UNIVERSIDAD AUTÓNOMA DE CHIHUAHUA

Facultad de Ingeniería

Ingeniería en Ciencias de la Computación

COMPUTO PARALELO Y DISTRIBUÍDO

1.34 Actividad 12: Simulación problemas de la cena de filósofos y el barbero dormilón

Alumna: ADRIAN (ADORA) GONZÁLEZ DOMÍNGUEZ *[359834]*

Asesor: JOSE SAUL DE LIRA MIRAMONTES

24 de septiembre de 2024

El problema del barbero

El barbero comienza durmiendo. Los clientes llegan con una demora simulada. Si hay espacio en la sala de espera, se sientan y esperan su turno. Si no hay espacio, se van. El barbero despierta cuando llega un cliente y comienza a cortarle el cabello. Una vez termina, atiende al siguiente cliente si lo hay.

El semáforo del barbero (barber_semaphore) asegura que el barbero se duerme cuando no hay clientes, y el semáforo de clientes (customer_semaphore) permite que los clientes esperen hasta ser atendidos. El uso de mutex es crucial para evitar condiciones de carrera, ya que varios hilos (clientes) pueden intentar modificar la lista de espera simultáneamente. El mutex garantiza que solo un cliente modifique la lista a la vez.

```
import threading
import time
import random
import signal

signal.signal(signal.SIGINT, signal.SIG_DFL)


# Define the maximum number of customers and the number of chairs in
the waiting room
MAX_CUSTOMERS = 5
NUM_CHAIRS = 3

# Define the semaphores for the barber, the customers, and the mutex
barber_semaphore = threading.Semaphore(0)
customer_semaphore = threading.Semaphore(0)
mutex = threading.Semaphore(1)

# Define a list to keep track of the waiting customers
waiting_customers = []

# Define the barber thread function
def barber():
    while True:
        print("The barber is sleeping...")
```

```
        barber_semaphore.acquire()
        mutex.acquire()
        if len(waiting_customers) > 0:
            customer = waiting_customers.pop(0)
            print(f"The barber is cutting hair for customer
{customer}")
            mutex.release()
            time.sleep(random.randint(1, 5))
            print(f"The barber has finished cutting hair for customer
{customer}")
            customer_semaphore.release()
        else:
            mutex.release()

# Define the customer thread function
def customer(index):
    global waiting_customers
    time.sleep(10)
    mutex.acquire()
    if len(waiting_customers) < NUM_CHAIRS:
        waiting_customers.append(index)
        print(f"Customer {index} is waiting in the waiting room")
        mutex.release()
        barber_semaphore.release()
        customer_semaphore.acquire()
        print(f"Customer {index} has finished getting a haircut")
    else:
        print(f"Customer {index} is leaving because the waiting room
is full")
        mutex.release()

# Create a thread for the barber
barber_thread = threading.Thread(target=barber)

# Create a thread for each customer
customer_threads = []
for i in range(MAX_CUSTOMERS):
    customer_threads.append(threading.Thread(target=customer,
args=(i,)))
```

```
# Start the barber and customer threads
barber_thread.start()
for thread in customer_threads:
    thread.start()

# Wait for the customer
```

Problema de los filosofos

Hay n filósofos sentados alrededor de una mesa circular. Entre cada par de filósofos hay un palillo. Un filósofo solo puede comer si tiene ambos palillos (el de la izquierda y el de la derecha). Los filósofos alternan entre dos actividades: pensar y comer. Para evitar conflictos, el filósofo debe adquirir los dos palillos antes de comer, y liberarlos cuando termine para que otros filósofos puedan usarlos.

Un semáforo para cada palillo (forks). Inicialmente, cada palillo es libre (Semaphore(1)), lo que significa que puede ser tomado por un filósofo. Cuando un filósofo toma un palillo, adquiere el semáforo correspondiente con acquire(), y lo libera con release() cuando termina de comer. Un semáforo (mutex) es utilizado para proteger la sección crítica donde los filósofos intentan tomar los palillos. Esto garantiza que solo un filósofo intente tomar palillos a la vez para evitar condiciones de carrera

```
import threading
import time
import random

# Define the number of philosophers and forks
num_philosophers = 5
num_forks = num_philosophers

# Define semaphores for the forks and the mutex
forks = [threading.Semaphore(1) for i in range(num_forks)]
mutex = threading.Semaphore(1)
```

```
# Define the philosopher thread function
def philosopher(index):
    while True:
        print(f"Philosopher {index} is thinking...")
        time.sleep(random.randint(1, 5))

        mutex.acquire()

        left_fork_index = index
        right_fork_index = (index + 1) % num_forks

        forks[left_fork_index].acquire()
        forks[right_fork_index].acquire()

        mutex.release()

        print(f"Philosopher {index} is eating...")
        time.sleep(random.randint(1, 5))

        forks[left_fork_index].release()
        forks[right_fork_index].release()

# Create a thread for each philosopher
philosopher_threads = []
for i in range(num_philosophers):
    philosopher_threads.append(threading.Thread(target=philosopher,
args=(i,)))

# Start the philosopher threads
for thread in philosopher_threads:
    thread.start()

# Wait for the philosopher threads to complete
for thread in philosopher_threads:
    thread.join()
```