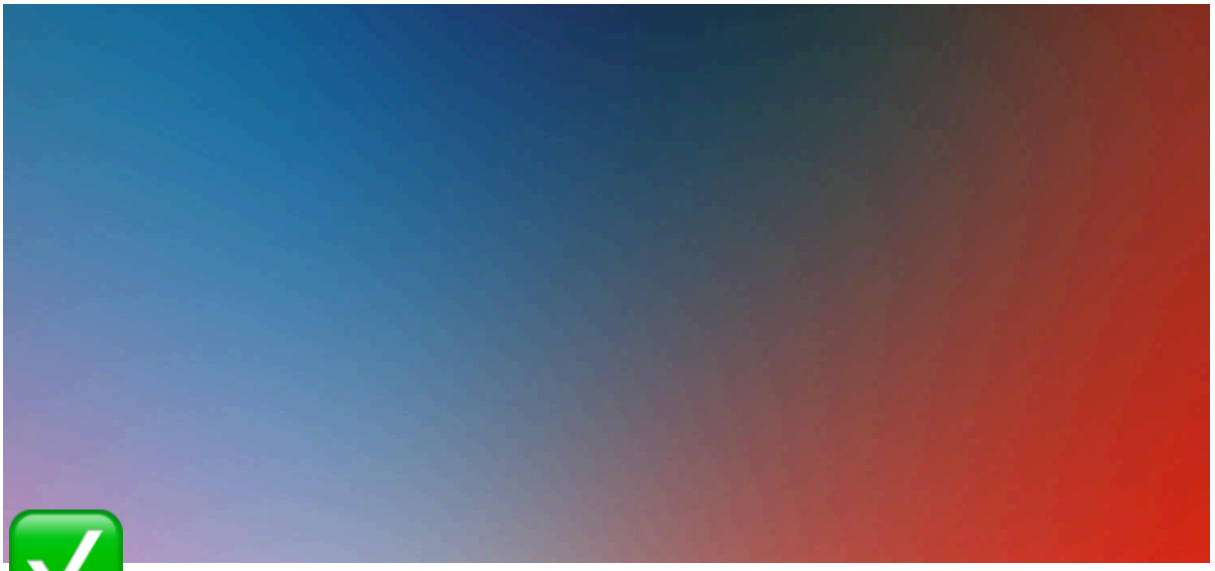

✅

PROCESOS

1. Fundamentos de procesos

¿Qué es un proceso?

Un **proceso** es una instancia en ejecución de un programa. Mientras que un **programa** es un conjunto de instrucciones almacenadas en un archivo ejecutable, un **proceso** es un programa que ha sido cargado en memoria y está siendo ejecutado por el sistema operativo.

Cada proceso tiene atributos clave que lo identifican:

- **PID (Process ID):** Identificador único del proceso.
- **Estado:** Puede ser *ejecutando*, *en espera*, *suspendido*, *terminado*, etc.
- **Espacio de direcciones:** Contiene código, datos, pila y heap.
- **Recursos asignados:** Archivos abiertos, memoria, permisos, etc.

Diferencias entre programa y proceso

Característica	Programa	Proceso
Estado	Pasivo (archivo en disco)	Activo (cargado en memoria)
Ubicación	Almacenamiento persistente (HDD/SSD)	Memoria RAM

Multiplicidad	Puede haber un solo archivo ejecutable	Puede haber múltiples procesos ejecutando el mismo programa (ej. varias instancias de un navegador)

Historia y evolución del concepto de procesos

En los primeros sistemas operativos, los programas se ejecutaban secuencialmente (uno a la vez). Con la llegada de los sistemas multiprogramados, los procesos se volvieron fundamentales para administrar la ejecución concurrente de múltiples programas. UNIX fue uno de los primeros sistemas en implementar un modelo robusto de procesos con la llamada `fork()`, permitiendo la creación de procesos hijos.

---

2. El modelo de procesos en UNIX/Linux

Los sistemas operativos tipo UNIX (Linux, macOS, BSD) utilizan un modelo jerárquico de procesos. Vamos a explorar cómo funciona.

2.1 Jerarquía de procesos y herencia

Cuando un proceso crea otro, este nuevo proceso se convierte en su **hijo**, formando una jerarquía.

🔷 **Ejemplo real:** Cuando abres una terminal y ejecutas un comando (`ls`, `python script.py`), la terminal (proceso padre) crea un nuevo proceso para ejecutar el comando.

Cada proceso en Linux tiene:

- **Un PID** (Process ID) único.
- **Un PPID** (Parent Process ID) que identifica a su proceso padre.

Puedes ver la jerarquía de procesos con:

```
pstree -p
```

2.2 El proceso init/systemd

🔷 **¿Quién crea los procesos?**

En sistemas modernos, el primer proceso que se ejecuta tras el arranque es **systemd** (o **init** en sistemas más antiguos). Este proceso tiene **PID 1** y es

responsable de iniciar todos los demás procesos.

Para verificarlo, usa:

```
ps -fp 1
```

Esto te mostrará información sobre el proceso `systemd` (o `init`).

2.3 Visualización de procesos con herramientas del sistema

Puedes ver los procesos en ejecución con:

🔷 `ps` (lista de procesos actuales)

🔷 `top` o `htop` (procesos en tiempo real)

🔷 `pstree` (muestra la jerarquía de procesos)

3. Manipulación de procesos con Python

Python nos permite crear y administrar procesos utilizando el módulo `os`. Aquí aprenderemos:

✅ Cómo crear procesos hijos con `fork()`

✅ Cómo ejecutar otros programas con `exec()`

✅ Cómo esperar a que los procesos hijos terminen con `wait()`

3.1 Creación de procesos hijos con `fork()`

En sistemas UNIX/Linux, la llamada `os.fork()` crea un nuevo proceso.

🔷 **El proceso padre recibe el PID del hijo.**

🔷 **El proceso hijo recibe un 0.**

📌 **Ejemplo de `fork()` en Python:**

```
import os

pid = os.fork()

if pid > 0:
    print(f"Soy el proceso padre, mi PID es {os.getpid()} y mi hijo tiene PID {pid
```

```
else:
    print(f"Soy el proceso hijo, mi PID es {os.getpid()} y mi padre tiene PID {os.
```

3.2 Ejecución de programas con `exec()`

El proceso hijo puede reemplazar su código con otro programa usando `exec()`.

📌 **Ejemplo de `exec()` en Python:**

```
import os

pid = os.fork()

if pid == 0:  # Código del hijo
    os.execlp("ls", "ls", "-l")  # Reemplaza el proceso hijo con el comando `ls
-l`
else:
    os.wait()  # El padre espera que el hijo termine
    print("Proceso hijo finalizado")
```

🔷 El hijo ejecuta `ls -l`, y el padre espera su finalización antes de continuar.

3.3 Espera y sincronización básica con `wait()`

Cuando un padre crea un hijo, puede usar `os.wait()` para esperar que termine.

📌 **Ejemplo de `wait()`:**

```
import os

pid = os.fork()

if pid == 0:
    print(f"Hijo (PID {os.getpid()}): Voy a dormir 3 segundos...")
    os.sleep(3)
    print("Hijo: Desperté y termino mi ejecución")
else:
    print(f"Padre (PID {os.getpid()}): Esperando a que termine mi hijo...")
    os.wait()
    print("Padre: Mi hijo terminó, ahora sigo yo")
```

🔷 **Observa el orden de ejecución y cómo el padre espera al hijo.**

3️⃣ ¿Por qué es útil `wait()`?

Si un padre no usa wait(), el hijo puede convertirse en un **proceso zombin** (veremos esto más adelante). wait() permite que el padre recoja el estado de terminación del hijo y libere sus recursos.

4. Procesos zombis y huérfanos

En esta sección aprenderemos:

✅ Qué es un proceso **zombi** y cómo evitarlo

✅ Qué es un proceso **huérfano** y cómo se maneja

4.1 Procesos zombis 👻

🔷 ¿Qué es un proceso zombi?

Un proceso zombi es aquel que **ya terminó su ejecución**, pero su información sigue en la tabla de procesos porque su padre no ha llamado `wait()`.

🔷 Ejemplo de proceso zombi:

```
import os
import time

pid = os.fork()

if pid == 0:  # Código del hijo
    print(f"Hijo (PID {os.getpid()}) terminando...")
    exit(0)  # El hijo termina inmediatamente
else:  # Código del padre
    print(f"Padre (PID {os.getpid()}), pero no llamaré wait()...")
    time.sleep(10)  # El padre sigue vivo sin llamar a wait()
```

🔷 Cómo detectar un zombi:
Mientras el programa está en ejecución, abre otra terminal y usa:

```
ps aux | grep Z
```

Si ves un proceso con estado **Z+**, es un zombi.

🔷 **Cómo evitar zombis:**

Usar `wait()`, `waitpid()` o manejar la señal `SIGCHLD`.

📌 **Ejemplo evitando zombis con `wait()`:**

```
import os
import time

pid = os.fork()

if pid == 0:
    print(f"Hijo (PID {os.getpid()}) terminando...")
    exit(0)
else:
    os.wait()  # El padre recoge el estado del hijo
    print("Padre: Mi hijo terminó, no hay zombis.")
```

4.2 Procesos huérfanos 🏚️

🔷 **¿Qué es un proceso huérfano?**

Un proceso huérfano es un proceso cuyo **padre termina antes que él**. En este caso, el sistema operativo **re-adecua el proceso huérfano bajo el proceso init/systemd** (PID 1).

🔷 **Ejemplo de proceso huérfano:**

```
import os
import time

pid = os.fork()

if pid > 0:
    print(f"Padre (PID {os.getpid()}) terminando...")
else:
    time.sleep(5)  # El hijo sigue corriendo después de que el padre muere
    print(f"Hijo (PID {os.getpid()}), ahora soy huérfano y mi nuevo padre es {os
```

🔷 **Cómo detectar un proceso huérfano:**

Ejecuta el código y luego usa:

```
ps -o pid,ppid,cmd
```

Si ves un proceso con **PPID 1**, es un huérfano.

RESPUESTAS DE MIS DUDAS

🔷 Si el valor de pid es:

```
==0: Estamos en el proceso hijo.

> 0: Estamos en el proceso padre, y pid contiene el PID del hijo recién crea
do.
```

📌 `exit(0)` → Evita que el hijo siga ejecutando el código del padre.

Expresión	¿Qué devuelve?	¿Dónde se usa?
`os.getpid()`	PID del proceso actual	En cualquier proceso
`pid`	PID del hijo (en el padre) o `0` (en el hijo)	Solo después de `fork()`
`os.getppid()`	PID del proceso padre del actual	En un proceso hijo