Universidad Nacional Autónoma de México
Facultad de Ciencias

Computación Concurrente

Práctica 2

Bonilla Reyes Dafne - 319089660
García Ponce Jose Camilo - 319210536
Juárez Ubaldo Juan Aurelio - 421095568

Exclusión Mutua: Locks y Pools en Java

Descripción de los programas

- `ColaConcurrente.java:`

  Programa que simula una cola que trabaja de manera concurrente usando *ExecutorService*.

- `ColaSecuencial.java:`

  Programa que simula una cola que trabaja de manera secuencial (código de la profesora).

- `ColaConcurrenteLock.java:`

  Programa que simula una cola que trabaja de manera concurrente usando *ExecutorService* y candados.

- `Nodo.java:`

  Programa que simula un nodo (código de la profesora).

- `Tarea.java:`

  Programa que simula el envío y realización de tareas usando un semáforo y candado.

- `Scheduler.java:`

  Programa que simula personas que envían tareas mediante el uso de hilos.

Reporte de ejercicios

1. **El programa *ColaSecuencial* es una implementación de una cola secuencial. Implementa una Cola concurrente utilizando una pool de hilos (*ExecutorService*). No utilices candados ni *synchronized*.**

   El programa se implementó en el archivo `ColaConcurrente.java`

2. **Ejecuta varias veces tu implementación con diferentes secuencias de llamadas a métodos ¿Tu implementación funciona de acuerdo a lo que se espera de una cola o suceden *inconsistencias*? Por ejemplo, que se hace *enq(a), enq(b) y deq()* y al final ambos elementos a y b están en la cola. Si existe una ejecución así toma una captura pantalla**

del resultado de tu programa que lo ejemplifique. ***Hint: Para analizar si hay inconsistencias, utiliza la interfaz Future.***

Hay inconsistencias, por ejemplo:

- Se hizo la ejecución de:

```
ColaConcurrente queue = new ColaConcurrente();
queue.deq();
queue.enq("1");
queue.enq("2");
queue.deq();
queue.enq("3");
queue.enq("4");
queue.deq();
queue.deq();
queue.enq("5");
queue.enq("6");
queue.deq();
queue.deq();
queue.deq();
queue.enq("7");
queue.enq("8");
queue.deq();
queue.print();
queue.getExecutor().shutdown();
```

Lo cual nos dio el siguiente resultado:

```
Orden de operaciones
Result: empty
Result: enqueued de 1 lista
Result: enqueued de 2 lista
Result: 1
Result: enqueued de 3 lista
Result: enqueued de 4 lista
Result: 2
Result: 2
Result: enqueued de 5 lista
Result: enqueued de 6 lista
Result: 3
Result: 4
Result: 5
Result: enqueued de 7 lista
Result: enqueued de 8 lista
Result: 6
Elementos de la cola
hnull
7
8
```

Notemos que el 2 solo se encoló una vez, pero salió 2 veces, lo cual es una inconsistencia, porque las colas no funcionan de esa forma.

3. **¿Existen *data races* o *race-conditions*? Explica en qué variables y por qué suceden.**

   Sí, dado que se intenta modificar las variables *tail* y la *head* de la cola, las cuales son variables compartidas (generando *race-conditions*), entonces al tener varios hilos intentando modificarlas pueden causar que varios hilos saquen el elemento mismo elemento como vimos en el inciso anterior (generando *data races*), lo cual no es correcto.

4. **Utiliza candados y/o *synchronized* en tu implementación de forma que los métodos *enq()* y *deq()* sean una sola sección crítica y contesta: ¿existen inconsistencias?**

   El programa se implementó en el archivo `ColaConcurrenteLock.java`. Usamos *synchronized* en los métodos de *enqS()* y *deqS()*, que son los equivalentes de *enq()* y *deq()* originales.

   Además, con esta implementación no hay inconsistencias, solo que en el *print* seguramente no se ve como funciona una cola, debido a que se imprimen los *futures* en orden de como se van agregando y no de como van terminando, por lo tanto, como algunos hilos terminan antes que otros las ejecuciones pueden variar, pero aquí ya no nos pasa el error del inciso anterior al desencolar 2 veces al mismo elemento.

5. **En una pool de hilos (ExecutorService), ¿si utilizamos más hilos equivale a un mayor *throughput*? Argumenta por qué.**

   Si aumentamos en el número de hilos, puede mejorar el *throughput*, pero no siempre, ya que varias características influencian esto, como los recursos del sistema y el tipo de tareas. También, el tiempo de finalización de las tareas es importante, ya que si las tareas son cortas o fáciles, más hilos pueden generar mayor *throughput*. En contraste, si las tareas son largas o tardadas, una cantidad mayor de hilos puede consumir muchos recursos y no generar un mayor *throughput*.[1]

6. **Problema. Supón que perteneces a un equipo de trabajo en el cual estás a cargo de dar acceso a un servidor. Seis personas te pueden mandar tareas para que las ejecutes en el servidor, sin embargo, tienes la instrucción de aceptar una tarea por integrante y no puedes dar acceso a más de tres al mismo tiempo. Diseña e implementa una solución. *Hint: Apóyate del programa Scheduler y Tarea, decide como utilizar un candado y un semáforo.***

   - ¿Tu implementación cumple con Justicia?
     Veamos que nuestra implementacion cumple con justicia, al momento de subir una tarea al servidor, los hilos que no pudieron adquirir el candado se bloquean hasta que el hilo actual lo libere (despues de haber subido la tarea o en caso de haber un error), y todos los hilos eventualmente van a adquirir el candado y subir su tarea. En la ejecucion de tareas, al crear el semaforo utilizamos el parametro true, que es un parametro adicional para configurar la justicia, lo que hace que los hilos que llamaron a acquire() primero podran ejecutar su tarea cuando este disponible (FIFO)
   - Si no es así, describe cómo podrías garantizarla.

Referencias

[1] What are some best practices for using a thread pool executor service in Java? (2023, 25 agosto). https://www.linkedin.com/advice/0/what-some-best-practices-using-thread-pool-executor