Laboratorio 4

María Marta Ramirez Gil 21342

Sistemas Operativos

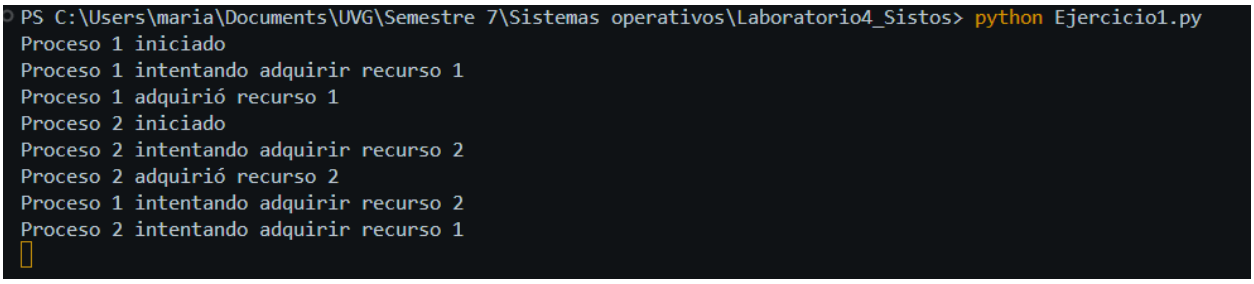
Ejercicio 1

Programa:

```
import threading
import time

# se crean los recursos compartidos
recurso1 = threading.Lock()
recurso2 = threading.Lock()

# funcion para el primer proceso
def proceso1():
    print("Proceso 1 iniciado")
    while True:
        print("Proceso 1 intentando adquirir recurso 1")
        recurso1.acquire()
        print("Proceso 1 adquirió recurso 1")
        time.sleep(1)
        print("Proceso 1 intentando adquirir recurso 2")
        recurso2.acquire()
        print("Proceso 1 adquirió recurso 2")
        # Simulamos el uso de los recursos
        time.sleep(1)
        # Liberamos los recursos
        recurso2.release()
```

```
        recurso1.release()

# funcion para el segundo proceso
def proceso2():
    print("Proceso 2 iniciado")
    while True:
        print("Proceso 2 intentando adquirir recurso 2")
        recurso2.acquire()
        print("Proceso 2 adquirió recurso 2")
        time.sleep(1)
        print("Proceso 2 intentando adquirir recurso 1")
        recurso1.acquire()
        print("Proceso 2 adquirió recurso 1")
        # Simulamos el uso de los recursos
        time.sleep(1)
        # Liberamos los recursos
        recurso1.release()
        recurso2.release()

# se crean los hilos para cada proceso
hilo_proceso1 = threading.Thread(target=proceso1)
hilo_proceso2 = threading.Thread(target=proceso2)

# Inician los hilos
hilo_proceso1.start()
hilo_proceso2.start()

# Esperamos a que los hilos terminen (Nunca va a pasar)
hilo_proceso1.join()
hilo_proceso2.join()
```

Resultados:



Preguntas:

1. **Describe cómo funciona un algoritmo de detección de deadlock y cómo se relaciona con la concurrencia en sistemas operativos.**
   Los algoritmos de detección de deadlocks buscan ciclos en un grafo que representa las asignaciones y solicitudes de recursos entre procesos. Son esenciales para mantener la estabilidad en sistemas concurrentes al identificar y resolver situaciones de interbloqueo.

2. **¿Qué estrategias podrían implementarse para prevenir deadlocks en sistemas concurrentes más complejos que el ejemplo proporcionado?**
   Algunas estrategias para prevenir deadlocks podrían ser:
   - Asignación ordenada de recursos.
   - Uso de prioridades o números de recursos.
   - Implementación de timeouts en las solicitudes de recursos.

3. **Explica cómo se podría modificar el código para introducir una situación de interbloqueo más sutil que no sea tan evidente como la inversión de recursos.**
   Se podría modificar el código para introducir más solicitudes de recursos en diferentes combinaciones y órdenes, aumentando la complejidad de la interacción entre los procesos.

4. **¿Qué métodos de detección y resolución de deadlocks conoces y cómo se aplican en sistemas operativos modernos?**

    - Algoritmos de detección automática de ciclos: Los sistemas operativos utilizan algoritmos para examinar las relaciones de asignación de recursos entre procesos y detectar la presencia de ciclos en el grafo de asignación de recursos. Si se encuentra un ciclo, se activa una alerta de deadlock.
    - Resolución mediante la terminación de procesos involucrados o el rollback de asignaciones: Una vez que se detecta un deadlock, el sistema operativo puede tomar medidas para resolverlo. Una estrategia común es terminar uno o más procesos involucrados en el deadlock para liberar los recursos que están reteniendo.
    - Implementados en el kernel de sistemas operativos modernos para mantener la estabilidad del sistema: Estos métodos están integrados en el kernel del sistema operativo y se ejecutan automáticamente en segundo plano para garantizar la estabilidad del sistema.

Ejercicio 2

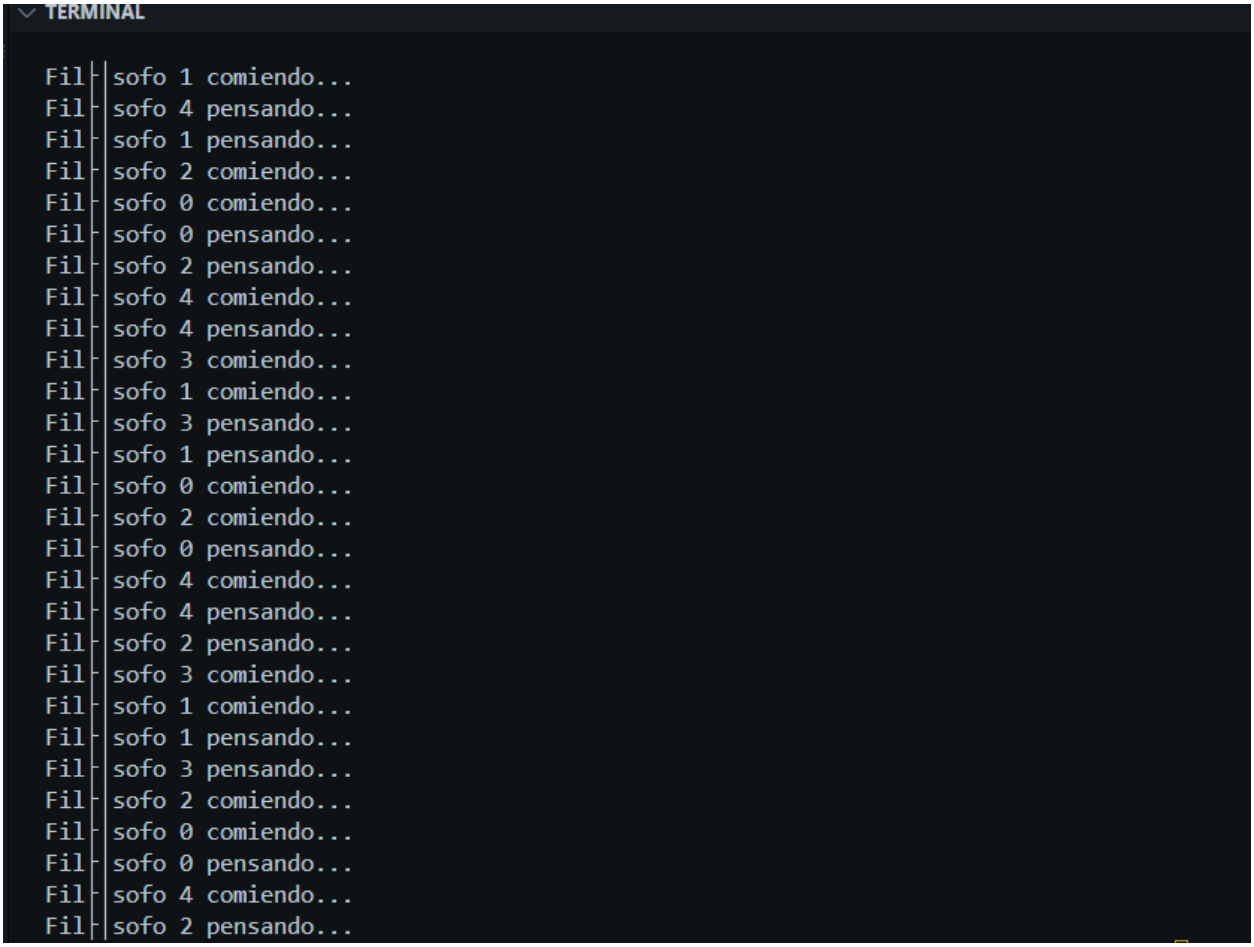
Programa:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define NUM_FILOSOFOS 5

pthread_mutex_t tenedores[NUM_FILOSOFOS];
sem_t mutex;
```

```
void *filosofo(void *arg) {
    int id = *(int*)arg;
    int izquierda = id;
    int derecha = (id + 1) % NUM_FILOSOFOS;

    while (1) {
        printf("Filósofo %d pensando...\n", id);
        usleep(rand() % 3000000); // Tiempo pensando

        // Solicitar los tenedores
        sem_wait(&mutex);
        pthread_mutex_lock(&tenedores[izquierda]);
        pthread_mutex_lock(&tenedores[derecha]);
        sem_post(&mutex);

        printf("Filósofo %d comiendo...\n", id);
        usleep(rand() % 3000000); // Tiempo comiendo

        // Liberar los tenedores
        pthread_mutex_unlock(&tenedores[izquierda]);
        pthread_mutex_unlock(&tenedores[derecha]);
    }
}

int main() {
    pthread_t filosofos[NUM_FILOSOFOS];
    int ids[NUM_FILOSOFOS];

    // Inicializar semáforo
    sem_init(&mutex, 0, 1);

    // Inicializar mutex para los tenedores
    for (int i = 0; i < NUM_FILOSOFOS; i++) {
        pthread_mutex_init(&tenedores[i], NULL);
    }
```

```
    // Crear hilos para los filósofos
    for (int i = 0; i < NUM_FILOSOFOS; i++) {
        ids[i] = i;
        pthread_create(&filosofos[i], NULL, filosofo, &ids[i]);
    }

    // Esperar a que terminen los hilos
    for (int i = 0; i < NUM_FILOSOFOS; i++) {
        pthread_join(filosofos[i], NULL);
    }

    // Destruir mutex y semáforo
    sem_destroy(&mutex);
    for (int i = 0; i < NUM_FILOSOFOS; i++) {
        pthread_mutex_destroy(&tenedores[i]);
    }

    return 0;
}
```

Resultados:



Preguntas:

1. **¿Cuáles son las limitaciones del enfoque de solución con semáforos en el problema de los filósofos cenando? ¿Se pueden mejorar estas soluciones?**
   - Los semáforos pueden ser propensos a errores si no se manejan correctamente.
   - No garantizan la prevención de deadlocks de forma inherente.
   - Pueden resultar complicados de usar en aplicaciones más complejas.

2. **Describe un escenario en el que la implementación actual podría conducir a un deadlock y propón una solución alternativa para evitarlo.**

   Todos los filósofos intentan tomar su tenedor izquierdo simultáneamente y

quedan bloqueados esperando el derecho.
Solución: Implementar un algoritmo de detección de deadlock y permitir un número máximo de intentos de adquisición de tenedores.

3. **¿Qué diferencias clave existen entre la solución con semáforos y otras técnicas de concurrencia, como monitores o variables de condición?**

   Los monitores ofrecen un enfoque más estructurado para la sincronización. Las variables de condición permiten a los hilos esperar eventos específicos.

4. **¿Cómo se puede garantizar la equidad en la asignación de recursos en el problema de los filósofos cenando?**

   Implementar un algoritmo de asignación justo que evite que un filósofo quede bloqueado indefinidamente y distribuya equitativamente los recursos.

Ejercicio 3

Programa:

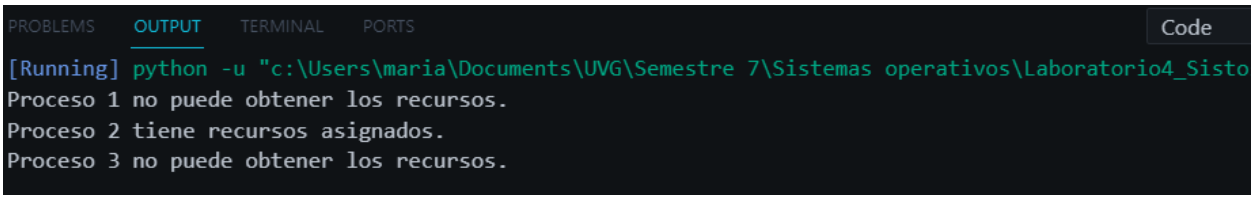
```
class BankerAlgorithm:
    def __init__(self, n_processes, n_resources):
        self.n_processes = n_processes
        self.n_resources = n_resources
        self.max_resources = [5, 3, 7]  # Cantidad máxima de rec
        self.available_resources = self.max_resources.copy()

        self.allocation = [[0] * n_resources for _ in range(n_pr
        self.max_claim = [[0] * n_resources for _ in range(n_pro

        self.initialize_processes()

    def initialize_processes(self):
```

```
        # Simulación de asignación máxima para cada proceso
        self.max_claim[0] = [7, 5, 3]
        self.max_claim[1] = [3, 2, 2]
        self.max_claim[2] = [9, 0, 2]
        self.max_claim[3] = [2, 2, 2]
        self.max_claim[4] = [4, 3, 3]

    def request_resources(self, process_id, request):
        # Verificar si la solicitud es válida
        if all(0 <= request[i] <= self.max_claim[process_id][i]
            # Verificar si hay suficientes recursos disponibles
            if all(request[i] <= self.available_resources[i] for
                # Simular asignación temporal
                for i in range(self.n_resources):
                    self.available_resources[i] -= request[i]
                    self.allocation[process_id][i] += request[i]

                # Verificar si la asignación temporal es segura
                if self.is_safe():
                    return True
                else:
                    # Deshacer la asignación temporal si no es s
                    for i in range(self.n_resources):
                        self.available_resources[i] += request[i
                        self.allocation[process_id][i] -= reques
                    return False
            else:
                return False
        else:
            return False

    def release_resources(self, process_id, release):
        # Liberar los recursos asignados
        for i in range(self.n_resources):
            self.available_resources[i] += release[i]
            self.allocation[process_id][i] -= release[i]
```

```
    def is_safe(self):
        # Verificar si existe una secuencia segura utilizando el
        work = self.available_resources.copy()
        finish = [False] * self.n_processes

        while True:
            # Buscar un proceso que pueda ejecutar de manera seg
            found = False
            for i in range(self.n_processes):
                if not finish[i] and all(self.allocation[i][j] <
                    # Ejecutar el proceso de manera segura
                    for j in range(self.n_resources):
                        work[j] += self.allocation[i][j]
                    finish[i] = True
                    found = True

            # Si no se encuentra ningún proceso seguro, salir de
            if not found:
                break

        # Verificar si todos los procesos se ejecutaron de manera
        return all(finish)

# Ejemplo de uso
banker = BankerAlgorithm(n_processes=5, n_resources=3)

# Proceso 1 solicita recursos
if banker.request_resources(process_id=0, request=[3, 2, 2]):
    print("Proceso 1 tiene recursos asignados.")
else:
    print("Proceso 1 no puede obtener los recursos.")

# Proceso 2 solicita recursos
if banker.request_resources(process_id=1, request=[1, 0, 2]):
    print("Proceso 2 tiene recursos asignados.")
```

```
else:
    print("Proceso 2 no puede obtener los recursos.")

# Proceso 1 libera recursos
banker.release_resources(process_id=0, release=[1, 1, 1])

# Proceso 3 solicita recursos
if banker.request_resources(process_id=2, request=[4, 2, 2]):
    print("Proceso 3 tiene recursos asignados.")
else:
    print("Proceso 3 no puede obtener los recursos.")
```

Resultados:



Preguntas:

1. **¿Qué requisitos deben cumplirse para que un sistema esté en un estado seguro según el algoritmo del banquero? ¿Por qué es importante?**

   Para que un sistema esté en un estado seguro, debe haber suficientes recursos disponibles para satisfacer las solicitudes máximas de todos los procesos. Esto significa que, en cualquier momento, el sistema debe poder asignar recursos de manera segura sin conducir a un deadlock.
   Es importante porque garantiza que el sistema pueda ejecutar todas las tareas de manera eficiente sin quedar bloqueado en un estado de deadlock, lo que puede provocar la parálisis del sistema y la pérdida de rendimiento.

2. **Explica cómo se pueden detectar ciclos de espera en un grafo de asignación de recursos y cómo esto se relaciona con la posibilidad de un deadlock.**

   Los ciclos de espera en un grafo de asignación de recursos se pueden

detectar buscando ciclos en el grafo donde los procesos están esperando recursos que están siendo retenidos por otros procesos en el ciclo.
Esto se relaciona con la posibilidad de un deadlock porque un ciclo de espera en el grafo significa que los procesos están bloqueados esperando recursos que nunca serán liberados, lo que puede conducir a un deadlock si no se toman medidas para evitarlo.

3. **¿Cómo afectaría una implementación incorrecta del algoritmo del banquero al sistema?Proporciona ejemplos concretos.**

   Una implementación incorrecta del algoritmo del banquero puede llevar a situaciones de inseguridad en las que el sistema asigna recursos de manera que podría conducir a un deadlock.
   Por ejemplo, si el algoritmo no verifica adecuadamente si la asignación de recursos podría llevar a un estado inseguro, podría permitir asignaciones que eventualmente conduzcan a un deadlock.

¿Cuál es la complejidad computacional del algoritmo del banquero y cómo podría impactar en

sistemas con un gran número de procesos y recursos?

La complejidad computacional del algoritmo del banquero es lineal, O(n * m^2), donde 'n' es el número de procesos y 'm' es el número de tipos de recursos.

Esto podría impactar en sistemas con un gran número de procesos y recursos ya que el tiempo de ejecución del algoritmo aumentaría proporcionalmente al cuadrado del número de recursos. En sistemas con recursos limitados, esto podría causar un aumento significativo en el tiempo de procesamiento y afectar el rendimiento del sistema.