GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

ceedcv CENTRE ESPECÍFIC D'EDUCACIÓ A DISTÀNCIA DE LA COMUNITAT VALENCIANA

Unidad 1.5

Programación multihilo

Índice

Sumario

Licencia

1. Sincronismo con semáforos

Los semáforos son una estructura de datos que se utiliza para controlar el acceso a recursos compartidos en un entorno multihilo. Un semáforo permite que un número máximo de hilos accedan a un recurso compartido simultáneamente. A diferencia de un bloqueo (lock), que permite que solo un hilo acceda a un recurso a la vez, los semáforos permiten un mayor grado de paralelismo al permitir que múltiples hilos accedan al recurso bajo ciertas restricciones.

1.1. Funcionamiento de los Semáforos

Un semáforo tiene un contador interno que rastrea cuántos hilos tienen permiso para acceder a un recurso compartido en un momento dado. Cuando un hilo intenta acceder al recurso, el semáforo verifica el contador. Si el contador es mayor que cero (indicando que hay permisos disponibles), el hilo obtiene acceso y el contador disminuye en uno. Si el contador es cero, el semáforo coloca al hilo en espera hasta que algún otro hilo libere un permiso.

Los semáforos se pueden utilizar de varias maneras, pero dos operaciones comunes son:

- El método **acquire()**: Si un hilo desea acceder al recurso compartido, realiza una operación acquire() en el semáforo. Si el contador es mayor que cero, el hilo obtiene acceso y el contador del semáforo disminuye. Si el contador es cero (el semáforo no lo deja pasar), el hilo se bloquea hasta que otro hilo realice una operación release()
- El método **release()**: Cuando un hilo ha terminado de utilizar el recurso compartido, realiza una operación release() en el semáforo, lo que incrementa el contador del semáfor. Esto libera un permiso y permite que otro hilo acceda al recurso.

1.2. Tipos de semáforos

Hay dos tipos principales de semáforos: binarios y contadores.

- **Semáforos Binarios**:

  Un semáforo binario tiene dos estados: abierto (`1`) o cerrado (`0`). A menudo se utiliza para controlar el acceso a un recurso compartido único. Si el semáforo está cerrado, los hilos que intentan acceder al recurso esperarán hasta que se abra.

  Funciona igual que el lock()
- **Semáforos Contadores**:

  Un semáforo con un rango de valores mayores a 1 se conoce como semáforo contador. Este tipo de semáforo se utiliza para controlar el acceso a varios recursos idénticos. Cada vez que un hilo accede a un recurso, el contador del semáforo se decrementa. Cuando el recurso se libera, el contador se incrementa. Si el contador llega a cero, los hilos esperan hasta que el recurso esté disponible.

1.3. Ejemplo con semáforos

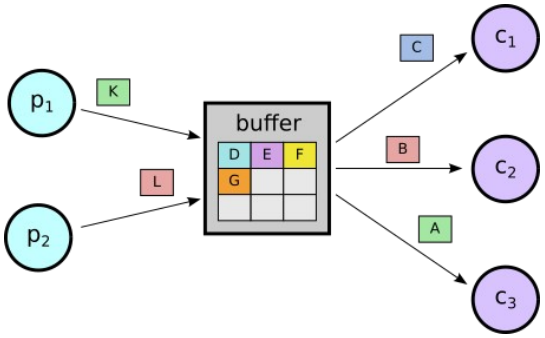
Supongamos que tienes una aplicación que simula una sala con un número limitado de asientos (10) y varias personas que quieren ingresar a la sala. Utilizaremos semáforos para controlar el acceso a la sala:

- El semáforo (**semaforo_sala**) se inicializa con un contador a 10, lo que significa que solo se permitirá la entrada a 10 personas simultáneamente.
- Cada hilo (**hilo_persona**) representa a una persona que quiere entrar a la sala.
- La función **persona_entra**: simula el proceso de entrada y salida de la sala.
  - Cuando una persona entra, adquiere un asiento utilizando **acquire()**.
  - Después de un tiempo simulado en la sala, la persona libera el asiento utilizando **release()**.

```python
import threading
import time

# Crear un semáforo con un contador inicial de asientos disponibles
semaforo_sala = threading.Semaphore(10)  # Supongamos que hay 10 asientos disponibles

def persona_entra(numero_persona):
    print(f"Persona {numero_persona} esperando para entrar a la sala.")

    # Intentar adquirir un asiento
    semaforo_sala.acquire()

    print(f"Persona {numero_persona} ha entrado a la sala.")
    time.sleep(2)  # Simular tiempo en la sala

    # Liberar el asiento al salir
    semaforo_sala.release()
    print(f"Persona {numero_persona} ha salido de la sala.")


hilos_personas = []

# Crear hilos que representan personas que quieren entrar a la sala
for i in range(15):
    hilo_persona = threading.Thread(target=persona_entra, args=(i,))
    hilos_personas.append(hilo_persona)
    hilo_persona.start()

# Esperar a que todos los hilos terminen
for hilo_persona in hilos_personas:
    hilo_persona.join()

print("La sala está vacía. Programa principal terminado.")
```

2. Sincronización entre producto-consumidor

La sincronización entre productor y consumidor es un problema clásico en la programación concurrente y multihilo. Este patrón se utiliza cuando hay un proceso de producción de datos (productor) y un proceso de consumo de datos (consumidor) que operan de forma concurrente. El objetivo es garantizar que el consumidor no intente consumir datos antes de que el productor los haya producido y que el productor no produzca datos adicionales si el consumidor aún no ha consumido los datos existentes.



Autor: Gracker - https://androidperformance.com/2019/12/15/Android-Systrace-Triple-Buffer/

Este patrón es útil en situaciones en las que los datos se generan en una velocidad diferente a la que se consumen, y es fundamental para evitar problemas como desbordamiento de memoria, bloqueos o condiciones de carrera. Aquí hay algunos conceptos clave en la sincronización entre productor y consumidor:

- **Buffer o Cola Compartida y Limitada**: Un buffer o cola es una zona de almacenamiento temporal que puede ser accedida por múltiples hilos. En el contexto de la programación concurrente, un buffer limitado es aquel que tiene una capacidad definida y no puede almacenar datos más allá de esa capacidad.
- **Productor y Consumidor:**

  En el patrón de sincronización entre productor y consumidor, dos tipos de hilos interactúan: el productor y el consumidor.

  - **El Productor** es responsable de generar datos y colocarlos en el buffer o cola. Por ejemplo, podría ser un hilo que descarga datos de la red y los almacena temporalmente en el buffer.
  - **El Consumidor** es responsable de tomar los datos del buffer o cola, utilizarlos para alguna tarea y repetir este proceso hasta que la tarea asignada esté completa. Por ejemplo, podría ser un hilo que lee datos descargados de Internet y los almacena en una base de datos.
- **Bloqueo y Espera Activa**: Los productores y consumidores deben utilizar mecanismos de sincronización para asegurarse de que están coordinados. Esto se logra típicamente con bloqueos o semáforos. Cuando el buffer está lleno (para productores) o vacío (para consumidores), los hilos deben esperar hasta que la condición sea adecuada. La "espera activa" implica comprobar de forma repetida la condición hasta que sea verdadera.

- **Sincronización Bidireccional**: La sincronización debe ser bidireccional. El productor debe esperar si el buffer está lleno, y el consumidor debe esperar si el buffer está vacío. Cuando un productor coloca datos en el buffer, notifica al consumidor que hay datos disponibles, y cuando el consumidor retira datos, notifica al productor que hay espacio disponible.
- **Balance y Rendimiento**: En un sistema bien diseñado, se debe lograr un equilibrio entre la velocidad de producción y la velocidad de consumo para evitar desbordamiento o ineficiencia. Esto puede requerir ajustar el tamaño del buffer y la velocidad de producción/consumo según las necesidades del sistema.

Uno de los desafíos clave en la sincronización entre productor y consumidor es lidiar con la diferencia en la velocidad de operación entre los hilos. Puede ocurrir que el Productor sea más rápido en la generación de datos que el Consumidor en la tarea de procesamiento, o viceversa.

2.1. Solución: Uso de Semáforos

Para abordar estos desafíos, se utilizan semáforos, que son herramientas de sincronización entre hilos. Se emplean tres semáforos para resolver los problemas en el patrón de productor-consumidor:

- **Vacío**: Este semáforo controla cuántos espacios vacíos hay en el buffer. Inicialmente, su valor es igual a la capacidad máxima del buffer. Cuando el Productor intenta agregar datos, adquiere este semáforo y disminuye su valor en 1. Si el valor es 0, indica que el buffer está lleno, y el Productor debe esperar hasta que haya espacio.
- **Lleno**: Este semáforo controla cuántos espacios están llenos en el buffer. Inicialmente, su valor es 0. Cuando el Consumidor intenta tomar datos, adquiere este semáforo. Si el valor es 0, significa que el buffer está vacío, y el Consumidor debe esperar hasta que haya datos.
- **Exclusión mutua**: Este semáforo maneja la condición de carrera y garantiza que solo un hilo a la vez puede operar en el buffer. Inicialmente, su valor es 1. Antes de trabajar en el buffer, tanto el Productor como el Consumidor intentan adquirir este semáforo. Si su valor es 0, indica que el otro hilo está trabajando en el buffer, y el hilo en espera se bloqueará hasta que se libere el semáforo.

  Se comporta igual que el **Lock()**.

2.2. Ejemplos sincronización entre productor y consumidor

Este tipo de sincronización entre productor y consumidor se utiliza en los siguientes escenarios, entre otros:

- **Almacenamiento en Memoria Intermedia de Archivos Multimedia**: En aplicaciones de reproducción de medios, como reproductores de audio y video, los productores leen datos del archivo multimedia y los colocan en un búfer en memoria. Los consumidores, como los decodificadores de audio y video, toman esos datos del búfer y los reproducen. La sincronización garantiza una reproducción fluida.
- **Procesamiento de Pedidos en una Tienda en Línea**: En un sistema de comercio electrónico, los clientes realizan pedidos que se procesan de forma asincrónica. Los productores son los pedidos que llegan, y los consumidores son los sistemas de procesamiento que verifican, empaquetan y envían los productos. La sincronización asegura que los pedidos se manejen correctamente.
- **Pool de Hilos en un Servidor Web**: En un servidor web, se puede utilizar un pool de hilos para manejar las solicitudes de los clientes. Los productores son las solicitudes entrantes, y los consumidores son los hilos que manejan esas solicitudes. La sincronización garantiza que las solicitudes se manejen de manera ordenada y eficiente.
- **Sistema de Monitoreo de Sensores IoT**: En un sistema de monitoreo de sensores IoT (Internet de las cosas), los sensores producen datos constantemente, y un sistema de procesamiento (consumidor) los procesa. La sincronización es necesaria para que los datos se procesen a medida que llegan y no se pierdan ni se sobrecargue el sistema.

Estos ejemplos ilustran cómo la sincronización entre productor y consumidor es esencial en una variedad de aplicaciones en las que se manejan datos o tareas de manera concurrente. La sincronización garantiza que los recursos se utilicen eficientemente y que los procesos se ejecuten de manera ordenada.

Vamos a poner un ejemplo de un problema de productor-consumidor en Python utilizando semáforos contadores. En este caso, vamos a usar semáforos para controlar que no haya más de 10 productos en el buffer. El productor añadirá 20 productos y el consumidor recogerá 20 productos:

```python
import threading
import time
import random
# Variables globales
BUF_SIZE = 10
buffer = [None]*BUF_SIZE

# Semáforos
semaforo_productor = threading.Semaphore(BUF_SIZE) # Controla la cantidad máxima de productos en el buffer
semaforo_consumidor = threading.Semaphore(0) # Inicialmente, el buffer está vacío

# Hilo productor
class Productor(threading.Thread):
    def run(self):
        global BUF_SIZE, buffer
        global semaforo_productor, semaforo_consumidor
        indice_entrada = 0
        for i in range(20):
            time.sleep(random.uniform(0.1, 0.5)) # Simula la producción de un elemento
            item = f"Item {i+1}" #Producido el elemento i
            semaforo_productor.acquire()
            buffer[indice_entrada] = item
            indice_entrada = (indice_entrada +1)%BUF_SIZE
            print(f"El producto ha añadido el elemeno {item}")
            semaforo_consumidor.release()

# Hilo consumidos
class Consumidor(threading.Thread):
    def run(self):
        global BUF_SIZE, buffer
        global semaforo_productor, semaforo_consumidor
        indice_salida = 0
        for i in range(20):
            time.sleep(random.uniform(0.1, 0.5)) # Simula el tiempo de procesamiento
            semaforo_consumidor.acquire()
            producto = buffer[indice_salida]
            indice_salida = (indice_salida +1)%BUF_SIZE
            print(f"El consumidr ha sacado el producto {producto}")
            semaforo_productor.release()
```

```
if __name__ == "__main__":
    # Creamos hilos para el productor y el consumidor
    hilo_productor = Productor()
    hilo_consumidor = Consumidor()

    # Iniciamos los hilos
    hilo_productor.start()
    hilo_consumidor.start()

    # Esperamos a que ambos hilos terminen
    hilo_productor.join()
    hilo_consumidor.join()
    print("Todas las operaciones han finalizado")
```

Explicación:

¿Porqué el productor hace primero un semaforo_productor.acquire() y lueg un semaforo_consumidor.release()?

semaforo_productor.acquire(): Cuando un productor quiere agregar un producto al buffer, primero intenta adquirir el semáforo semaforo_productor. Este semáforo se inicializa con un valor de 10, lo que significa que hay 10 permisos disponibles para que los productores coloquen hasta 10 productos en el buffer. Mientras hayan espacios disponibles para dejar productos (es decir, el buffer no está lleno), el productor podrá meter productos en el buffer. Si ha llenado el buffer con 10 productos el semáforo impide que se pongan más productos y evita así un oveflow.

semaforo_consumidor.release(): Inicialmente, este semáforo tiene un valor de 0, lo que significa que no hay productos disponibles para consumir (inicialmente el buffer está vacío, no tiene productos). Una vez que el productor ha añadido un producto libera un permiso en el semáforo consumidor. De esta forma indica al consumidor ya puede recogerlo.

¿Porqué el productor hace primero un semaforo_consumidor.acquire() y lueg un semaforo_productor.release()?

semaforo_consumidor.acquire(): Primero el consumidor intenta coger un producto, como el semáforo está a 0 espacios, tendrá que esperar. En el momento en que un productor introduzca un producto, liberará un espacio y el consumido entrará a pillarlo.

semaforo_productor..release(): Cuando un consumidor ha tomado un producto del buffer, libera un permiso en el semáforo productor. De esta forma indica al productor que vuelve a haber espacio en el buffer y puede rellenarlo.

3. Barreras

En Python, las barreras son estructuras de sincronización que se utilizan para coordinar la ejecución de múltiples hilos o procesos en un punto específico de un programa. Las barreras se utilizan para asegurarse de que un grupo de hilos alcanza un punto de sincronización antes de continuar la ejecución. En otras palabras, todos los hilos deben "encontrarse" en la barrera antes de poder continuar.

La implementación de barreras en Python se proporciona a través del módulo threading en la forma de la clase Barrier.

Creación de una Barrera: Para usar una barrera, primero debes crear una instancia de la clase Barrier. Puedes especificar el número de hilos que deben llegar a la barrera antes de que todos puedan continuar. Por ejemplo, si deseas que cinco hilos se sincronicen en una barrera, puedes crear una barrera con el argumento n igual a 5.

```
from threading import Barrier

# Crear una barrera para sincronizar 5 hilos
barrera = Barrier(5)
```

Espera en la Barrera: Cada hilo que debe sincronizarse en la barrera ejecuta el método wait() de la instancia de la barrera. Cuando un hilo llama a **wait()**, espera a que el número requerido de hilos (especificado al crear la barrera) llegue a la barrera.

Espera hasta que todos los Hilos lleguen: La barrera bloqueará cada hilo que llama a wait() hasta que se alcance el número especificado de hilos en la barrera. Una vez que se alcanza ese número, todos los hilos son liberados simultáneamente y pueden continuar su ejecución.

Las barreras son útiles en situaciones donde necesitas que varios hilos o procesos alcancen un punto de sincronización antes de continuar, como en algoritmos de paralelismo o cuando deseas esperar a que todos los trabajadores completen una tarea antes de procesar los resultados.

3.1. Diferencia entre barrera y join()

En una barrera, no hay un orden específico en el que los 5 hilos lleguen a la barrera. Los hilos pueden llegar y quedarse en espera en la barrera sin un orden predefinido. Una vez que se alcanza el número especificado de hilos en la barrera, todos los hilos pueden continuar simultáneamente, lo que significa que no hay un orden forzado de llegada.

Por otro lado, cuando utilizas join() en los 5 hilos, estás esperando a que cada hilo individual termine en el orden en el que se llama a join(). Esto significa que el primer join() esperará a que el primer hilo termine, el segundo join() esperará al segundo hilo, y así sucesivamente. Por lo tanto, join() impone un **orden secuencial** en el que los hilos se esperan a que finalicen, basado en el orden en el que se aplican los join().

En resumen, mientras que en una barrera no existe un orden específico de llegada y todos los hilos pueden continuar simultáneamente, el uso de join() impone un orden secuencial en el que los hilos se esperan a que finalicen, en función del orden en el que se aplican los join().

3.2. Ejemplo de código con barreras

```python
import threading
import time

# Función para simular la preparación de un equipo
def preparar_equipo(equipo_id, barrera_preparacion):
    print(f'Equipo {equipo_id} está preparando el coche')
    time.sleep(3)  # Simulamos la preparación
    barrera_preparacion.wait()  # Se espera a que todos los coches estén en la parrilla de salida

# Función para simular la vuelta de formación
def vuelta_de_formacion(piloto_id, barrera_formacion):
    print(f'El piloto del equipo {piloto_id} está preparado para hacer la vuelta de formación')
    time.sleep(2)  # Simulamos la vuelta de formación
    barrera_formacion.wait()  # Todos los coches deben acabar la vuela de formación
    print(f'El piloto {piloto_id} ha terminado de calentar sus frenos y neumáticos')

if __name__ == "__main__":
    num_equipos = 5

    # Crear barreras para sincronización
    barrera_preparacion = threading.Barrier(num_equipos+1)  # 5 equipos + el programa principal espera en la barrera
    barrera_formacion = threading.Barrier(num_equipos)  # Conductores de los 5 eqipos en la parrilla de salida

    # Crear e iniciar hilos de equipos
    equipos = []
    pilotos = []

    for i in range(num_equipos):
        hilo = threading.Thread(target=preparar_equipo, args=(i + 1, barrera_preparacion))
        equipos.append(hilo)
        hilo.start()

    # El programa principal también espera en la barrera
    barrera_preparacion.wait()

    print("Los equipos han acabado de preparar los coches")

    for i in range(num_equipos):
        hilo = threading.Thread(target=vuelta_de_formacion, args=(i + 1, barrera_formacion))
        pilotos.append(hilo)
        hilo.start()

    # Hacemos el join() para esperar a que todos los hilos imprman el print que hay después de la barrera
    for hilo in pilotos:
        hilo.join()


print('¡La carrera ha comenzado!')
```

4. Timer

La clase `Timer` en Python es parte del módulo `threading` y se utiliza para ejecutar una función después de un cierto tiempo de retraso. Es especialmente útil cuando deseas realizar una tarea después de un período de espera específico o realizar una operación de manera periódica.

Creación de un Timer:

```python
from threading import Timer

def tarea_programada():
    print("¡Tarea programada realizada!")

# Crear un Timer con un retraso de 5 segundos y la función a ejecutar
mi_timer = Timer(5, tarea_programada)

# Iniciar el Timer
mi_timer.start()
```

En este ejemplo, se crea un objeto `Timer` que ejecutará la función `tarea_programada` después de un retraso de 5 segundos. Es importante destacar que el hilo principal no se bloquea durante el tiempo de espera; puede continuar ejecutándose otras tareas mientras el Timer está en cuenta regresiva.

4.1. Enviar Argumentos a la Función:

Puedes enviar argumentos a la función que se ejecutará mediante el Timer:

```python
from threading import Timer

def tarea_con_argumentos(nombre, edad):
    print(f"¡Hola {nombre}! Tienes {edad} años.")

# Crear un Timer con un retraso de 3 segundos y enviar argumentos a la función
mi_timer_args = Timer(3, tarea_con_argumentos, args=("Juan", 25))

# Iniciar el Timer
mi_timer_args.start()
```

4.2. Cancelación del Timer:

Puedes cancelar un Timer antes de que se haya ejecutado la función si es necesario:

```
from threading import Timer

def tarea_programada():
    print("¡Tarea programada realizada!")

# Crear un Timer con un retraso de 5 segundos y la función a ejecutar
mi_timer = Timer(5, tarea_programada)

# Iniciar el Timer
mi_timer.start()

# Cancelar el Timer antes de que se ejecute
mi_timer.cancel()
```

La cancelación del Timer es útil si, por alguna razón, decides que ya no deseas ejecutar la tarea programada.

4.3. Ejemplo con timer

Un escenario común para utilizar la clase `Timer` es cuando necesitas ejecutar una tarea periódica o una tarea después de un tiempo específico. Aquí tienes un ejemplo sencillo donde un temporizador se utiliza para realizar una acción después de un retraso y luego repetir esa acción periódicamente:

```python
from threading import Timer

def tarea_periodica():
    print("¡Tarea periódica realizada!")

# Crear un Timer que ejecutará la tarea_periodica después de 2 segundos
mi_timer = Timer(2, tarea_periodica)

# Iniciar el Timer
mi_timer.start()

# Realizar otras operaciones mientras el temporizador cuenta regresiva
print("Realizando otras operaciones mientras se espera...")

# Esperar a que el temporizador complete su cuenta regresiva
mi_timer.join()

# Luego, crear un Timer que ejecutará la tarea_periodica cada 5 segundos
mi_timer_periodico = Timer(5, tarea_periodica)

# Iniciar el Timer periódico
mi_timer_periodico.start()

# Realizar otras operaciones mientras el temporizador periódico está en ejecución
for _ in range(3):
    print("Realizando otras operaciones mientras se espera la próxima ejecución...")

# Cancelar el Timer periódico después de 15 segundos
mi_timer_periodico.cancel()

print("Programa principal terminado.")
```

En este ejemplo:

1. Se crea un `Timer` que ejecutará `tarea_periodica` después de 2 segundos.
2. Mientras el temporizador cuenta regresiva, el programa principal puede realizar otras operaciones.
3. Después de la ejecución única, se crea un nuevo `Timer` periódico que ejecutará `tarea_periodica` cada 5 segundos.
4. Mientras el temporizador periódico está en ejecución, el programa principal puede realizar otras operaciones.
5. Después de 15 segundos, el temporizador periódico se cancela.

Esto ilustra cómo usar un temporizador para ejecutar tareas después de ciertos retrasos y también de manera periódica. Es útil en situaciones donde necesitas realizar acciones programadas en aplicaciones concurrentes o en segundo plano.