Universidad Nacional Autónoma de México

Facultad de ciencias

Computación Concurrente 2024-1

Profesora:

Gilde Valeria Rodríguez Jiménez

Ayudantes:

Gibrán Aguilar Zuñiga

Luis Angel Leyva Castillo

Rogelio Alcantar Arenas

Equipo:

Concorriente

Integrantes:

DJLP

ADLG

Práctica 2

EJERCICIOS

1.- Implementación del Algoritmo de Peterson

Implementar el algoritmo de Peterson visto en clase, esto con el fin de generar un CANDADO, con el fin de solucionar el problema de Exclusión Mutua, para 2 hilos.Deberá venir implementado con los métodos lock() y unlock(). (Se proporcionará un archivo que contiene dichos métodos).

Los archivos que resuelven este punto se encuentran en la carpeta Ejercicio1, donde están: ***Peterson.java*** *con los metodos lock() y unlock(),* ***MyThread.java*** *representando el hilo con el metodo run y el* ***Main.java*** *que es el que se ejecutara una vez compilados los archivos de la carpeta con el clásico: java Main*

2.- Implementación del Algoritmo del Filtro Modificado

Utilizando la idea del Algoritmo del Filtro visto en la clase, tienen que crear una implementación que permita pasar a lo más L hilos a la sección Crítica con L < N, es decir, que varios hilos puedan entrar a la sección crítica. Deben venir los métodos acquire() y release() los cuales serán implementados.

Los archivos que resuelven este punto se encuentran en la carpeta Ejercicio2, donde se encuentran: ***MultiThreadedSection.java*** *que es la parte del Filtro Modificado con sus métodos acquire() y release() y* ***MyThread.java*** *representando el hilo con el metodo run y uso de los métodos del filtro modificado en el main. Una vez compilados los archivos.java ejecutar (para ver el funcionamiento del programa) el comando: java MyThread*

Para los ejercicios 3 y 4 se hizo la actualización indicativa de los pom de cada proyecto.

3.- Solución del Problema de los Inversores

En este problema deberemos sentar a los 4 inversionistas, junto al Zeratun en una mesa redonda, pues le encanta tener su Orden del Templo, en este caso, para ejemplificar de mejor manera se puede ver de la siguiente manera:

Como se puede notar, no hay suficientes tenedores debido a la falta de presupuesto, por lo que solo tenemos un total de 5 tenedores, debemos encontrar la manera de resolver el problema.

Usa el Algoritmo de Peterson para resolverlo de una Manera y el Algoritmo del Filtro Modificado para poder resolverlo de otra.

Tomando en cuenta estas consideraciones:

- En MestaTest.java en el metodo inicializaInversionistas() se cambio esta linea de codigo “inversionistas[i] = clazz.getDeclaredConstructor(cArg).newInstance(semaforo);” por esta “inversionistas[i] = new InversionistaFiltro();”
- Se agregaron 2 constructores vacíos en las 2 clases de inversionistas.
- También se agregó un ciclo while que se repetirá 500 veces, esto por indicaciones en el Override run de la clase abstracta Inversionista.java

Con dichas consideraciones y con lo implementado en los archivos.java pasan los tests.

4.- Solución del Problema del Estacionamiento

En este problema, se tiene que controlar el acceso, aunque no hay orden de entrada para esto, lo que sí debe cumplir es lo siguiente:

• Solo hay 50 lugares disponibles
• Para el ingreso se tiene una caseta que controlara dicho ingreso
• Al estacionamiento pueden llegar un sin fín de automóviles
• Por el lugar se pueden pelear 2 automóviles, pues no hay tanto espacio
• Si hay lugar disponible, el auto puede estacionarse
• Se tiene un contador general que dice cuantos lugares disponibles / lugares ocupados
Consideraciones:
• Los carros solo permanecen cierto tiempo estacionados, por lo que paulatinamente desocupan lugares.
• Aunque llegue un sinfín de automóviles, se puede llegar a configurar de tal manera que llegue una cantidad N de carros.

Aquí lo que se agregó fue la clase Filtro.java que extiende de la interfaz Semaphore ya que viene a ser el filtro modificado; hasta el momento con lo implementado en los archivos.java pasan los tests.

Como nota adicional hay 2 carpetas “Inversionistas archivos.java” y “Estacionamiento archivos.java” que contienen las clases que se implementaron, esto por cualquier error que pudiese ocurrir con los proyectos.

TEORÍA

1. Para la parte de teoría debes demostrar tus dos códigos creados es decir:

a. Demuestra tu algoritmo del Filtro Modificado

i. Exclusión Mutua

Como podemos ver en la clase MyThread se creó un objeto del tipo MultiThreadedSection “section” que hace uso de los métodos acquire() y release().

Estos métodos pertenecen a la clase MultiThreadedSection; entonces en esa clase tenemos que el código utiliza un objeto `ReentrantLock` para crear un cerrojo (lock) en el constructor de la clase. Un cerrojo es un mecanismo que permite que solo un hilo acceda a una sección crítica de código a la vez. Cuando un hilo adquiere el cerrojo, bloquea a otros hilos hasta que se libera el cerrojo.

1. El método `acquire()` se llama cuando un hilo intenta ingresar a la sección crítica. Antes de permitir que un hilo ingrese, el código verifica si el número actual de hilos dentro de la sección crítica (`count`) es menor que el número máximo permitido (`L`). Si es así, el hilo puede ingresar de inmediato. De lo contrario, el hilo se bloquea utilizando `condition.await()`. Esto asegura que, si la sección crítica está llena (es decir, ya se alcanzó el límite de hilos permitidos), el hilo en espera no puede ingresar hasta que se libere espacio.
2. El método `release()` se llama cuando un hilo sale de la sección crítica. En este punto, el contador `count` se decrementa. Si el valor de `count` es menor que el número máximo permitido `L`, se utiliza `condition.signal()` para despertar a un hilo en espera. Esto significa que, cuando un hilo sale de la sección crítica, otro hilo en espera (si lo hay) tendrá la oportunidad de entrar.

Por lo que este algoritmo del Filtro Modificado cumple la exclusión mutua.

ii. No Deadlock

1. Uso de un solo tipo de bloqueo: El código utiliza un único objeto `ReentrantLock` para gestionar el acceso a la sección crítica. Esto garantiza que los hilos adquieran y liberen el bloqueo de manera coherente, evitando situaciones en las que los hilos se bloqueen indefinidamente debido a la falta de coherencia en el uso de bloqueos.

2. Uso de la condición para la espera: Cuando un hilo intenta adquirir la sección crítica y encuentra que el número máximo de hilos (`L`) ya está dentro de la sección crítica, se bloquea utilizando `condition.await()`. Esto significa que el hilo liberará el bloqueo y entrará en un estado de espera hasta que la condición se cumpla (es decir, hasta que haya espacio en la sección crítica). Esto evita que los hilos entren en un estado de espera activa y, en su lugar, se bloqueen de manera eficiente hasta que sea apropiado despertarlos.

3. Liberación del bloqueo antes de señalar la condición: Cuando un hilo sale de la sección crítica, primero decrementa el contador y luego verifica si hay espacio disponible en la sección crítica (`count < L`). Solo después de verificar esto, señala la condición usando `condition.signal()`. Esto asegura que un hilo recién liberado no señale inmediatamente a otro hilo antes de que pueda entrar a la sección crítica.

En resumen, el código utiliza de manera efectiva un solo objeto de bloqueo (`ReentrantLock`) junto con una condición (`Condition`) para gestionar la entrada y salida de hilos en la sección crítica. Esto garantiza que los hilos se bloqueen de manera adecuada y que no se produzcan situaciones de deadlock.

Por lo tanto, este algoritmo del Filtro Modificado cumple con la propiedad de deadlock-free.

iii. Libre de Hambruna

A continuación, se explican las razones por las cuales este código cumple con esta propiedad:

1. Uso de un ReentrantLock: El uso de un ReentrantLock garantiza que los hilos obtendrán acceso exclusivo a la sección crítica de manera justa y sin preferencias arbitrarias. Cada hilo que intenta adquirir el bloqueo espera su turno en la cola de bloqueo.

2. Uso de una Condition: La Condition se utiliza para coordinar el acceso a la sección crítica. Cuando un hilo intenta adquirir el bloqueo en el método `acquire()`, si ya se han alcanzado el número máximo de hilos permitidos en la sección crítica (L), se bloquea en la Condition mediante `condition.await()`. Esto significa que los hilos que llegan después del límite esperarán pacientemente hasta que se libere espacio.

3. Liberación del bloqueo y señalización: Cuando un hilo sale de la sección crítica en el método `release()`, decrementa el contador de hilos en la sección crítica y verifica si hay espacio para más hilos (es decir, `count < L`).

Si es así, despierta a uno de los hilos en espera mediante `condition.signal()`. Esto asegura que un hilo recién liberado permita que otro hilo entre en la sección crítica, evitando así la inanición.

En resumen, el código garantiza que ningún hilo quedará bloqueado indefinidamente en espera de entrar en la sección crítica. Siempre que haya espacio disponible (según el valor de L) y un hilo desee entrar, eventualmente se le permitirá hacerlo una vez que otros hilos salgan de la sección crítica y liberen espacio o sean notificados para que puedan intentar adquirir el bloqueo nuevamente.

Por lo tanto, este algoritmo del Filtro Modificado cumple con la propiedad de "starvation-free".

2. Tu solución del problema de los inversionistas usando el candado, cumple:

a. Justicia

1. **Exclusión mutua:** La propiedad de justicia en este contexto se refiere a la garantía de que cuando un inversionista desea tomar un tenedor (bloquear) o soltar un tenedor (desbloquear) [esta lógica viene implementada en TenedorImpl.java], eventualmente se le permitirá hacerlo, es decir, no quedará bloqueado indefinidamente esperando un tenedor o un acceso al tenedor. La implementación de `PetersonLock` garantiza la exclusión mutua, ya que solo un inversionista puede bloquear y desbloquear un tenedor a la vez, evitando así que varios inversionistas intenten tomar el mismo tenedor al mismo tiempo.
2. **Alternancia:** El algoritmo de Peterson es conocido por su capacidad de alternar entre dos hilos que intentan bloquear y desbloquear al mismo tiempo. Esto asegura que ningún inversionista quede permanentemente bloqueado, ya que si dos inversionistas intentan bloquear al mismo tiempo, uno de ellos se convertirá en el "víctima" y esperará hasta que el otro termine su operación. Esto garantiza que todos los hilos tengan la oportunidad de avanzar y evitar la inanición.

En resumen, el código cumple con la propiedad de "justicia" en el contexto del problema de los inversionistas porque garantiza la exclusión mutua entre los hilos que desean tomar tenedores y utiliza el algoritmo de Peterson para alternar entre hilos, lo que evita que un filósofo quede bloqueado indefinidamente y asegura que todos tengan la oportunidad de avanzar en su lógica y acceder a los tenedores cuando sea posible.

3. Tu solución del problema de los inversionistas usando el Filtro Modificado, cumple:

a. Justicia

1. **Uso de Exclusión Mutua**: El código utiliza el algoritmo de candados y un semáforo (filtro modificado) para garantizar la exclusión mutua. Cada inversionista debe adquirir el semáforo antes de tomar los tenedores y liberarlo después de comer que es lo que se hace en InversionistaFiltro.java. Esto asegura que solo un inversionista a la vez puede tomar tenedores y comer, evitando conflictos y garantizando la exclusión mutua.
2. **Alternancia entre Inversionistas**: Los inversionistas compiten por el semáforo para acceder a la mesa. Si un inversionista no puede adquirir el semáforo debido a que otros están comiendo, esperará en cola hasta que sea su turno. Esto garantiza que todos los inversionistas tengan la oportunidad de acceder a la mesa en un orden justo y no queden bloqueados indefinidamente.
3. **Liberación del Semáforo**: Después de que un inversionista ha terminado de comer y suelta los tenedores, libera el semáforo. Esto permite que otro inversionista entre en la sección crítica (la mesa) si está esperando, asegurando que no haya bloqueo permanente o inanición.

En resumen, el código en InversionistaFiltro.java implementa una solución que cumple con la propiedad de justicia al garantizar que todos los inversionistas tengan la oportunidad de acceder a la mesa, comer y liberarla de manera justa y sin bloqueos indefinidos.

Por lo tanto, satisface la propiedad de justicia en el contexto de la cena de los inversionistas usando el filtro modificado.

4. Tu solución del estacionamiento cumple:

a. Exclusión Mutua

1. **Interfaz `Lock` y `Semaphore`:**
    - Las interfaces `Lock` y `Semaphore` proporcionan una estructura genérica para la implementación de candados y semáforos.
    - Estas interfaces definirían métodos como `lock()`, `unlock()`, `acquire()`, y `release()`, que son fundamentales para lograr la exclusión mutua. Sin embargo solo hicimos uso del filtro modificado ya que permite hacer mejor uso en la cuestión de la capacidad del estacionamiento y el número de carros. Dichos métodos del filtro son usados para el Override del run y el método estaciona().

2. **Clase `Filtro` (implementación de Semaphore):**
    - La clase `Filtro` implementa la interfaz `Semaphore`.
    - Utiliza un `Lock` (`ReentrantLock`) y una `Condition` para controlar el acceso a una sección crítica, lo que es una forma estándar de garantizar la exclusión mutua.
    - Los métodos `acquire()` y `release()` se utilizan para adquirir y liberar el semáforo, respectivamente, y se aseguran de que no más de un número máximo de hilos pueda entrar en la sección crítica a la vez.
3. **Clase `Lugar` y método `estaciona()`:**
    - La clase `Lugar` contiene un método `estaciona()` que se utiliza para simular que un carro se estaciona en un lugar.
    - Dentro de este método, se utiliza el semáforo `Filtro` (implementado como una instancia de `Semaphore`) para garantizar que solo un carro a la vez pueda estacionarse en el lugar, lo que satisface la exclusión mutua.

En resumen, el código satisface la propiedad de Exclusión Mutua al utilizar un semáforo (`Filtro`) para controlar el acceso concurrente a una sección crítica (el lugar de estacionamiento) y garantizar que solo un carro (hilo) pueda estacionarse en un lugar a la vez. Esto se logra a través de la implementación adecuada de la interfaz `Semaphore` y el uso de un `Lock` y una `Condition` para gestionar la concurrencia de manera segura.

b. No Deadlock

El código utiliza semáforos (`Filtro` implementa la interfaz `Semaphore`) para controlar el acceso a la sección crítica en la clase `Lugar`. Con los semáforos con esta herramienta prevenimos los deadlocks. Los métodos `acquire()` y `release()` se implementan de manera que los hilos adquieran y liberen recursos de manera ordenada y eviten bloqueos mutuos.

Se utiliza un `Lock` reentrante (`ReentrantLock`) en la clase `Filtro` para gestionar el acceso a la sección crítica. Los locks reentrantes permiten que el mismo hilo adquiera y libere el candado varias veces, lo que reduce la posibilidad de bloqueo.

No hay un ciclo de dependencias de recursos que puedan causar que los hilos se bloqueen indefinidamente esperando unos a otros ya que contamos con el semáforo y el filtro.

Por lo que el código si cumple la propiedad de Deadlock-Free

c. Libre de Hambruna

1. **Interfaz `Lock` y `Semaphore`:**
    - Las interfaces `Lock` y `Semaphore` proporcionan una estructura genérica para la implementación de candados y semáforos, que son utilizados para coordinar el acceso concurrente a recursos críticos.
2. **Clase `Filtro` (implementación de Semaphore):**
    - La clase `Filtro` implementa la interfaz `Semaphore` y utiliza un mecanismo de bloqueo (`ReentrantLock`) y una condición (`Condition`) para gestionar el acceso a una sección crítica.
    - El método `acquire()` espera hasta que se cumpla cierta condición (por ejemplo, hasta que haya espacio disponible en la sección crítica) y luego permite que los hilos entren. Esto garantiza que ningún hilo espere indefinidamente y satisface la propiedad "starvation-free".
3. **Clase `Lugar` y método `estaciona()`:**
    - La clase `Lugar` modela un lugar de estacionamiento y utiliza el semáforo `Filtro` para controlar el acceso a ese lugar.
    - El método `estaciona()` adquiere el semáforo antes de estacionar un carro y lo libera después, lo que garantiza que solo un carro a la vez pueda estacionarse en un lugar.
4. **Clase `Estacionamiento`:**
    - La clase `Estacionamiento` gestiona una colección de lugares de estacionamiento y proporciona métodos para obtener lugares disponibles y asignar lugares a carros. Además del uso que se hace del filtro modificado en el Override run() de dicha clase.

En resumen, el código utiliza semáforos (`Filtro`) y bloqueos (`ReentrantLock`) de manera adecuada para coordinar el acceso a recursos críticos, como los lugares de estacionamiento. Esto asegura que los carros no esperen indefinidamente para estacionarse y, por lo tanto, cumple con la propiedad de "starvation-free". La implementación garantiza que todos los carros tengan la oportunidad de estacionarse en algún momento sin quedar atrapados en una espera infinita.

d. Justicia

1. **Exclusión mutua:** La propiedad de justicia en este contexto se refiere a la garantía de que cuando un número de coches se estaciona, eventualmente irán por barbacoa y de ahí se les terminara el tiempo del estacionamiento con lo cual saldrán y desocuparon un lugar, es decir, no quedará bloqueado indefinidamente un lugar ya que todos los coches que entran eventualmente tendrán que salir.

2. **Alternancia:** Esto se cumple por la cuestión de nuestro filtro modificado que maneja de buena manera la entrada de los coches con la capacidad del estacionamiento. El uso del filtro modificado garantiza que todos los hilos tengan la oportunidad de avanzar y evitar la inanición, además de claramente la cuestión de entrar, estacionarse, ir por barbacoa y salir del estacionamiento.

En resumen, el código cumple con la propiedad de "justicia" en el contexto del problema del estacionamiento porque garantiza la exclusión mutua entre los hilos que desean entrar al estacionamiento y utiliza el algoritmo del filtro modificado para alternar entre hilos, lo que evita que un coche quede estacionado indefinidamente y asegura que los otros coches tengan la oportunidad de avanzar en su lógica y acceder a los lugares del estacionamiento cuando sea posible.