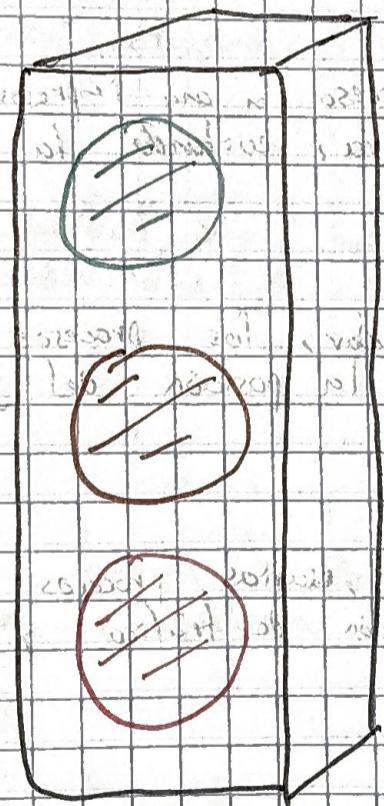


13/10/2025

Universidad Politécnica De  
Pachuca

Tecnología En Software  
Programación Concorrente - STW-0701

2do Parcial Apunte 1



## ► Coordinación y Colaboración

Los procesos pueden coordinar sus acciones para lograr objetivos comunes. Esto puede implicar dividir una tarea en subprocesos que trabajen juntos para cumplirla. La colaboración implica la comunicación y el intercambio de resultados parciales para lograr un objetivo conjunto.

## ► Exclusión Mutua

La exclusión mutua es un concepto clave en la programación concurrente para evitar conflictos y garantizar que un recurso compartido solo sea accedido por un proceso a la vez. Mecanismos como los semáforos binarios y los bloqueos aseguran que solo un proceso pueda acceder al recurso en un momento dado.

## EJEMPLOS ILUSTRATIVOS

### ► Comunicación Directa

Ejemplo: Dos procesos colaboran en la ~~directa~~ edición de un documento compartido. Utilizan una cola para enviar y recibir actualizaciones en tiempo real.

### ► Comunicación Indirecta

Ejemplo: Varios procesos comparten acceso a una impresora. Utilizan un semáforo para controlar el acceso a la impresora, evitando la impresión simultánea.

### ► Sincronización

Ejemplo: En un videojuego multijugador, los procesos que representan a los jugadores esperan su turno para actualizar la posición del jugador en un bucle sincronizado.

## ► Coordinación y Colaboración

Ejemplo: En un sistema de navegación, varios procesos cooperan para encontrar la ruta más rápida. Comparten información de tráfico y ajustan las rutas en función de los datos recibidos.

### ► Exclusión Mutua

Ejemplo: En un sistema de reservas en línea, varios procesos intentan reservar con mismo asiento. Utilizan un bloqueo para asegurarse de que solo un proceso pueda reservar el asiento.

## Problemas y Patologías

### ► Condiciones de Carrera

Las condiciones de carrera ocurren cuando múltiples procesos intentan acceder o modificar recursos compartidos al mismo tiempo, lo que puede llevar a resultados inconsistentes.

### ► Bloques

Los bloques ocurren cuando un proceso queda esperando indefinidamente por un recurso que está bloqueado por otro proceso, provocando una paralización del sistema.

### ► Dead Lock

Un deadlock ocurre cuando dos o más procesos se bloquean mutuamente mientras esperan que el otro libere un recurso que necesitan.

### ► Starvation

La starvation ocurre cuando un proceso no puede obtener acceso a recursos necesarios debido a la prioridad otorgada a otros procesos.

### ► Inanición (Live lock)

Un livelock es similar a un deadlock, pero los procesos no están bloqueados. Si no que están en un ciclo interminable de competición por liberación de recursos sin progresar.

### ► Problemas de Sincronización

Los problemas de sincronización ocurren cuando los procesos no se coordinan adecuadamente, lo que puede llevar a ejecuciones incorrectas.

### ► Falta de Cohesión de Datos

Procesos concurrentes pueden introducir inconsistencias si no se controla el acceso a los datos compartidos.

### ► Problemas de Rendimiento

La programación concurrente mal gestorada puede impactar el rendimiento debido a recursos compartidos y sincronización inefficientes.

## ► Problemas de Planificación

Planificar procesos de manera incorrecta puede llevar a un uso "ineficiente" de recursos y baja capacidad de respuesta.

## ► Dificultad de Depuración

Errores en programación concurrente pueden ser difíciles de reproducir y depurar debido a la naturaleza intercalada de la ejecución.

## ► Comunicación Excesiva

Comunicación constante entre procesos puede llevar a un exceso de intercambio de mensajes afectando el rendimiento.

## ► Problemas de Prioridad

Asignación incorrecta de prioridades puede afectar la justicia en el acceso a los recursos y ejecución de tareas.

## ► Race Conditions

Race conditions ocurren cuando el resultado depende del orden de ejecución, llevando a resultados inconsistentes.

## ► Escalabilidad Limitada

En algunos casos, la programación concurrente puede no escalar eficientemente al aumentar el número de procesos.

## Abordar los Problemas

Para abordar estos problemas, se deben aplicar técnicas de sincronización, coordinación y planificación adecuadas. Además, se debe tener un diseño cuidadoso de la arquitectura concurrente y utilizar herramientas y mecanismos apropiados.

# Semáforos

USIAD0

Un controlador de sincronización es un Semáforo. Semaphore proporciona acceso sincronizado a un número limitado de recursos para subprocesos.

Un semáforo se puede ver como una variable que representa cuantos recursos están disponible ahora. Por ejemplo, en estacionamiento en un centro comercial que funciona como un semáforo tiene varios espacios disponibles en un nivel determinado.

El valor del semáforo debe ser mayor o menor que los recursos disponibles. Las operaciones de adquirir y liberar están asociadas con el semáforo. El valor del semáforo se reduce cuando un subproceso "adquiere" uno de los recursos que se utilizan para la sincronización. El valor del semáforo aumenta cuando un subproceso "libera" uno de los recursos sincronizadores.

## Semáforos en Python

La implementación de python del concepto de semáforo utiliza una clase del módulo threading. Semáforo es el nombre de esta clase. sus funciones acquire() y release() están incluidas en la clase Semaphore, junto con un constructor de funciones.

Si el recuento de semáforos es mayor que cero, se utilizará la función acquire() para rebajar el recuento. De lo contrario se bloqueará hasta que el recuento sea menor que 0. Uno de los subprocesos que se ejecutan en el semáforo se ejecuta mediante el uso de la función release(), que también aumenta el cuenta del semáforo. Su sintaxis es la siguiente:

```
object_name = threading.Semaphore(count)
```

El objeto de la clase **Semáforo** se indica mediante el **nombre objeto** en la sintaxis anterior. El número de subprocesos a los que se permite acceder a la vez se especifica mediante el argumento **recuento** de la clase **Semáforo**. El valor predeterminado de este parámetro es 1. El valor del parámetro **count** se reduce en una cada vez que un subproceso utiliza la función **adquirir()**. El valor del parámetro **count** se incrementa en una cada vez que un hilo utiliza la función **release()**.

De acuerdo con esta afirmación, cada vez que llamemos al método **adquirir()**, el valor del parámetro **count** disminuirá; sin embargo, cuando llamamos a la función **release()**, el valor del parámetro **count** aumentará.

# CÓDIGO

20 octubre 2023

```
import threading
sema = threading.Semaphore(1)    ➡️ Librería para usar hilos en Python  
def test():
    print(f"Value Of Semaphore → {sema_value}") } Se Define la función teste
    sema.acquire() ➡️ Un hilo toma el semáforo Si es > 0 (como 1) lo regresa a 0
    print(f"acquired lock → {threading.current_thread().name}") } Asegura como cada
    print(f"Value Of Semaphore → {sema_value}") } garantiza de que el
    print(f"release lock → {threading.current_thread().name}") } hilo a la vez estará
    sema.release() ➡️ El hilo que tenía el semáforo lo libera, incrementando el contador.
    print(f"Value Of Semaphore → {sema_value}") ➡️ fines demostrativo mostrando
el semáforo interno cambia de 1 a 0 cuando es adquirido
```

t1 = threading.Thread(target=test) } Se crean tres objetos de tipo Thread donde
t2 = threading.Thread(target=test) } el target indica a cada hilo que la
t3 = threading.Thread(target=test) } función que debe ejecutar es test

t1.start() } Inician la ejecución de los tres hilos y comienza a correr
t2.start() } de forma CONCURRENTE y "compiten" por adquirir
t3.start() } el semáforo dentro de la función test

Los Subprocesos se sincronizan si count se establece en 1, como en el
código anterior. Si observamos la salida del código anterior, notaremos que
será el primer y segundo subproceso, luego el tercer subproceso tendrá acceso
al código entre "adquirir" y "Liberar".

## SALIDA

Value Of Semaphore  $\rightarrow 1$

Value Of Semaphore  $\rightarrow 1$

acquired lock  $\rightarrow$  Thread-1 (test)

Value Of Semaphore  $\rightarrow 0$

Value Of Semaphore  $\rightarrow 0$

release lock  $\rightarrow$  Thread-1 (test)

Value Of Semaphore  $\rightarrow 1$

acquired lock  $\rightarrow$  Thread-2 (test)

Value Of Semaphore  $\rightarrow 0$

release lock  $\rightarrow$  Thread-2 (test)

Value Of Semaphore  $\rightarrow 1$

acquired lock  $\rightarrow$  Thread-3 (test)

Value Of Semaphore  $\rightarrow 0$

release lock  $\rightarrow$  Thread-3 (test)

Value Of Semaphore  $\rightarrow 1$

Cuando se ejecutan el código, los tres hilos arrancan casi al mismo tiempo y todas intentar llamar a `Sema.acquire()`

Si embargo debido al semáforo sobre uno de ellos (el que llegue primero) logrará tomar el semáforo y ese primer hilo ejecutará el código dentro de la sección e imprimirá los mensajes mientras que los otros dos hilos se quedarán en espera.

Cuando el primer hilo termine y libere el semáforo el semáforo queda libre e inmediatamente uno de los dos hilos lo adquirirá y el proceso se repetirá hasta que TODOS los hilos hayan cumplido su ejecución.