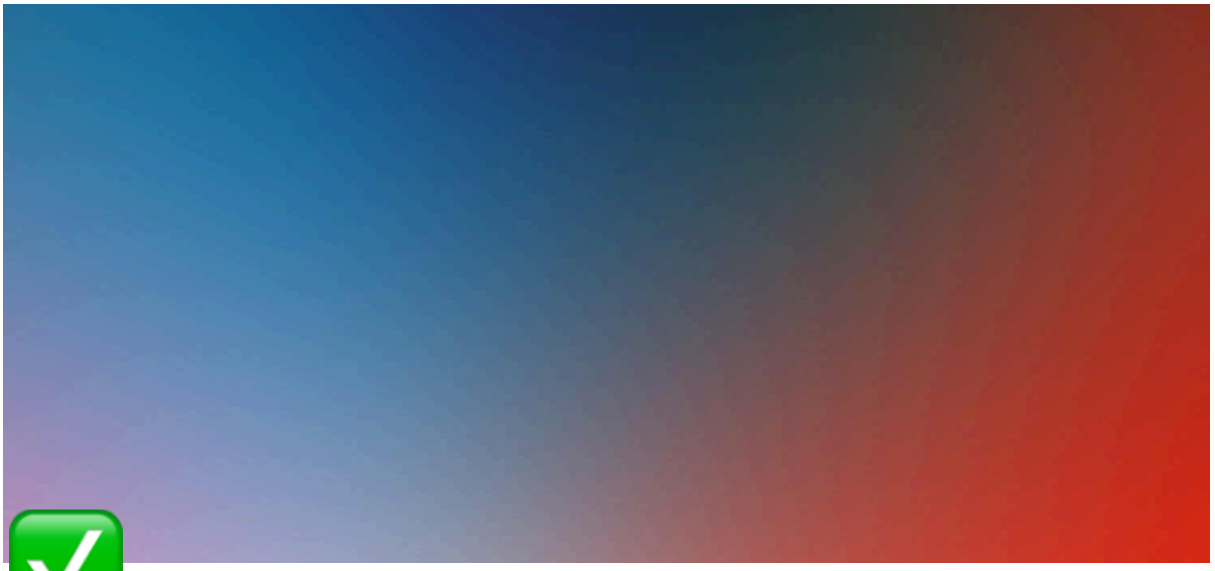

SEÑALES

🔷 Tema 1: ¿Qué son las señales y por qué son importantes?

📘 Explicación teórica clara y profunda

¿Qué es una señal en UNIX/POSIX?

Una **señal** es un mecanismo que usan los sistemas UNIX y POSIX para notificar a un proceso que ha ocurrido un evento. Este mecanismo es **asíncrono**, es decir, la señal puede interrumpir el flujo de ejecución de un proceso en cualquier momento.

Es como si el sistema operativo le dijera al proceso:

> “¡Oye, acaba de pasar esto! ¡Haz algo al respecto ahora mismo!”

Las señales permiten que un proceso:

- Se comunique con otros procesos.
- Responda a eventos externos (como una combinación de teclas o la expiración de un temporizador).
- Termine o modifique su comportamiento ante situaciones críticas.

Ejemplos comunes de señales:

Señal	Nombre	Significado común
2	`SIGINT`	Interrupción (Ctrl+C)
9	`SIGKILL`	Terminación forzada (no capturable)
15	`SIGTERM`	Solicitud de terminación ordenada
11	`SIGSEGV`	Violación de segmento (segfault)
17/18/20	`SIGCHLD`	Hijo terminó
19	`SIGSTOP`	Detener proceso (no capturable)

¿Cómo funcionan?

- El **kernel** detecta un evento (como una tecla presionada o una señal enviada por otro proceso).
- Coloca una señal en la "cola de señales" del proceso.
- Cuando el proceso está ejecutando, **interrumpe su flujo normal**, y:
  - Ejecuta un **manejador de señales (handler)** definido por el programador, o
  - Ejecuta la acción por defecto de esa señal (como terminar el proceso).

¿Por qué son importantes?

- Permiten **control y comunicación simple** entre procesos.
- Son esenciales para manejar eventos como interrupciones del usuario, finalización de procesos hijos, o límites de tiempo.
- En programación concurrente y sistemas embebidos, se usan para sincronizar y coordinar procesos sin compartir memoria.

3. ¿Por qué no se puede capturar señales como `SIGKILL` o `SIGSTOP`?

- Porque son señales **reservadas por el kernel** para controlar directamente los procesos. Permitir que un proceso las capture o ignore **iría en contra de la estabilidad del sistema operativo**.

🔷 Tema 2: `signal.signal()` y funciones relacionadas en Python

📘 1. Explicación teórica clara

El módulo `signal` de Python te permite controlar cómo reacciona un proceso frente a ciertas **señales del sistema operativo**.

🧩 ¿Qué es `signal.signal()`?

Es la función que **registra un manejador** (handler) para una señal específica. Su forma general es:

```
signal.signal(señal, manejador)
```

- `señal`: una constante del módulo `signal`, como `signal.SIGINT`, `signal.SIGTERM`, etc.
- `manejador`: una función definida por vos que se va a ejecutar cuando llegue la señal.

🔧 Por defecto, si no hay manejador definido, el sistema hace una **acción predefinida** (terminar, ignorar, etc.).

⚙️ Funciones relacionadas útiles

Función	Propósito
`signal.signal()`	Registrar un handler para una señal.
`signal.getsignal()`	Saber qué handler está registrado para una señal.
`signal.pause()`	Pausa la ejecución hasta que llegue una señal.
`signal.SIG_IGN`	Ignorar una señal.
`signal.SIG_DFL`	Restaurar el comportamiento por defecto de una señal.
`os.kill(pid, señal)`	Enviar señales entre procesos.
`os.getpid()`	Obtener el PID del proceso actual.

🚫 Restricciones importantes

- **Solo el hilo principal puede usar `signal.signal()` en Python.**
- No se pueden interceptar `SIGKILL` ni `SIGSTOP`.

- Las señales funcionan bien en **procesos**, pero su uso en **multihilo** requiere cuidado (veremos más adelante).

🔷 Tema 3: `kill`, `sigqueue` y `sigaction` (con referencia cruzada a C)

📘 1. Explicación teórica clara

Las señales no solo pueden generarse desde el teclado (como `Ctrl+C`), también pueden ser enviadas **programáticamente** entre procesos. Aquí aparecen herramientas clave:

🔨 `kill` (en shell y en C)

- En terminal:

```
kill -SIGINT <PID>
```

- Envía una señal (`SIGINT`, `SIGTERM`, etc.) al proceso con PID `<PID>`.
- En C (o Python con `os.kill()`):

```
kill(pid, SIGINT);
```

```
import os, signal
os.kill(pid, signal.SIGINT)
```

🔄 `sigqueue` (C)

- Permite enviar **una señal con datos adicionales** (`union sigval`), a diferencia de `kill`.
- Sintaxis en C:

```
sigqueue(pid, SIGUSR1, (union sigval){.sival_int = 42});
```

En Python no existe `sigqueue()` directamente. Es parte de **señales en tiempo real**, que veremos **más adelante**.

⚙️ `sigaction` (C)

- API más avanzada que `signal()` en C.
- Permite configurar mejor el comportamiento del handler:

  máscaras, flags, recibir información extendida (`siginfo_t`), etc.

Ejemplo en C:

```
struct sigaction sa;
sa.sa_handler = mi_manejador;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGINT, &sa, NULL);
```

Python no usa `sigaction` directamente, pero **el módulo `signal` de Python lo abstrae internamente**.

⚠️ Comparación rápida

Función	Disponible en Python	Comentario
`kill`	✅ (`os.kill()`)	Enviar señales
`sigqueue`	❌ (solo en C)	Enviar señales con datos extra
`sigaction`	❌ (solo en C)	Control avanzado del manejador

🔷 Tema 4: Uso de señales para sincronizar procesos

📘 1. Explicación teórica clara

Las señales también se pueden usar como una forma **básica de comunicación entre procesos**, especialmente para **sincronizarlos**: por ejemplo, que un proceso espere a que otro le dé una orden antes de continuar.

🧠 ¿Qué significa sincronizar procesos?

- Asegurar que **un proceso no avance** hasta que **otro haya hecho algo específico**.

- Ejemplo típico: el proceso padre crea un hijo y **espera una señal** de este antes de continuar (o viceversa).

🧰 ¿Por qué usar señales para sincronizar?

- Son livianas, no requieren memoria compartida ni sockets.
- Se pueden usar para enviar eventos como: *"ya terminé"*, *"podés continuar"*, etc.
- **Limitación**: las señales no transportan datos (excepto `sigqueue`), por lo que solo sirven como **disparadores**.

📌 Proceso típico de sincronización con señales

1. **Registrar un handler** con `signal.signal()`.
2. **Crear el otro proceso** (con `os.fork()`).
3. El proceso que espera llama a `signal.pause()` o `time.sleep()` en bucle.
4. El otro proceso envía la señal con `os.kill()`.

🔷 Tema 5: Manejo seguro y async-signal-safe

📘 1. Explicación teórica clara

Las señales en sistemas UNIX tienen un comportamiento especial en cuanto a la seguridad de su manejo. Algunos manejadores de señales pueden **interrumpir el flujo normal de ejecución** de un programa, lo que podría causar problemas si no se gestionan correctamente.

Manejo seguro de señales significa que un proceso puede manejar señales de manera confiable sin corromper su estado interno ni causar errores difíciles de rastrear.

🧠 ¿Qué significa "async-signal-safe"?

- **Funciones async-signal-safe** son aquellas que se pueden llamar de manera segura **desde un handler de señales**.
- El problema principal con el manejo de señales es que **no se pueden llamar funciones no seguras** dentro de un handler, como `malloc()` (en C) o incluso funciones de la biblioteca estándar de C, que pueden no ser reentrantes.

- En Python, las funciones seguras incluyen cosas como `os._exit()` (termina el proceso de manera inmediata y segura) o `signal.signal()`, pero **no puedes usar cosas como `input()` o `print()` dentro de un handler**, ya que no son seguras.

🧰 ¿Cómo implementar un manejador seguro?

1. **Evitar operaciones complicadas** dentro de un handler, como:
   - **Lecturas o escrituras a archivos**.
   - **Acceso a estructuras complejas**.
   - **Uso de bibliotecas no reentrantes**.
2. **Evitar el uso de funciones bloqueantes** como `input()` o `time.sleep()`.

🔷 Tema 6: Señales en sistemas multihilo

📘 1. Explicación teórica clara

En sistemas con **multihilos**, el manejo de señales puede ser un poco más complejo. En estos sistemas, **no todas las señales se entregan a todos los hilos** de manera directa. La señal es entregada a un hilo en particular, y **no siempre al hilo principal**, que puede ser el que espera o maneja señales.

🧠 ¿Cómo funciona el manejo de señales en un entorno multihilo?

1. **Señales y el hilo principal**:

   En muchos sistemas UNIX y POSIX, las señales son entregadas **solo al hilo principal** si no se especifica otro hilo al que enviarlas. Esto significa que, si tienes varios hilos en tu proceso, solo el hilo principal recibirá la señal por defecto.

2. **Uso de `pthread_sigmask` y `sigwait` (en C)**:

   En sistemas multihilo, para controlar de manera explícita las señales que se entregan a ciertos hilos, se utilizan funciones como `pthread_sigmask` para bloquear señales y `sigwait` para que un hilo las espere.

3. **Manejo en Python**:

   En **Python**, el módulo `signal` maneja señales en el hilo principal, y el manejo de señales desde hilos secundarios **no es directo**. Si un hilo secundario

quiere recibir una señal, **deberá hacerlo a través del hilo principal**.

🧰 ¿Cómo gestionar señales en un programa multihilo en Python?

En Python, si queremos gestionar señales en un programa multihilo, necesitamos configurar el manejo de señales en el hilo principal, ya que solo el hilo principal puede registrar handlers de señales.

🔷 Tema 7: Comparación de señales con otros mecanismos de comunicación entre procesos (IPC)

📘 1. Explicación teórica clara

En los sistemas operativos UNIX y POSIX, las señales son solo **uno de varios mecanismos de IPC (Inter-Process Communication)**. A continuación, comparamos **señales** con otros mecanismos comunes:

Mecanismo IPC	¿Qué es?	Ventajas	Desventajas
Señales	Notificaciones asíncronas de eventos (como interrupciones)	Rápidas, simples, estándar	Limitadas en contenido (solo int), no garantizan orden
Pipes	Canal unidireccional para enviar datos entre procesos	Fáciles de usar, permiten flujo de datos	Solo entre procesos relacionados, no asíncronos
Colas de mensajes	Permiten enviar estructuras complejas entre procesos	Se pueden priorizar mensajes	Más complejas de manejar, mayor overhead
Memoria compartida	Áreas de memoria accesibles por varios procesos	Muy rápida, eficiente para mucho dato	Requiere sincronización manual (semáforos)
Sockets	Comunicación entre procesos (locales o en red)	Muy flexibles, permiten red/distribuido	Mayor complejidad, más código necesario

📌 ¿Cuándo usar señales?

Usá señales cuando:

- Necesites notificar un evento simple (por ejemplo, detener o reiniciar un proceso).

- No necesites enviar datos complejos.
- Requieras manejar interrupciones externas como `Ctrl+C`.

No son la mejor opción para enviar **grandes cantidades de datos o estructuras**.