

```
$ ./prog && echo "OK" || echo "FAIL"   # chequea exit code implícitamente
$ echo $?                              # muestra: 0 (éxito) o distinto (fallo)
$ false; echo $?                        # falso siempre devuelve 1
1
$ ls /no/existe; echo $?                # archivo no encontrado = 2
2
```

> ***Regla informal:*** *código* `0` *= éxito. Código* `1-255` *= fallo (cada número puede tener un significado). Códigos mayores a 255 se truncan a 8 bits:* `exit(300)` *equivale a* `exit(44)` *.*

5.2 Señales: el asesino silencioso

Un proceso puede ser matado por señales enviadas por el sistema, otros procesos o él mismo:

Señal	Número	Causa típica
`SIGINT`	2	Ctrl+C del usuario
`SIGKILL`	9	`kill -9 PID` (no se puede capturar)
`SIGSEGV`	11	Segmentation fault (acceso inválido a memoria)
`SIGABRT`	6	`abort()` llamado explícitamente
`SIGPIPE`	13	Escribir a un pipe sin lectores

```c
#include <signal.h>

void handler(int sig) {
    printf("Recibí señal %d\n", sig);
}

signal(SIGINT, handler);  // Capturar Ctrl+C y actuar en vez de morir
```

Práctica de la Parte 5

Ejercicio 23.8 — Analizador de salidas

Conceptos · `waitpid`, macros `WIFEXITED`, `WEXITSTATUS`, `WIFSIGNALED`, `WTERMSIG`

Contexto · Escribí un programa que ejecute múltiples hijos (cada uno con exit code diferente o señal) y analice el estado de cada uno después de que terminen.

Consigna · Crea 5 hijos:

1. Uno que termina con `exit(0)` (éxito).
2. Uno que termina con `exit(42)` (fallo con código específico).
3. Uno que se mata a sí mismo con `raise(SIGSEGV)` (segfault).
4. Otro que usa `_exit(127)`.
5. Un último que termina con `exit(255)`.

El padre espera a todos y para cada uno imprime: PID, si terminó normalmente o por señal, y el código/señal.

Criterios de aceptación

- [ ] Crea 5 hijos con comportamientos distintos.
- [ ] Usa `waitpid()` para esperar a cada uno.
- [ ] `Usa correctamente las macros para interpretar` status`.
- [ ] El output es claro y muestra tanto códigos como señales.

Ejercicio 23.9 — "¿Por qué moriste?"

Conceptos · Señales, `signal()`, `raise()`, `kill()`, `errno` post-signal

Contexto · Escribí un programa que demuestre cómo diferentes señales terminan procesos y cómo el padre puede detectar cada una.

Consigna · Crea dos hijos: uno que se mata con `SIGTERM` y otro con `SIGKILL`. El padre usa `waitpid()` para detectar ambos y reporta la señal que los mató. Bonus: un tercer hijo que captura `SIGINT` con `signal()` e imprime "¡No me mates!" en vez de morir.

Criterios de aceptación

- [ ] Detecta correctamente `SIGTERM`, `SIGKILL` y el hijo capturador.
- [ ] Usa `WIFSIGNALED()` y `WTERMSIG()` para reportar.
- [ ] El hijo con `signal()` muestra "¡No me mates!" en stdout.

PARTE 6 — Variables de entorno y `argv`

Cada proceso tiene un conjunto de **variables de entorno** (PATH, HOME, USER...) que usa el sistema y las aplicaciones para configurarse. También recibe argumentos a través de `argv`.

6.1 `argv`: los argumentos del programa

```c
int main(int argc, char *argv[]) {
    // argc = cantidad de argumentos
    // argv[0] = nombre del programa
    // argv[1], argv[2], ... = argumentos pasados desde la línea de comandos

    for (int i = 0; i < argc; i++) {
        printf("Arg %d: %s\n", i, argv[i]);
    }

    return 0;
}
```

```
$ ./mi_programa Hola Mundo
Arg 0: ./mi_programa
Arg 1: Hola
Arg 2: Mundo
```

6.2 Variables de entorno con `environ`

Las variables de entorno están disponibles a través del símbolo externo `environ`, declarado en `<unistd.h>`:

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

extern char **environ;

int main(void) {
    // Recorrer todas las variables de entorno
    for (int i = 0; environ[i] != NULL; i++) {
        printf("%s\n", environ[i]);
    }

    // O buscar una específica con getenv():
    char *home = getenv("HOME");
    if (home) {
        printf("HOME=%s\n", home);
    }

    return 0;
}
```

***`getenv()`**, **`setenv()`**, **`putenv()`** están en `<stdlib.h>`:*

- `getenv(name)` → *devuelve el valor de la variable o NULL si no existe.*
- `setenv(name, value, overwrite)` → *establece una variable (`overwrite=1` sobrescribe).*
- `putenv("NAME=value")` → *igual pero la cadena debe permanecer en memoria.*

Práctica de la Parte 6

Ejercicio 23.10 — Listador de envvars

Conceptos · `environ`, `getenv`, variables de entorno, `setenv`

Contexto · Escribí un programa que liste todas las variables de entorno del proceso actual y permita buscar una específica por nombre.

Consigna · El programa recibe opcionalmente el nombre de una variable como argumento:

- Sin argumentos: lista **todas** las variables de entorno (una por línea).
- Con un argumento (ejemplo: `./env PATH`): busca y muestra solo esa variable.
- Si la variable no existe, imprime "Variable no encontrada".

Criterios de aceptación

- ☐ Usa `environ` para recorrer todas las variables.
- ☐ Soporta búsqueda específica con `getenv()`.
- ☐ Maneja el caso donde la variable no existe.
- ☐ Muestra output en formato `NOMBRE=valor`.

Ejercicio 23.11 — "Mi propio export"

Conceptos · `setenv`, `putenv`, `getenv`, herencia de variables en fork/exec

Contexto · Escribí un programa que funcione como un mini-`export`: permite establecer, mostrar y eliminar variables de entorno. Al final, crea un proceso hijo con `execvp()` para verificar que las variables seheredan al nuevo programa.

Consigna · Un bucle interactivo que acepte:

- `SET nombre valor` → establece la variable.
- `SHOW [nombre]` → muestra todas o solo esa.
- `UNSET nombre` → elimina la variable (`unsetenv`).
- `RUN comando...` → ejecuta un comando (con `fork` / `exec`) que imprime las variables de entorno para verificar la herencia.

Criterios de aceptación

- [ ] Implementa SET, SHOW, UNSET correctamente.
- [ ] Usa `setenv()`, `getenv()` y `unsetenv()` de `<stdlib.h>`.
- [ ] El comando RUN ejecuta un proceso hijo que muestra sus propias variables de entorno.
- [ ] Las variables establecidas en el padre aparecen en el hijo.

PARTE 7 — UID, GID, umask: tu identidad en Linux

Además del PID, cada proceso tiene una identidad de seguridad: usuario real/efectivo, grupo, y permisos por defecto para nuevos archivos (umask).

7.1 UID real vs efectivo

```c
#include <unistd.h>
#include <sys/types.h>

uid_t getuid(void);      // Usuario real (quién ejecutó el programa)
uid_t geteuid(void);     // Usuario efectivo (qué permisos tiene el programa)
gid_t getgid(void);      // Grupo real
gid_t getegid(void);     // Grupo efectivo
```

La diferencia entre UID real y efectivo aparece con **setuid**: cuando un ejecutable tiene el bit setuid, se ejecuta con los permisos del **propietario del archivo**, no del que lo lanzó:

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 68K ... /usr/bin/passwd  # ← 's' en vez de 'x' = setuid bit!
$ whoami
horacio
$ /usr/bin/passwd
Cambiar contraseña para horacio.
(Puedo modificar archivos de /etc/shadow porque el binary es owned por root)
```

7.2 Umask: los permisos que heredás al crear archivos

El **umask** determina qué permisos se **quitan** a nuevos archivos cuando los creás con `open()` / `mkdir()`:

Umask	Permisos resultantes de un archivo creado con 0666
`0022` (default)	`0644` (rw-r--r--)
`0002` (grupo writable)	`0664` (rw-rw-r--)
`0077` (solo propietario)	`0600` (rw-------)

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

mode_t umask(mode_t mask);  // Devuelve el umask anterior, establece uno nuevo.

// Guardar el umask actual y ponerle cero temporalmente:
mode_t old_mask = umask(0);     // Quitar todas las restricciones de umask
int fd = open("privado.txt", O_WRONLY | O_CREAT, 0666);  // Crea con permisos exactos (no modificados por umask)
umask(old_mask);                // Restaurar el umask original
close(fd);
```

> ***Regla mnemotécnica:*** *`umask` es una **máscara de exclusión**. Lo que tenga 1 en la máscara se quita. Un umask de `0022` significa "quitar write al grupo y a otros". Por eso los archivos nuevos son `644`: 6 (rw-) sin nada, 4 (r--) sin write de grupo, 4 (r--) sin write de otros.*

Práctica de la Parte 7

Ejercicio 23.12 — "¿Quién soy realmente?"

Conceptos · `getuid`, `geteuid`, `getgid`, `getegid`, `umask`

Contexto · Escribí un programa que muestre toda la información de seguridad del proceso actual: UIDs, GIDs, umask actual, y permisos efectivos.

Consigna · El programa debe imprimir:

1. UID real y efectivo.
2. GID real y efectivo.
3. Umask actual (en octal).
4. Permisos que tendría un archivo creado ahora con `open("f", 0666)`.
5. Si el proceso tiene setuid activo (UID real != efectivo).

Criterios de aceptación

- [ ] Usa correctamente `getuid()`, `geteuid()`, `getgid()`, `getegid()`.
- [ ] Obtienes y muestras el umask con `umask(0)` guardando/restaurando el valor.
- [ ] Calcula los permisos efectivos de un nuevo archivo a partir del umask.
- [ ] Indica si setuid está activo.

PARTE 8 — Ejercicio jefe: copiador recursivo paralelo

Llegó el momento de juntar todo lo que aprendiste sobre `fork()`, syscalls crudas, monitoreo asíncrono y estadísticas en un solo programa realista: un **copiador de directorios que crea un proceso hijo por archivo copiado**.

8.1 El desafío

Escribí un programa `copiar-dir` que recibe dos argumentos por línea de comandos:

1. **directorio origen** (ruta del directorio a copiar)
2. **directorio destino** (donde se copiará el árbol)

El programa debe hacer lo siguiente:

Estructura general

```
main(argc, argv[1]=origen, argv[2]=destino)
│
├── scan_and_copy(dir_origen, dir_destino, &stats)
│   │
│   ├── Para cada entry del directorio (opendir/readdir):
│   │   │
│   │   ├── Si es archivo regular:
│   │   │   ├── fork()
│   │   │   │   HIJO: copy_file(origen/archivo, destino/archivo) + exit(0 o 1)
│   │   │   └── PADRE: stats.total++ (contador local de este nivel)
│   │   │
│   │   └── Si es subdirectorio (no "." ni ".."):
│   │       ├── mkdir(dir_destino/nombre) con permisos 0755
│   │       └── scan_and_copy(origen/nombre, destino/nombre, &stats) ← padre llama
recursivamente
│   │
│   └── Durante el recorrido: waitpid(-1, &status, WNOHANG) para recoger estadísticas de
hijos terminados
│
└── Al final imprimir resumen:
    Archivos copiados:    42
    Archivos fallidos:     0
    Total de archivos:   42
    Directorios creados:  7
```

Reglas clave

1. **Cada archivo regular se copia en un proceso hijo separado.** El hijo usa `open/read/write` (del capítulo 22) para copiar el contenido. No usás `exec`: el hijo mismo hace la copia.
2. **Los subdirectorios se crean en el padre** (no spawnea hijos para directorios). Se recurren de forma secuencial desde el padre.
3. **El padre monitorea hijos con `waitpid(-1, &status, WNOHANG)`** en cada vuelta del bucle principal para recolectar estadísticas sin bloquearse.
4. **Cada hijo termina con `exit(0)` si copió bien o `exit(1)` si falló.** El padre verifica el estado con `WIFEXITED` y `WEXITSTATUS`.

8.2 Funciones auxiliares sugeridas

```c
// Estructura de estadísticas compartida por toda la recursión.
typedef struct {
    long files_copied;    // archivos copiados con éxito
    long files_failed;    // archivos que fallaron al copiarse
    int  dirs_created;    // directorios creados
} CopyStats;

// Copiar un solo archivo: abre origen, lee en chunk, escribe destino.
int copy_file(const char *src_path, const char *dst_path);

// Crear recursivamente un árbol de directorios copiando archivos con fork() por cada
uno.
void scan_and_copy(const char *src_dir, const char *dst_dir, CopyStats *stats);

// Mostrar estadísticas finales formateadas.
void print_stats(const CopyStats *stats, double elapsed_sec);
```

8.3 Puntos clave de implementación

Medir tiempo total

Usá `clock_gettime(CLOCK_MONOTONIC, &ts)` para obtener timestamps y calcular la duración:

```c
#include <time.h>

struct timespec start_ts, end_ts;
clock_gettime(CLOCK_MONOTONIC, &start_ts);

// ... ejecutar copia ...

clock_gettime(CLOCK_MONOTONIC, &end_ts);
double elapsed = (end_ts.tv_sec - start_ts.tv_sec) +
                 (end_ts.tv_nsec - start_ts.tv_nsec) / 1e9;
```

Recolectar estadísticas sin bloquear

En cada vuelta del bucle principal (después de spawnea un hijo o antes de procesar el siguiente entry), llamá a `waitpid` con `WNOHANG`:

```
// Recoger todos los hijos terminados que estén disponibles SIN bloquearse.
while (1) {
    int status;
    pid_t pid = waitpid(-1, &status, WNOHANG);

    if (pid == 0) {
        // No hay más hijos terminados — seguir con el bucle.
        break;
    } else if (pid > 0) {
        // Un hijo terminó.
        if (WIFEXITED(status)) {
            int code = WEXITSTATUS(status);
            if (code == 0) {
                stats->files_copied++;
            } else {
                stats->files_failed++;
            }
        }
    } else {
        // Error real — salir del bucle.
        break;
    }
}
```

Copiar un archivo con syscalls crudas

El hijo debe copiar un archivo usando `open/read/write`:

```c
int copy_file(const char *src, const char *dst) {
    int infd = open(src, O_RDONLY);
    if (infd < 0) return 1;  // Falló abrir origen

    int outfd = open(dst, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (outfd < 0) { close(infd); return 1; }  // Falló abrir destino

    char buf[4096];
    ssize_t n;
    while ((n = read(infd, buf, sizeof(buf))) > 0) {
        ssize_t written = 0;
        while (written < n) {
            ssize_t w = write(outfd, buf + written, n - written);
            if (w < 0) break;
            written += w;
        }
        if (written < n) break;  // Error de escritura
    }

    close(infd);
    close(outfd);
    return (n >= 0 || errno == 0) ? 0 : 1;
}
```

Nota: *esta función devuelve* `0` *si la copia fue exitosa y* `1` *si falló en algún paso. El hijo la llama y luego hace* `exit(code)`.

8.4 Ejercicio: implementación completa

Ejercicio 23.13 — Copiador recursivo paralelo

Conceptos · `fork` , `waitpid` , `WNOHANG` , `dirent.h` , `sys/stat.h` , `open/read/write` , `mkdir()` , `errno.h` , estadísticas concurrentes, `clock_gettime`

Contexto · Copiar un directorio grande archivo por archivo es lento si se hace secuencialmente. Este ejercicio crea un proceso hijo **por cada archivo**, que copia su contenido, mientras el padre monitorea el progreso sin copiar nada. Es una simplificación porque no manejamos enlaces simbólicos, permisos especiales ni casos de error extremos.

Consigna · Implementá `copiar-dir` siguiendo la estructura y reglas descritas en las secciones 8.1-8.3.

El programa debe:

1. Recorrer recursivamente el árbol de directorios origen.
2. Por cada **archivo regular**, crear un **proceso hijo** con `fork()` que copie ese archivo. El hijo usa `open/read/write` (no `exec`). Termina con `exit(0)` si copió bien, `exit(1)` si falló.
3. Por cada **subdirectorio** (excluyendo `.` y `..`): crear el directorio en destino con `mkdir()` y llamar recursivamente a `scan_and_copy()` desde el proceso padre (**no** spawnea hijos para directorios).
4. Mientras los hijos corren, el padre debe llamar a `waitpid(-1, &status, WNOHANG)` en cada vuelta del bucle para **recolectar estadísticas sin bloquearse**.
5. Al terminar, imprimir un resumen con formato:

```
Archivos copiados:    42
Archivos fallidos:     0
Total de archivos:   42
Directorios creados:  7
Tiempo total:         0.34s
```

Entrada · Dos rutas por argumento: `./copiar-dir /origen /destino`

Salida · El árbol copiado + resumen estadístico en stdout.

Ejemplo de ejecución: Dado un directorio origen así:

```
origen/
├── README.md
├── src/
│   ├── main.c
│   └── util.c
└── data/
    └── input.txt
```

La salida por consola sería:

```
Archivos copiados:     3
Archivos fallidos:     0
Total de archivos:     3
Directorios creados:   2
Tiempo total:          0.12s
```

Criterios de aceptación

- [ ] Copia recursivamente todos los archivos y directorios del árbol origen al destino.
- [ ] Cada archivo regular se copia en un **proceso hijo separado** con `fork()`.
- [ ] El padre usa `waitpid` con `WNOHANG` para monitorear hijos sin bloquearse.
- [ ] Los subdirectorios se crean en el proceso padre (no spawnea hijos para directorios).
- [ ] Imprime estadísticas correctas: copiados, fallidos, total, directorios creados.
- [ ] Maneja errores: si un hijo falla al copiar, se cuenta como "fallido" y no se detiene el resto.
- [ ] El árbol destino es idéntico (misma estructura) al origen.
- [ ] Mide tiempo con `clock_gettime(CLOCK_MONOTONIC, ...)`.

Criterios de aceptación bonus:

- [ ] Soporta enlaces simbólicos (solo los lista, no copia contenido).
- [ ] Imprime progreso en tiempo real (porcentaje o contador de archivos restantes).
- [ ] Permite establecer un límite máximo de hijos concurrentes con una variable de entorno.