Moroni Veronica - Oficialdegui Nicolas

Trabajo Práctico Nº 4
Concurrencia (Sincronización)

1. Explica qué es el problema de la sección crítica en concurrencia entre procesos y da un ejemplo de código o pseudocódigo explicando los fragmentos relevantes

El problema de la sección crítica se refiere a la situación en la que varios procesos o hilos comparten recursos y pueden acceder a ellos simultáneamente. Esto puede llevar a resultados inesperados o incorrectos si no se gestiona adecuadamente. Es decir que es una sección del código que no debe ser ejecutada por varios procesos o hilos simultáneamente.

Ejercicio 1 en Git:
Este código en Python utiliza el módulo threading, que proporciona funcionalidades para trabajar con hilos.
Se crea una variable global llamada contador, que inicia en 0.
Se define una función llamada incrementar(). Esta función incrementa la variable contador 200 veces usando un bucle for.
Se crean el hilo1 y hilo2 utilizando la clase Thread.
Se inician ambos hilos llamando al método start()en cada uno de ellos. Esto comienza la ejecución de las funciones incrementar()en paralelo en cada uno de los hilos.
Se utiliza el método join()en ambos hilos (hilo1 y hilo2) para esperar a que ambos hilos completen su ejecución antes de continuar.
Una vez que ambos hilos han terminado de ejecutar la función incrementar(), el programa principal imprime el valor final de la variable contador. Dado que ambos hilos incrementan la variable en paralelo, el valor final puede variar en cada ejecución debido a condiciones de concurrencia, y es posible que no sea exactamente 400 (200 incrementos por hilo).

2. Que es un semáforo y para que se utilizan?. ¿Qué tipos de semáforos existen y cuáles son sus diferencias?

Los semáforos son herramientas fundamentales en la programación concurrente para garantizar la sincronización y la exclusión mutua entre procesos o hilos, es decir que se utilizan en programación concurrente para controlar el acceso a recursos compartidos por múltiples procesos o hilos.
Existen dos tipos de semáforos: semáforos binarios y semáforos contadores y las diferencias son las siguientes:
Un semáforo binario sólo tiene dos estados abierto(0) o cerrado(1), mientras que un semáforo contador puede tener múltiples valores enteros.

Los semáforos binarios se utilizan principalmente para la exclusión mutua, es decir que solo un proceso o hilo puede acceder a una sección crítica a la vez, mientras que los semáforos contadores se utilizan para controlar el acceso a recursos compartidos con límites específicos.
Los semáforos binarios son simples y adecuados cuando solo se necesita controlar el acceso a una sección crítica. Los semáforos contadores son más flexibles y se utilizan para controlar el acceso a múltiples instancias de un recurso compartido.
Un semáforo contador puede tener un valor entero mayor que 1, lo que permite un acceso concurrente limitado al recurso compartido.

3. Da un ejemplo con código de una posible utilización de semáforos (súbelo a GitHub y adjunta el enlace).
Ejercicio 3 en Git: se crea un semáforo con capacidad de 2. Cada hilo intentará adquirir el semáforo antes de ejecutar su tarea. Si el semáforo está disponible, el hilo lo adquiere y realiza la tarea. Después de completar la tarea, el hilo libera el semáforo para que otros hilos puedan adquirirlo. Esto garantiza que solo se ejecuten simultáneamente hasta 2 hilos al mismo tiempo.

4. Describe brevemente qué son los monitores y cómo se utilizan. ¿Qué diferencia tiene con los semáforos?

Los monitores son un conjunto de procedimientos que proporcionan el acceso con la exclusión mutua a un recurso o conjunto de recursos compartidos por un grupo de procesos.
Los monitores están diseñados para crear una estructura organizativa clara alrededor de los datos compartidos lo que facilita la gestión de la concurrencia. Los datos compartidos y las operaciones que los manipulan están encapsulados dentro del monitor, es decir actúan como una barrera alrededor de un recurso compartido. Solo permite que un proceso entre y utilice ese a la vez, imponiendo una secuencia específica de acceso. Los procesos que deseen utilizar el recurso deben esperar su turno fuera del monitor. En cambio los semáforos no encapsulan directamente los datos o las operaciones relacionadas con ellos.
A diferencia de los semáforos, los monitores ofrecen la ventaja de que los programadores solo necesitan invocar métodos de entrada al monitor, lo que reduce la posibilidad de errores en la gestión de recursos compartidos. Si un proceso necesita esperar dentro del monitor, se suspende y permite que otro proceso entre para evitar bloqueos

5. Explica con tus palabras:

a. Problema del búfer limitado
b. El problema de los lectores-escritores
c. El problema de los filósofos comensales

Realiza un ejemplo de código de cada uno de ellos utilizando monitores y semáforos (súbelo a GitHub y adjunta el enlace)

a.Problema del búfer limitado: este problema involucra una memoria intermedia o buffer con una capacidad máxima limitada que se comparte entre productores y consumidores. Los productores colocando elementos en el búfer y los consumidores los retirarán. El problema principal es garantizar que los productores no coloquen elementos en el búfer si está lleno y que los consumidores no retiren elementos si está vacío.
Ejercicio5 a en Git: el código implementa un monitor para resolver el problema del búfer limitado y garantizar la sincronización entre productores y consumidores. El monitor asegura que los hilos esperen de manera segura cuando sea necesario y se notifiquen mutuamente cuando cambie la disponibilidad de recursos compartidos.

b. El problema de los lectores-escritores:

Este problema trata sobre el acceso a un recurso compartido, como una base de datos, por parte de Múltiples procesos (lectores y escritores). Los lectores pueden acceder al recurso simultáneamente y leer su contenido, pero los escritores necesitan un acceso exclusivo para escribir en él. El objetivo es permitir la máxima concurrencia de lectores y, al mismo tiempo, garantizar la integridad de los datos al controlar el acceso de los escritores.

Ejercicio 5.b en Git: se utiliza semáforos para controlar el acceso a los recursos compartidos. El semáforo `database_semaphore` se utiliza para permitir o bloquear el acceso a la base de datos. El semáforo `readers_semaphore` se utiliza para permitir o bloquear el acceso de los lectores. El semáforo `writers_semaphore` se utiliza para permitir o bloquear el acceso de los escritores.
Cuando un lector quiere leer, primero adquiere el semáforo `writers_semaphore` para asegurarse de que no haya escritores escribiendo. Luego incrementa el número de lectores activos y si es el primer lector, adquiere el semáforo `database_semaphore` para tener acceso exclusivo a la base de datos. Después de eso, libera el semáforo `writers_semaphore` y `readers_semaphore` para permitir que otros lectores lean.
El lector realiza la lectura simulada de datos y luego realiza las operaciones inversas antes de terminar: adquiere el semáforo `readers_semaphore`, disminuye el número de lectores activos y si no hay más lectores activos, libera el semáforo `database_semaphore`.
El escritor sigue un proceso similar. Adquiere el semáforo `database_semaphore` para obtener acceso exclusivo a la base de datos, realiza la escritura simulada de datos y luego libera el semáforo `database_semaphore`.

Después de crear los hilos de lectura y escritura, se inician y esperan a que todos los hilos terminen. Al final, se imprime un mensaje indicando que todos los lectores y escritores han terminado.

C. El problema de los filósofos comensales:

Este problema involucra a un grupo de filósofos sentados alrededor de una mesa con platos de comida y palillos. Cada filósofo debe alternar entre pensar y comer, pero solo pueden hacerlo si tienen acceso a dos palillos adyacentes. El desafío es evitar el bloqueo mutuo o el hambre infinita, donde un filósofo no puede comer debido a la competencia por los palillos.

Ejercicio 5.c en Git: en este codigo cada filósofo es un hilo y hay un semáforo para cada tenedor. Se utiliza un semáforo mutex para evitar que varios filósofos intenten tomar los mismos tenedores al mismo tiempo. Cada filósofo primero piensa durante dos segundos, luego adquiere el mutex y los dos tenedores adyacentes antes de comenzar a comer durante otros dos segundos. Luego, los tenedores se liberan y el filósofo vuelve a pensar.

6. Pueden utilizarse los monitores junto a los semáforos?. Es caso afirmativo da un ejemplo de cómo se haría.

Sí, es posible utilizar semáforos junto con monitores en programación concurrente para lograr la sincronización de hilos y evitar problemas de concurrencia. Sin embargo, es importante entender cómo funcionan ambos conceptos y cómo se pueden combinar de manera efectiva. En general, los monitores tienden a ser más convenientes y seguros, ya que abstraen la sincronización a un nivel más alto y evitan errores comunes relacionados con la concurrencia.

Ejercicio 6 en Git:
La clase `IntersectionMonitor` actúa como un monitor para controlar el acceso a la intersección. Utiliza un semáforo para permitir que solo un coche pueda entrar a la vez.
La función `car_thread` simula el movimiento de un coche. Cada coche espera aleatoriamente antes de llegar a la intersección, luego solicita entrar al monitor y finalmente espera un tiempo aleatorio dentro de la intersección antes de salir.
En el bloque principal del programa, se crean hilos para representar cada coche. Estos hilos ejecutan la función `car_thread` pasando el identificador del coche y el objeto `IntersectionMonitor`. Luego, se inician los hilos.

Finalmente, se espera a que todos los hilos terminen utilizando el método `join()` en cada hilo. Esto asegura que el programa principal no termine antes de que todos los hilos hayan finalizado su ejecución.

7. Investiga y analiza qué sistema de solución de problemas de concurrencia proporciona el lenguaje Java, Python y PHP.

Java: proporciona un sólido soporte para la programación concurrente. Algunas de las características y bibliotecas clave incluyen:

- Permite crear y gestionar hilos (Threads) para lograr la concurrencia. La clase java.lang.Thread y la interfaz java.lang.Runnable son componentes esenciales para trabajar con hilos.

- Proporciona un conjunto de clases e interfaces en el paquete java.util.concurrent que simplifican la programación concurrente. Esto incluye clases como Executor, ThreadPoolExecutor, Semaphore.
- Ofrece mecanismos de sincronización como synchronized y volatile para garantizar que varios hilos accedan a datos compartidos de manera segura.

Python:
- El módulo threading se usa para trabajar con hilos, pero debido al Global Interpreter Lock (GIL), no es eficaz para la concurrencia real en sistemas multi-core.
- Ofrece también el módulo multiprocessing permite la creación de procesos independientes para trabajar con procesos en lugar de hilos y el módulo asyncio para la programación asíncrona.

PHP: no ofrece soporte nativo para la concurrencia a nivel de hilos. Sin embargo, se pueden utilizar técnicas como la programación asíncrona o el uso de extensiones específicas como `pthreads` para trabajar con hilos en PHP. Sin embargo, esta extensión no es tan utilizada ni tan madura como las soluciones de concurrencia en Java.

8. ¿A que hace referencia el pasaje de mensajes entre procesos?. Explícalo brevemente y da un ejemplo.

El pasaje de mensajes entre procesos es una técnica de comunicación utilizada en sistemas operativos y programación concurrente para permitir que los procesos (programas en ejecución) se comuniquen entre sí. En lugar de compartir directamente

memoria o variables, los procesos envían y reciben mensajes para intercambiar información de manera segura y controlada.

Ejercicio 8 en Git:

Se crea un proceso hijo utilizando la función `Process()` y se le pasa un número como argumento utilizando el argumento `args`. El proceso hijo realiza un cálculo simple multiplicando el número por 2 y luego imprime el resultado en pantalla. El proceso principal espera a que el proceso hijo termine utilizando el método `join()` y después imprime un mensaje indicando que ha terminado.

9. Que es interbloqueo o bloqueo mutuo (DeadLock)?. Explícalo con un ejemplo.

El interbloqueo, también conocido como bloqueo mutuo o deadlock, es una situación en la que dos o más procesos quedan atrapados y no pueden continuar ejecutándose debido a que cada uno de ellos está esperando por un recurso que está siendo utilizado por otro proceso.

Por ejemplo, considera dos procesos, A y B, que necesitan acceder a dos recursos exclusivos, R1 y R2, respectivamente. Si el proceso A adquiere el recurso R1 y luego espera para obtener R2, mientras que el proceso B adquiere R2 y espera para obtener R1, se crea un interbloqueo. Ambos procesos están esperando a que el otro libere el recurso que necesitan para continuar, pero ninguno de ellos puede hacerlo.

En esta situación, los procesos quedan atrapados indefinidamente y la ejecución del programa se detiene. El interbloqueo es un problema importante en sistemas concurrentes y debe ser evitado o resuelto mediante técnicas como la asignación ordenada de recursos o el uso de algoritmos de detección y recuperación de interbloqueo.

10. ¿Qué diferencia existe entre LiveLock y DeadLock?. Da ejemplos.

LiveLock y Deadlock son dos situaciones problemáticas que involucran a procesos o hilos de ejecución, pero se diferencian en cómo se manifiestan y cómo los procesos intentan lidiar con ellas. En un deadlock, los procesos están completamente bloqueados y no pueden continuar ejecutándose. En un livelock, los procesos están activos y ejecutándose, pero no pueden avanzar debido a una situación de competencia. Ambas situaciones son indeseables en sistemas concurrentes, y se deben tomar medidas para evitarlas o resolverlas cuando ocurren.

Por ejemplo, supongamos que dos personas están caminando hacia el otro en un pasillo estrecho. Si ambas personas intentan moverse al mismo lado para permitir que

el otro pase primero, terminarán bloqueadas y no podrán avanzar. Esto es un ejemplo de DeadLock.

Por otro lado, si ambas personas intentan moverse al mismo lado para permitir que la otra pase primero, pero luego cambian de dirección al mismo tiempo, terminarán moviéndose en círculos sin avanzar. Esto es un ejemplo de LiveLock.