Universidad del Valle de México

LAUREATE INTERNATIONAL UNIVERSITIES®

Actividad 7
Ejercicios

PROGRAMACIÓN CONCURRENTE

Docente. Cesar Garcia Martinez

José Emiliano Jauregui Guzmán

Por siempre responsable de lo que se ha cultivado

10 de febrero - 17 de febrero del 2025

EJERCICIOS SOBRE PROGRAMACIÓN CONCURRENTE

1. Con base en el material consultado en la unidad resuelve los ejercicios que se plantean acerca de los siguientes temas:

   - Concurrencia
   - OpenMP
   - Herramientas automáticas para la detección de problemas de concurrencia

PROGRAMACIÓN CONCURRENTE

Introducción

En los problemas clásicos de la concurrencia se tratan situaciones donde múltiples procesos deben interactuar con recursos compartidos sin crear condiciones de carrera o bloqueos. Los problemas de los fumadores y el barbero dormilón son ejemplos clásicos de sincronización entre procesos.

Instrucciones:

- Revisa los ejemplos de implementación en cada uno de los lenguajes de programación que se presentan en los materiales de estudio y abstrae la forma de implementar la concurrencia en cada uno.
- Realiza un ejemplo en C y uno en Java, puedes tomar los que se proporcionan en los materiales de revisión o realizar uno diferente.
- Redacta un informe en este mismo documento que incluya:
  - Código de ejemplo
  - Descripción del ejercicio realizado
  - Imagen con resultado de la ejecución
- Realiza una explicación general de la implementación haciendo énfasis en la parte de concurrencia.

1. Problema de los Fumadores

Este problema involucra a tres fumadores que tienen un ingrediente cada uno para fumar un cigarro (tabaco, fósforos y papel), pero necesitan los otros dos ingredientes que están disponibles en una mesa. Un agente coloca los ingredientes en la mesa, y los fumadores, al ver que los ingredientes que necesitan están disponibles, toman lo que les falta y fuman.

Se usan semáforos para sincronizar los procesos. Los semáforos controlan el acceso a los recursos compartidos y garantizan que los fumadores no se bloqueen por los ingredientes.

main.py

```python
import threading
import random
import time

# Semáforos para sincronización
mutex = threading.Semaphore(1)
ingredientes = [threading.Semaphore(0), threading.Semaphore(0), threading.Semaphore(0)]

fumador_tabaco = 0
fumador_papel = 1
fumador_fosforos = 2

# Ingredientes en la mesa
mesa = [False, False, False]

# Función del agente
def agente():
    while True:
        time.sleep(random.uniform(1, 3))
        i1, i2 = random.sample([0, 1, 2], 2)
        mesa[i1] = True
        mesa[i2] = True
        print(f"El agente pone en la mesa: {['Tabaco', 'Papel', 'Fósforos'][i1]} y {['Tabaco', 'Papel', 'Fósforos'][i2]}")

        # Despierta a los fumadores que necesitan esos ingredientes
        ingredientes[i1].release()
        ingredientes[i2].release()

# Función de cada fumador:
def fumador(tipo):
    while True:
        ingredientes[tipo].acquire()

        # El fumador toma los otros dos ingredientes de la mesa y fuma.
        mutex.acquire()  # El fumador bloquea el acceso a la mesa.
        if tipo == fumador_tabaco:
            while not (mesa[fumador_papel] and mesa[fumador_fosforos]):
                time.sleep(0.1)
            mesa[fumador_papel] = mesa[fumador_fosforos] = False
            print(f"Fumador con {['Tabaco', 'Papel', 'Fósforos'][tipo]} inicia a fumar.")
        elif tipo == fumador_papel:
            while not (mesa[fumador_tabaco] and mesa[fumador_fosforos]):
                time.sleep(0.1)
            mesa[fumador_tabaco] = mesa[fumador_fosforos] = False
            print(f"Fumador con {['Tabaco', 'Papel', 'Fósforos'][tipo]} inicia a fumar.")
        else:  # fumador_fosforos
            while not (mesa[fumador_tabaco] and mesa[fumador_papel]):
                time.sleep(0.1)
            mesa[fumador_tabaco] = mesa[fumador_papel] = False
            print(f"Fumador con {['Tabaco', 'Papel', 'Fósforos'][tipo]} inicia a fumar.")

        mutex.release()  # Libera el acceso a la mesa.

        # Simula el tiempo que tarda en fumar
        time.sleep(random.uniform(2, 4))  # El fumador fuma durante un tiempo aleatorio.

        print(f"Fumador con {['Tabaco', 'Papel', 'Fósforos'][tipo]} termina de fumar.")

# Crear y ejecutar hilos
def main():
    # Crear los hilos para los fumadores y el agente
    hilos = []
    for i in range(3):
        hilo = threading.Thread(target=fumador, args=(i,))
        hilos.append(hilo)
        hilo.start()

    hilo_agente = threading.Thread(target=agente)
    hilo_agente.start()

    # Unir todos los hilos
    for hilo in hilos:
        hilo.join()
    hilo_agente.join()

if __name__ == "__main__":
    main()
```

```
Fumador con Papel inicia a fumar.
El agente pone en la mesa: Fósforos y Papel
Fumador con Papel termina de fumar.
El agente pone en la mesa: Tabaco y Fósforos
Fumador con Fósforos inicia a fumar.
El agente pone en la mesa: Papel y Tabaco
Fumador con Tabaco inicia a fumar.
Fumador con Fósforos termina de fumar.
El agente pone en la mesa: Papel y Tabaco
Fumador con Tabaco termina de fumar.
El agente pone en la mesa: Papel y Fósforos
Fumador con Papel inicia a fumar.
El agente pone en la mesa: Papel y Fósforos
El agente pone en la mesa: Tabaco y Fósforos
Fumador con Fósforos inicia a fumar.
Fumador con Papel termina de fumar.
El agente pone en la mesa: Fósforos y Papel
Fumador con Tabaco inicia a fumar.
Fumador con Fósforos termina de fumar.
El agente pone en la mesa: Papel y Tabaco
Fumador con Tabaco termina de fumar.
```

La implementación utiliza concurrencia mediante hilos y semáforos para sincronizar a los fumadores y al agente. Los fumadores son representados por hilos que esperan a que los ingredientes que les faltan estén disponibles en la mesa. El agente, también ejecutado en un hilo, coloca dos ingredientes aleatorios sobre la mesa y notifica a los fumadores

correspondientes usando semáforos. Los fumadores bloquean el acceso a la mesa con un semáforo de tipo mutex mientras toman los ingredientes y fuman, y luego liberan los ingredientes para que otros fumadores puedan acceder a ellos. El proceso de fumar se simula con pausas aleatorias, y los mensajes indican cuándo cada fumador comienza y termina de fumar, mientras que el agente sigue colocando ingredientes de manera aleatoria. Esto garantiza una sincronización adecuada, evitando condiciones de carrera entre los hilos.

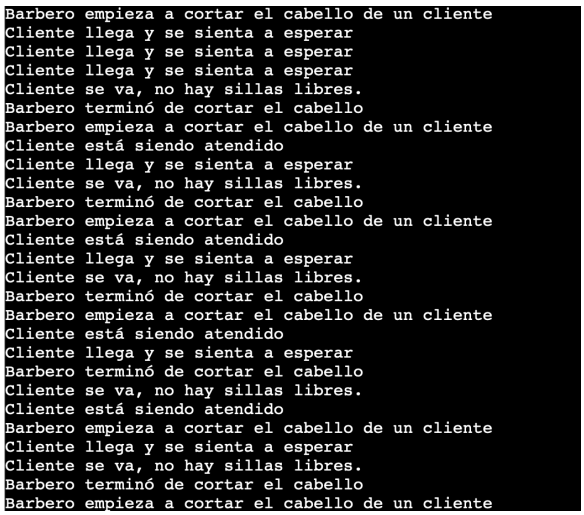
Thomas, H., Cormen, C. E., Leiserson, R. L., & Rivest, C. (s/f). *Introducción a los Algoritmos*. Recuperado el 16 de febrero de 2025, de https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf

2. Problema del Barbero Dormilón

Situación en la que un barbero atiende a los clientes en una barbería, pero solo puede atender a un cliente a la vez. Cuando no hay clientes, el barbero se duerme hasta que uno llega. Los clientes deben esperar en una sala de espera si el barbero está ocupado, pero si la sala está llena, se van sin ser atendidos. El desafío radica en coordinar la concurrencia entre el barbero y los clientes mediante semáforos y mutex, para asegurarse de que el barbero duerma cuando no hay clientes, los clientes esperen su turno sin ocupar recursos innecesarios, y no haya conflictos al acceder a la sala de espera o a la silla del barbero.

main.py

```python
import threading
import random
import time

# Número de sillas de espera
NUM_SILLAS = 3

# Semáforos y mutex
clientes = threading.Semaphore(0)
barbero = threading.Semaphore(0)
mutex = threading.Lock()

# Lista de espera
sillas_de_espera = 0

# Función para el barbero
def barbero_dormilon():
    global sillas_de_espera
    while True:
        clientes.acquire()
        mutex.acquire()
        sillas_de_espera -= 1
        print(f"Barbero empieza a cortar el cabello de un cliente")
        mutex.release()
        time.sleep(random.randint(2, 4))  # Tiempo de corte
        print("Barbero terminó de cortar el cabello")

        # Llama al siguiente cliente
        barbero.release()

# Función para los clientes
def cliente():
    global sillas_de_espera
    while True:
        time.sleep(random.randint(1, 3))
        mutex.acquire()
        if sillas_de_espera < NUM_SILLAS:
            sillas_de_espera += 1
            print("Cliente llega y se sienta a esperar")
            clientes.release()
            mutex.release()
            barbero.acquire()
            print("Cliente está siendo atendido")
        else:
            print("Cliente se va, no hay sillas libres.")
            mutex.release()

# Crear y ejecutar los hilos
def main():
    threads = []

    # Crear hilo para el barbero
    t_barbero = threading.Thread(target=barbero_dormilon)
    threads.append(t_barbero)
    t_barbero.start()

    # Crear hilos para los clientes
    for _ in range(5):
        t_cliente = threading.Thread(target=cliente)
        threads.append(t_cliente)
        t_cliente.start()

    # Esperar a que todos los hilos terminen
    for t in threads:
        t.join()

if __name__ == "__main__":
    main()
```



La concurrencia se gestiona mediante el uso de hilos, semáforos y un mutex para coordinar las interacciones entre el barbero y los clientes. Los clientes llegan en momentos aleatorios y deben esperar en una sala de espera hasta que haya sillas disponibles. Si no hay sillas, el cliente se va. Cuando un cliente se sienta, libera un semáforo que notifica al barbero que hay un cliente esperando. El barbero, por su parte, dormirá si no hay clientes, pero cuando un cliente llega, se despierta, atiende al cliente y luego libera el semáforo para permitir que el cliente termine su corte. Los semáforos aseguran que solo un cliente sea atendido a la vez y gestionan la sincronización para evitar que múltiples clientes sean atendidos simultáneamente. El mutex protege las variables compartidas para evitar condiciones de carrera, garantizando así que la concurrencia entre los hilos no interfiera con el flujo del programa.

Thomas, H., Cormen, C. E., Leiserson, R. L., & Rivest, C. (s/f). *Introducción a los Algoritmos*. Recuperado el 16 de febrero de 2025, de https://dl.ebooksworld.ir/books/Introduction.to.Algorithms.4th.Leiserson.Stein.Rivest.Cormen.MIT.Press.9780262046305.EBooksWorld.ir.pdf

2. Redacta una **conclusión** sobre la utilidad de los diversos lenguajes de programación y su relación con la concurrencia.

En Python que fue el lenguaje que utilice para esta actividad pude ver como la concurrencia es una herramienta muy útil que permite ejecutar varias tareas al mismo tiempo, lo que hace que las aplicaciones sean más rápidas y eficientes, especialmente cuando tienen que hacer muchas operaciones de entrada y salida, como los ejemplos que realice en el cual al principio me confundía que o tenia un orden como tal ya que en

realidad no lo necesitaba. Aunque Python no es el más rápido para tareas que requieren mucho procesamiento debido a algo llamado el Global Interpreter Lock, que impide que varios hilos trabajen al mismo tiempo dentro de un mismo proceso, ofrece buenas soluciones. Como el módulo threading podemos crear varios hilos que realizan tareas de manera simultánea, lo que es perfecto cuando tenemos que esperar mucho tiempo para recibir datos. Si necesitamos aprovechar más de un núcleo del procesador, podemos usar multiprocessing, que crea procesos independientes y hace que el programa pueda realizar tareas pesadas sin las limitaciones. Python tiene varias formas de manejar la concurrencia, aunque no siempre sean las más rápidas, son fáciles de usar y suficientes para la mayoría de las aplicaciones.

Referencias

(S/f-c). Scalahed.com. Recuperado el 15 de febrero de 2025, de https://gc.scalahed.com/recursos/files/r161r/w21040w/Estructuras_de_sistemas_operativos.pdf

Seminario Earlyadopters (2012). *Procesos e Hilos en C.* Recuperado el 16 de febrero de 2025, de https://www.um.es/earlyadopters/actividades/a3/PCD_Activity3_Session1.pdf

DuarteCorporation Tutoriales (Productor). (05 de Julio de 2015). *Tutorial de Thread en Lenguaje C (Hilos en Lenguaje C)* Recuperado el 16 de febrero de 2025, de https://www.youtube.com/watch?v=8D3CtQWB6DI

Cuenca, J. (s.f.). *Programación en el Supercomputador Ben Arabi Programación con OpenMP* Recuperado el 15 de febrero de 2025, de http://www.ditec.um.es/~javiercm/curso_psba/sesion_03_openmp/PSBA_OpenMP.pdf

Santamaría, R. (2018). *Java Threads, Sistemas Distribuidos* Recuperado el 16 de febrero de 2025, de http://vis.usal.es/rodrigo/documentos/sisdis/seminarios/javaThread.pdf