CANDADOS Y EXCLUSIÓN MUTUA

- AXEL ISRAEL AÑORVE GÓMEZ
- HÉCTOR JAIR LECHUGA CANALES
- LUIS ÁNGEL CALDERÓN ZIMBRÓN

- La gestión de la concurrencia en programación es crucial para la eficiencia y fiabilidad de las aplicaciones. La exclusión mutua previene problemas como las condiciones de carrera al asegurar que varios procesos no ejecuten secciones críticas simultáneamente, manteniendo la integridad de los datos.

INTRODUCCIÓN

CONTEXTUALIZACIÓN DE LA CONCURRENCIA

- La programación concurrente implica la ejecución simultánea de múltiples tareas en un sistema informático, donde procesos o subprocesos trabajan independientemente y comparten recursos. Esto mejora el rendimiento en sistemas con múltiples núcleos, pero también presenta desafíos como condiciones de carrera y bloqueo mutuo.

DESAFÍOS DE LA CONCURRENCIA

.En programación concurrente, se enfrentan desafíos como:

- Condiciones de Carrera: Resultan de múltiples procesos intentando acceder o modificar un recurso compartido sin sincronización, lo que puede causar resultados impredecibles.
- Bloqueo Mutuo (Deadlock): Ocurre cuando dos o más procesos se bloquean esperando indefinidamente por recursos que el otro posee, causando una parálisis en el sistema.
- Condiciones de Competencia (Livelock): Similar al bloqueo mutuo, pero los procesos siguen compitiendo sin hacer progreso en lugar de permanecer inactivos.
- Sincronización y Coherencia de Datos: Garantizar la manipulación segura y consistente de datos compartidos es crucial para evitar resultados incorrectos o corruptos.
- Escalabilidad y Rendimiento: Diseñar sistemas que escalen efectivamente y mantengan un rendimiento óptimo a medida que aumenta el número de procesos concurrentes es esencial.

CANDADOS (LOCKS) EN PROGRAMACIÓN CONCURRENTE

Son mecanismos de sincronización que controlan el acceso a recursos compartidos entre múltiples procesos o subprocesos.

Su objetivo principal es evitar condiciones de carrera, permitiendo que solo un proceso acceda exclusivamente a un recurso en un momento dado. Esto se logra mediante dos operaciones fundamentales: adquirir (lock) y liberar (unlock) el candado asociado al recurso.

Cuando un proceso intenta acceder a un recurso, adquiere el candado asociado. Si está disponible, el proceso continúa; de lo contrario, espera hasta que el candado esté libre. La liberación del candado se realiza cuando el proceso ha terminado de utilizar el recurso. Esto asegura la exclusión mutua, evitando inconsistencias y corrupciones en los datos compartidos que podrían ocurrir con accesos simultáneos.

LOS CANDADOS SE UTILIZAN PARA:

- Sincronización: Los candados se utilizan para sincronizar el acceso a secciones críticas del código, asegurando que ciertas operaciones se realicen de manera ordenada y predecible.
- Control de Concurrencia: Ayudan a evitar condiciones de carrera y garantizan la coherencia de los datos en sistemas multi-hilo.
- Gestión de Recursos: Los candados pueden utilizarse para gestionar el acceso a recursos compartidos, como archivos, bases de datos o dispositivos de entrada/salida, garantizando que solo un proceso tenga acceso al recurso en un momento dado.

TIPOS DE CANDADOS

1. Candados Reentrantes: En Java, la clase ReentrantLock proporciona candados reentrantes. Un hilo puede adquirir el mismo candado más de una vez sin causar bloqueos. Esto se logra utilizando el método lock() para adquirir el candado y el método unlock() para liberarlo. Un hilo puede llamar a lock() varias veces y debe llamar a unlock() la misma cantidad de veces para liberar completamente el candado.
2. Candados Justos: En Java, los candados justos no están garantizados por defecto, pero puedes implementar un comportamiento similar utilizando la clase ReentrantLock con el constructor que toma un parámetro booleano fair. Al crear un candado con fair establecido en true, se intenta asegurar que los hilos se atiendan en el orden en que realizaron la solicitud.
3. Candados de Lectura/Escritura: En Java, los candados de lectura/escritura se pueden implementar utilizando la clase ReadWriteLock, que tiene dos métodos para adquirir candados de lectura (readLock()) y escritura (writeLock()). Los hilos pueden adquirir el candado de lectura de forma simultánea, siempre que no haya ningún hilo que tenga el candado de escritura. Sin embargo, cuando un hilo desea escribir, se bloqueará hasta que todos los hilos que están leyendo hayan terminado

PROGRAMACIÓN CONCURRENTE

La exclusión mutua en la programación concurrente se refiere a garantizar que solo un proceso o hilo pueda acceder exclusivamente a un recurso compartido en un momento dado. Esto evita condiciones de carrera y asegura la coherencia de los datos. Se logra mediante el uso de candados, semáforos, mutex y otros mecanismos que permiten que un proceso obtenga acceso exclusivo al recurso, evitando que otros lo accedan hasta que se libere. Esto ayuda a mantener la integridad de los datos y evita problemas de sincronización.

MÉTODOS PARA LOGRAR LA EXCLUSIÓN MUTUA

- ***Semáforos y Mutex***

Son herramientas clave para la sincronización de hilos. Los semáforos, presentados por Allen B. Downey en "The Little Book of Semaphores", se utilizan para resolver problemas de exclusión mutua. Por otro lado, los mutex, detallados en "Java Concurrency in Practice" de Brian Goetz et al., son objetos que permiten a los hilos controlar el acceso exclusivo a recursos compartidos mediante la adquisición y liberación.

- ***Mutex en Java***

la interfaz **java.util.concurrent.locks.Lock** proporciona una implementación de mutex que ofrece mayor control y flexibilidad en comparación con la sincronización basada en la palabra clave **synchronized.** Baeldung ofrece una explicación detallada sobre qué es un mutex y cómo se implementa en Java, brindando una comprensión más profunda de la exclusión mutua en este lenguaje de programación.

EJEMPLOS DE ALGORITMOS DE EXCLUSION MUTUA

- ***Algoritmo de Peterson***

Método clásico para lograr la exclusión mutua entre dos procesos. Este algoritmo utiliza banderas compartidas y turnos para coordinar el acceso a la sección crítica

En este algoritmo, los procesos p0 y p1 no pueden estar en la sección crítica al mismo tiempo. Si p0 está en la sección crítica, entonces bandera[0] es verdadera, lo que significa que p1 ha terminado su sección crítica o está esperando para entrar. De manera similar, si turno es 0, p1 está esperando para entrar a la sección crítica1. Este algoritmo básico puede generalizarse fácilmente a un número arbitrario de procesos1. Es una herramienta fundamental para garantizar la exclusión mutua en sistemas concurrentes.

```python
# Algoritmo para dos procesos
bandera = [False, False]  # Inicialización de las banderas
turno = 0  # No es necesario asignar un turno

def proceso_0():
    bandera[0] = True
    turno = 1
    while bandera[1] and turno == 1:
        pass  # Espera activa
    # Sección crítica
    bandera[0] = False

def proceso_1():
    bandera[1] = True
    turno = 0
    while bandera[0] and turno == 0:
        pass  # Espera activa
    # Sección crítica
    bandera[1] = False
```

EJEMPLOS DE ALGORITMOS DE EXCLUSION MUTUA

- ***Algoritmo de Dekker***

Método clásico para la exclusión mutua entre dos procesos. Al igual que el algoritmo de Peterson, utiliza banderas y turnos para coordinar el acceso a la sección crítica. Ambos algoritmos son ejemplos importantes de cómo se puede lograr la exclusión mutua en entornos de programación concurrente.

```
process p1(var c: tControl);
begin
  repeat
    (* Preprotocolo *)
    c.p1p := true;
    while c.p2p do
      if c.turno <> 1 then
        begin
          c.p1p := false;
          while c.turno <> 1 do;
          c.p1p := true;
        end;
    (* Sección Crítica *)
    write('SC1 ');
    write('SC2 ');
    (* Postprotocolo *)
    c.p1p := false;
    c.turno := 2;
    (* Sección No Crítica *)
    write('SNC1 ');
    write('SNC2 ');
  forever
end;
```

```
type tControl = recor
    p1p,p2p: boolean;
    turno: integer;
    end;
```

APLICACIONES Y CASOS DE USO

La exclusión mutua es crucial en sistemas informáticos como sistemas operativos, bases de datos y aplicaciones concurrentes.

La falta de exclusión mutua puede conducir a condiciones de carrera, inconsistencia de datos y bloqueos, lo que afecta negativamente al rendimiento y la fiabilidad del sistema.

CONCLUSION

- La gestión efectiva de la concurrencia en entornos de programación es crucial para el desarrollo de software robusto y eficiente. La sincronización adecuada de procesos y subprocesos es esencial para evitar condiciones de carrera y garantizar la integridad de los datos en entornos multi-hilo. Los mecanismos de exclusión mutua, como los candados, permiten que solo un proceso acceda exclusivamente a un recurso compartido en un momento dado, evitando problemas como la inconsistencia de datos y el bloqueo mutuo. Es fundamental abordar desafíos como las condiciones de carrera y el bloqueo mutuo mediante técnicas adecuadas de sincronización y gestión de recursos. Al comprender y aplicar principios de exclusión mutua y sincronización, podemos desarrollar software que aproveche al máximo los recursos de hardware disponibles y garantice un comportamiento predecible y coherente en entornos multi-hilo. La programación concurrente y la exclusión mutua son elementos fundamentales en la construcción de sistemas informáticos escalables y confiables, y al utilizar adecuadamente los candados y otras técnicas de sincronización, podemos enfrentar los desafíos de la concurrencia y desarrollar software de alta calidad que satisfaga las demandas de los entornos multi-hilo actuales.

REFERENCIAS

[1] Peterson, G. L. (1981). Myths About the Mutual Exclusion Problem. Information Processing Letters, 12(3), 115–116.

[2] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., & Lea, D. (2006). Java Concurrency in Practice. Addison-Wesley.

[3] Allen, A. B. (2016). The Little Book of Semaphores. Green Tea Press.

[4] Baeldung. (s.f.). What Is Mutex? Recuperado de https://www.baeldung.com/cs/what-is-mutex

[5] Baeldung. (s.f.). Java Concurrent Locks. Recuperado de https://www.baeldung.com/java-concurrent-locks

[6]Silberschatz, A. (2005). Fundamentos de sistemas operativos (7ª edición). McGraw-Hill Interamericana.

[7] Hofri, M. (1990). Proof of a Mutual Exclusion Algorithm. Operating Systems Review, 24(1).

GRACIAS POR SU ATENCIÓN