UT1: PROGRAMACIÓN MULTIPROCESO Y MULTIHILO

Contenido

1. Ejecutables, Procesos y Servicios

Conceptos fundamentales

Un **ejecutable** es un archivo que contiene un programa compilado con todas las instrucciones necesarias para su ejecución por parte del sistema operativo. Puede ser un archivo .exe en Windows o un binario en sistemas Linux/Unix.

Un **proceso** es un programa en ejecución que incluye el código (instrucciones), datos(variables), estado de los registros del procesador, contador de programa (program counter) y recursos asignados por el sistema operativo. Cada proceso tiene un identificador único (PID-Process ID) que lo distingue de los demás.

Los **servicios** o **daemons** (en sistemas Unix/Linux) son procesos especiales que se ejecutan en segundo plano sin interacción directa con el usuario. Se inician automáticamente al arrancar el sistema y permanecen activos hasta el apagado, proporcionando funcionalidades específicas como servidores web, bases de datos o gestores de red.

Diferencias clave entre proceso y servicio/daemon:

- Los procesos tienen un inicio y un fin definidos; los daemons permanecen en ejecución continua
- Los procesos pueden ser interactivos; los servicios operan sin terminal de control
- Los servicios se gestionan mediante administradores de sistema como systemd en Linux o Services en Windows

Estructura de un proceso en memoria

Un proceso se organiza en varias secciones

- **Código (text)**: instrucciones del programa
- **Datos (data)**: variables globales
- **Memoria dinámica (heap)**: memoria asignada dinámicamente durante la ejecución
- **Pila (stack)**: preserva el estado en invocaciones de funciones y procedimientos

Además, el sistema operativo mantiene un **Bloque de Control de Proceso** (**PCB**) que almacena información vital: estado del proceso, identificador, contador de programa, registros de CPU, información de planificación y asignación de memoria.

2. Estados de un Proceso y Planificación

Un proceso transita por diferentes estados durante su ciclo de vida:

- **Nuevo** (New): El proceso acaba de ser creado pero aún no está listo para ejecutarse.
- **Listo** (Ready): El proceso está preparado para ejecutarse y solo espera que el planificador le asigne el procesador.

- **En ejecución** (Running): El proceso tiene asignado un procesador y está ejecutando sus instrucciones.

- **Bloqueado/Esperando** (Waiting/Blocked): El proceso no puede continuar su ejecución porque espera un evento externo, como la finalización de una operación de E/S o una señal.

- **Terminado** (Terminated): El proceso ha finalizado su ejecución y el sistema libera sus recursos.

Planificación de procesos

El **planificador de procesos** (scheduler) es el componente del sistema operativo responsable de decidir qué proceso pasa al estado activo. Los algoritmos de planificación buscan optimizar el uso del procesador y minimizar el tiempo de espera

1. **FIFO** (**First In First Out**): También llamado FCFS (First Come First Served). Los procesos se ejecutan en orden de llegada, sin interrupción hasta su finalización. Es simple pero puede bloquear procesos cortos si uno largo está en ejecución.

2. **SJF** (**Shortest Job First**): Selecciona el proceso con menor tiempo de ejecución estimado. Puede provocar inanición (starvation) de procesos largos.

3. **SRT** (**Shortest Remaining Time**): Versión apropiativa de SJF que considera nuevos procesos que llegan. Si un nuevo proceso requiere menos tiempo que el restante del proceso actual, se produce un cambio de contexto.

4. **Round Robin** (**RR**): Asigna un cuanto de tiempo fijo a cada proceso de forma circular. Garantiza un tiempo de respuesta razonable para todos los procesos, aunque genera sobrecarga por los cambios de contexto.

5. **Prioridades**: Cada proceso tiene asignada una prioridad y el de mayor prioridad pasa a ejecución. Para evitar inanición, se implementa **envejecimiento** (aging), aumentando la prioridad de procesos que llevan mucho tiempo esperando.

Los sistemas modernos suelen combinar algoritmos, utilizando "prioridad con SRT" o "prioridad con Round Robin".

3. Hilos (Threads)

Concepto de hilo

Un **hilo** (thread) es la unidad de ejecución más pequeña que el sistema operativo puede programar y ejecutar de forma independiente. Representa una secuencia de instrucciones dentro de un proceso.

Características de los hilos:

- Cada hilo tiene su propio contador de programa, pila y datos locales
- Los hilos de un mismo proceso comparten el espacio de memoria, datos globales y recursos del sistema

- Permiten la ejecución concurrente de múltiples tareas dentro de una aplicación

Diferencias entre procesos e hilos
Procesos: Son instancias independientes de un programa con su propio espacio de memoria.

- La comunicación entre procesos es compleja y requiere mecanismos específicos (IPC)
- Cada proceso tiene un nivel de aislamiento alto
- No están sujetos al GIL (Global Interpreter Lock) en Python

Hilos:

- Pertenecen a un proceso y comparten su memoria
- La comunicación entre hilos es más sencilla mediante memoria compartida
- Menor sobrecarga de creación y cambio de contexto que los procesos
- En Python (CPython), están sujetos al GIL, limitando la ejecución paralela real

Los hilos pueden encontrarse en estados similares a los procesos: nuevo, listo, en ejecución, bloqueado y terminado. El sistema operativo gestiona la planificación de hilos de manera similar a los procesos.

4. Desmitificando la Concurrencia, el Paralelismo y la Distribución: Una Guía para Principiantes

Introducción: Entendiendo el Flujo de Tareas

En el mundo del desarrollo de software, los términos concurrencia, paralelismo y distribución son fundamentales, pero a menudo se confunden. El objetivo de esta guía es aclarar las diferencias clave entre estos tres paradigmas de programación, especialmente para quienes se inician en el tema. Para entenderlos de forma intuitiva, imaginemos a un cocinero en una cocina:

- Concurrencia: Un solo cocinero prepara varios platos a la vez. Empieza a cortar verduras, luego pone una olla al fuego y, mientras se calienta, remueve una salsa. No hace todo al mismo tiempo, pero al alternar tareas rápidamente, da la *ilusión* de que todos los platos avanzan simultáneamente.

- Paralelismo: Ahora tenemos una cocina con varios cocineros (o un cocinero con múltiples brazos). Un cocinero corta las verduras *mientras* otro prepara la salsa, *al mismo tiempo*. Las tareas se ejecutan de forma verdaderamente simultánea, acelerando el proceso.

- Distribución: Pensemos en una gran cadena de restaurantes. Una cocina central prepara las bases de las salsas, otra se especializa en hornear el pan y una tercera ensambla los platos finales. Son equipos independientes, en lugares distintos, que se comunican a través de una red para completar un objetivo común a gran escala.

Con esta analogía en mente, profundicemos en los detalles técnicos de cada concepto.

Programación Concurrente: La Ilusión de Hacer Todo a la Vez

La programación concurrente es un modelo donde múltiples tareas parecen progresar al mismo tiempo, pero en realidad, **las tareas se intercalan en el tiempo**. Un único procesador cambia entre ellas tan rápidamente que para el usuario parece que se ejecutan de forma simultánea. Es una gestión inteligente del tiempo para mantener una aplicación fluida y receptiva.

Sus características más importantes son:

- **Hardware Requerido:** Funciona perfectamente sobre un procesador/núcleo.
    - **Beneficio:** No requiere hardware especializado. Es un paradigma enfocado en la estructura del software para gestionar tareas que, por naturaleza, implican esperas.
- **Caso de Uso Principal:** Es ideal para tareas I/O bound (vinculadas a Entrada/Salida).
    - **Beneficio:** Mejora enormemente la ***capacidad de respuesta*** de las aplicaciones. Por ejemplo, mientras una tarea espera la respuesta de una red o la lectura de un archivo, el procesador puede dedicarse a otra, como actualizar la interfaz de usuario, evitando que el programa se "congele".

La concurrencia nos da una sensación de simultaneidad, pero para lograr una ejecución verdaderamente simultánea, necesitamos más potencia de hardware.

Programación Paralela: El Poder de Hacer Varias Cosas Realmente al Mismo Tiempo

A diferencia de la concurrencia, la programación paralela implica que las tareas se ejecutan **simultáneamente en diferentes núcleos del procesador**. Aquí no hay ilusión; el trabajo se divide y se procesa en el mismo instante de tiempo, aprovechando al máximo el hardware moderno.

Sus características más importantes son:

- **Hardware Requerido:** Exige procesadores multinúcleo.
    - **Beneficio:** Permite que una aplicación aproveche toda la capacidad de cómputo de las CPUs modernas para acelerar la resolución de problemas complejos.
- **Caso de Uso Principal:** Es la mejor opción para tareas intensivas de CPU (CPU bound).
    - **Beneficio:** Reduce realmente el tiempo de ejecución. Tareas como el renderizado de video, los cálculos científicos o el análisis de grandes conjuntos de datos se completan mucho más rápido al dividirse entre varios núcleos que trabajan en paralelo.

El paralelismo nos permite resolver problemas más rápido en una sola máquina. Pero, ¿qué sucede cuando una sola máquina no es suficiente?

Programación Distribuida: El Equipo de Máquinas Trabajando Juntas

La programación distribuida lleva el concepto un paso más allá: en lugar de usar múltiples núcleos en una máquina, se **distribuyen las tareas entre múltiples equipos o nodos conectados en red**. Cada máquina es un componente de un sistema mayor y colabora para lograr un objetivo común.

Sus características más importantes son:

- **Hardware Requerido:** Se necesitan múltiples equipos interconectados.
- **Beneficio Principal:** Permite escalar horizontalmente para gestionar cargas de trabajo masivas, añadiendo más máquinas para aumentar la capacidad y la resiliencia.
- **Desafíos Clave:** Introduce complejidades adicionales como la latencia de red (comunicación más lenta que dentro de una misma máquina) y la tolerancia a fallos (el sistema debe ser diseñado para sobrevivir a la caída de nodos individuales).
- **Caso de Uso Principal:** Es fundamental para gestionar grandes volúmenes de datos y sistemas a gran escala. Piensa en los motores de búsqueda, las redes sociales o los servicios en la nube. Ninguna máquina individual podría manejar esa carga; se necesita un ejército de computadoras trabajando en conjunto.

Ahora que hemos explorado cada paradigma por separado, una tabla comparativa nos ayudará a consolidar sus diferencias.

Tabla Comparativa: Concurrente vs. Paralela vs. Distribuida

Esta tabla resume las distinciones clave que hemos discutido, ofreciendo una referencia rápida y clara.

Aspecto	Concurrente	Paralela	Distribuida
Hardware	Un procesador/núcleo	Múltiples núcleos	Múltiples equipos
Ejecución	Intercalada	Simultánea	Simultánea en red
Casos de uso	Tareas I/O bound (mejora de respuesta)	Tareas CPU bound (reducción de tiempo)	Grandes volúmenes de datos (escalado)

Conclusión: ¿Cuál Deberías Elegir?

La diferencia fundamental radica en **cómo se gestionan y ejecutan las tareas**: la concurrencia *gestiona* múltiples tareas a la vez en un solo núcleo, el paralelismo *ejecuta* múltiples tareas a la vez en varios núcleos, y la distribución *coordina* múltiples tareas a la vez en varias máquinas.

En última instancia, la elección correcta no depende de que un paradigma sea intrínsecamente "mejor" que otro. **La decisión se basa en la naturaleza del problema a resolver** y los **recursos de hardware disponibles**.

Por ejemplo, la concurrencia es ideal para **aplicaciones interactivas**, el paralelismo para **cálculos intensivos** y la distribución para sistemas que necesitan una escala masiva y alta disponibilidad. Cada uno es una herramienta poderosa con un propósito específico en el vasto campo de la informática.

5. Comunicación entre Procesos

La **comunicación entre procesos** (Inter-Process Communication - IPC) es el conjunto de mecanismos que permiten a procesos separados intercambiar información y coordinarse.

Mecanismos basados en mensajes

- Pipes (tuberías): Canal unidireccional que permite que un proceso escriba datos que otroproceso lee. Los datos fluyen en un buffer compartido.

- Named Pipes (FIFOs): Similar a los pipes pero con nombre en el sistema de archivos,permitiendo comunicación entre procesos no relacionados.

- Sockets: Permiten comunicación bidireccional entre procesos, incluso en diferentes máquinas.Existen sockets de dominio Unix (local) y sockets de red (TCP, UDP).

- Colas de mensajes (Message Queues): Almacenan mensajes en una cola que los procesospueden enviar y recibir.

Sincronización entre Procesos

La sincronización es esencial cuando procesos o hilos acceden a recursos compartidos para evitar condiciones de carrera e inconsistencias de datos.

Problemas de sincronización

- Condición de carrera (Race Condition): Ocurre cuando múltiples hilos/procesos modificansimultáneamente un recurso compartido, generando resultados impredecibles.

- Sección crítica: Porción de código donde se accede a recursos compartidos y que debeejecutarse de forma mutuamente exclusiva.

- Interbloqueo (Deadlock): Situación donde dos o más procesos esperan indefinidamente porrecursos que otros procesos mantienen bloqueados.

- Inanición (Starvation): Un proceso no puede acceder a un recurso porque otros procesos lomonopolizan.

6. Gestión de procesos en sistemas operativos

Los sistemas operativos proporcionan herramientas para crear, monitorizar, controlar y terminar procesos.

Herramientas de monitorización en Linux/Unix

- ps (Process Status): Muestra información sobre los procesos en ejecución. Permite ver PID,usuario, uso de CPU y memoria, tiempo de ejecución, etc.
- top: Monitor interactivo en tiempo real que muestra los procesos activos ordenados por uso derecursos. Se actualiza automáticamente.
- htop: Versión mejorada de top con interfaz más intuitiva y visual. Permite navegación con ratón, ordenación interactiva y gestión de procesos.
- pidstat: Proporciona estadísticas detalladas sobre procesos específicos.

- **nmon, vtop, bashtop, gtop**: Herramientas alternativas de monitorización con diferentes interfaces y características

7. Programación de aplicaciones multiproceso

El **multiprocesamiento** se refiere a un sistema que posee más de dos procesadores, y se utiliza para aumentar la ***velocidad*** y la ***potencia*** informática del sistema al añadir CPUs.

El multiprocesamiento permite la **ejecución paralela** de múltiples tareas, lo cual es fundamental para el escalamiento de sistemas de software.

Características y Conceptos Fundamentales

1. **Paralelismo y Ejecución:** En el multiprocesamiento, se ejecutan múltiples tareas de forma **simultánea** en diferentes núcleos del procesador. Esto es ideal para tareas intensivas de CPU (*CPU bound*)[1]. Un sistema multiprocesamiento permite ejecutar múltiples procesos simultáneamente.
2. **Rendimiento y Escalabilidad:** El multiprocesamiento ayuda a aumentar la potencia informática y a realizar más trabajo en un período más corto. Puede mejorar la **fiabilidad** del sistema y el **rendimiento** al descomponer un programa en tareas ejecutables paralelas. La reescritura de código para que se ejecute en múltiples procesos en lugar de uno solo puede reducir **el tiempo de procesamiento** y la **latencia**.
3. **Clasificación:** Los sistemas de multiprocesamiento se clasifican según la forma en que está organizada su memoria, y pueden ser **simétricos o asimétricos**.
4. **Procesos:** Un proceso es un **programa en ejecución** que incluye el código, los datos, el estado de los registros del procesador y recursos asignados por el sistema operativo. Cada proceso tiene un identificador único (**PID**). En la programación multiproceso, la unidad de trabajo es el proceso, y su creación es **lenta**.

Ventajas y Desventajas

Ventajas del Multiprocesamiento:

- Permite aprovechar múltiples CPUs y núcleos.
- Ayuda a **evitar las limitaciones del GIL** para CPython.
- Permite realizar el trabajo en un período más corto.

[1] Las **tareas intensivas de CPU** (o *CPU bound*) son aquellas que están limitadas principalmente por la velocidad del procesador y la cantidad de cálculo que realizan, requiriendo un uso intensivo de la Unidad Central de Procesamiento. Estas tareas implican una gran cantidad de **cálculos pesados** como operaciones de código de bytes, cálculo de números y similares

- Los sistemas multiprocesamiento **ahorran dinero** en comparación con los sistemas de un solo procesador al compartir periféricos y fuentes de alimentación.
- El **código suele ser sencillo**.
- Los procesos secundarios son en su mayoría interrumpibles o eliminables.

Desventajas del Multiprocesamiento:

- La Comunicación entre Procesos (IPC) es bastante complicada y requiere más gastos generales.
- Tiene una **mayor huella de memoria**.

Multiprocesamiento vs. Multihilo

El multiprocesamiento y el multihilo son **dos técnicas de ejecución concurrente**, con diferencias clave:

Parámetro	Multiprocesamiento	Multihilo
Recursos	Utiliza **más de dos** procesadores.	Utiliza múltiples **segmentos de código** (hilos) dentro de un solo proceso.
Memoria	Asigna memoria y recursos separados para cada proceso o programa, proporcionando un alto nivel de **aislamiento**.	Los hilos comparten la misma memoria y recursos del proceso.
Creación	La creación de un proceso es **lenta** y consume **recursos**.	La creación de un hilo es económica en tiempo y recursos, y son livianos.
Comunicación	La **comunicación** entre procesos (IPC) es **compleja** y requiere mecanismos específicos, lo que implica más sobrecarga. Se basa en *decapar* (*pickling*) objetos en la memoria para enviarlos a otros procesos.	La comunicación entre hilos es más sencilla a través de la memoria compartida. El multihilo evita el *decapado*.
Python y GIL	**Elimina las limitaciones del GIL** (*Global Interpreter Lock*) en CPython, permitiendo el uso eficiente de múltiples núcleos.	Está sujeto al GIL en CPython, lo que limita la ejecución paralela real.

Como puedes ver hay diferencias importantes entre un programa **multiproceso** y un programa **multihilo** en Python. La diferencia principal radica en cómo se aprovechan los recursos del sistema y en el tipo de tareas para las que cada modelo es más eficiente.

Diferencias clave

- Multiproceso (multiprocessing):

  - Crea varios procesos independientes, cada uno con su propio espacio de memoria.
  - Aprovecha varios núcleos de CPU y permite "paralelismo real".
  - No está limitado por el GIL (Global Interpreter Lock) de Python.
  - Ideal para tareas intensivas en CPU.
- Multihilo (threading):
  - Crea varios hilos dentro del mismo proceso y espacio de memoria.
  - Todos los hilos comparten los mismos recursos.
  - Limitado por el GIL, por lo que solo un hilo ejecuta código Python puro a la vez.
  - Adecuado para tareas de entrada/salida (E/S), como leer archivos o comunicaciones en red, pero no aprovecha varios núcleos en cálculos intensivos.

Ejemplo de Multiproceso (multiprocessing)

```python
from multiprocessing import Process
import time

def tarea(nombre):
    print(f"Proceso {nombre} inicia")
    time.sleep(2)
    print(f"Proceso {nombre} termina")

procesos = []
for i in range(3):
    p = Process(target=tarea, args=(i,))
    procesos.append(p)
    p.start()

for p in procesos:
    p.join()

print("Todos los procesos han terminado")
```

Cada proceso se ejecuta de forma independiente y puede correr en diferentes núcleos. Ideal para operaciones que requieren mucho CPU.

Ejemplo de Multihilo (threading)

```python
import threading
import time

def tarea(nombre):
    print(f"Hilo {nombre} inicia")
    time.sleep(2)
    print(f"Hilo {nombre} termina")

hilos = []
for i in range(3):
    hilo = threading.Thread(target=tarea, args=(i,))
    hilos.append(hilo)
    hilo.start()

for hilo in hilos:
    hilo.join()

print("Todos los hilos han terminado")
```

Todos los hilos comparten memoria y recursos. Si las tareas fueran intensivas en CPU, no se tendría paralelismo real por el GIL.

Cuándo usar cada uno

- Usa **multihilo** si tu tarea es de **E/S intensiva** y necesita compartir datos fácilmente entre hilos.
- Usa **multiproceso** cuando tu tarea es **intensiva en CPU** o cuando necesitas aprovechar más de un núcleo realmente.

En resumen: aunque ambos permiten ejecutar código de manera concurrente, solo el multiproceso en Python permite verdadero paralelismo para tareas de cálculo pesado. El multihilo en Python es útil para manejar muchas esperas (E/S), pero no para acelerar cálculos paralelos en varios núcleos.