

Taller Práctico: Gestión, Planificación y Sincronización de Procesos

Objetivo: Reforzar los conceptos teóricos de asignación de CPU y analizar el comportamiento de los mecanismos de sincronización a nivel de código en el sistema operativo.

Parte 1: Planificación de Procesos

Con base en la siguiente tabla de procesos, dibuja el **Diagrama de Gantt** y calcula el **Tiempo Promedio de Espera (Tep)** y el **Tiempo Promedio de Retorno (Trp)** para los siguientes algoritmos de planificación:

Proceso	Tiempo de Llegada	Ráfaga de CPU (Duración)	Prioridad (1 = Mayor)
P1	0 ms	6 ms	3
P2	2 ms	3 ms	1
P3	4 ms	1 ms	4
P4	5 ms	4 ms	2

1. Algoritmo FCFS (First Come First Served):

- Dibuja el Diagrama de Gantt asumiendo que se atienden estrictamente por orden de llegada.
- ¿Se observa el "Efecto Convoy" en este escenario? Justifica tu respuesta.

2. Algoritmo Round Robin (Quantum = 3 ms):

- Dibuja el Diagrama de Gantt asumiendo que la lista se trata como una cola FIFO y el algoritmo es siempre expulsivo al terminar el quantum.
- Calcula el tiempo de retorno (regreso) exclusivo para el proceso P2.

Parte 2: Sincronización y Problemas Clásicos

Responde de forma concisa basándote en la teoría de procesos:

3. Condición de Carrera:

- Explica con tus propias palabras qué es una condición de carrera (*race condition*) y por qué suele resultar en la corrupción de los datos compartidos (como el saldo de una cuenta).

4. Identificación de Problemas: Relaciona la situación con el problema clásico de sincronización que le corresponde (Cena de los Filósofos, Lectores/Escritores, Productor/Consumidor):

- *Situación A:* Varios procesos quieren revisar un archivo PDF, pero solo uno tiene permiso para editarlo. Si alguien edita, nadie puede leer.
- *Situación B:* Un proceso descarga paquetes de internet y los pone en la memoria, mientras otro los saca para reproducir un video. Si la memoria se llena, el que descarga debe pausarse.

Parte 3: Laboratorio de Código en C (Análisis en WSL)

Con base en las prácticas realizadas en la terminal de Ubuntu/WSL, analiza los siguientes fragmentos de código:

5. Análisis de Semáforos:

En nuestra práctica, utilizamos la librería `<semaphore.h>` y la instrucción `sem_init(&semaforo, 0, 2)`.

- ¿Qué representan las operaciones atómicas `wait` y `signal` (`sem_wait` y `sem_post` en C) sobre este semáforo contador?
- ¿Qué ocurriría (y cómo se llama a ese estado) si un programador olvida escribir la instrucción `sem_post(&semaforo)` al final de la sección crítica y todos los procesos siguen pidiendo entrar?

6. Monitores vs Paso de Mensajes:

- En el código del **Cajero Automático (Monitor)**, utilizamos la función `pthread_cond_wait(&condicion, &mutex)` cuando el saldo era insuficiente. ¿Qué hace internamente esta función con la "puerta" (`mutex`) de la cabina blindada para permitir que otro hilo entre a depositar?

- Para compilar el Monitor y los Semáforos fue obligatorio usar la bandera `gcc -pthread`. Sin embargo, para el **Paso de Mensajes (Pipes)** con `fork()`, no fue necesario. Explica por qué el Paso de Mensajes no necesita herramientas de exclusión mutua para proteger variables compartidas.