UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

Computación Concurrente

Práctica 02

Estrada Garcia Luis Gerardo	319013832
Mora Hernández Dulce Julieta	319236448
Noriega Rodríguez Marcos Julián	319284061

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*.

   Para nuestra implementación usamos el archivo proporcionado de ColaSecuencial.java y Nodo.java, por lo que compilamos el archivo de la siguiente manera:

   javac Nodo.java ColaSecuencial.java

   Y así ya podemos compilar nuestro archivo ColaConcurrente.java:

   javac ColaConcurrente.java

   Para la implementación, comentamos el main en secuencial para hacerlo en el archivo de la cola concurrente, en el cual creamos la pool de hilos y le pasamos a cada hilo la cola que creamos y la tarea **MyRunnable**, la cual hace la función de encolar y desencolar objetos dependiendo del valor de la i del hilo que le pasamos. Al final esperamos un tiempo a que todos los hilos terminen, imprimimos qué tarea realizó cada uno e imprimimos la cola final.

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.*

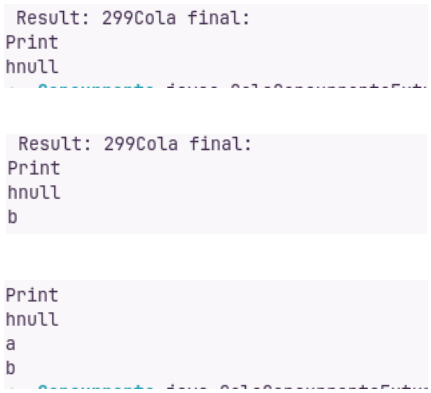
   Para probar nuestro programa creamos el archivo *ColaConcurrenteFuture.java*, el cual se compila y ejecuta de la siguiente manera:

   javac ColaConcurrenteFuture.java
   java ColaConcurrenteFuture

   En este archivo igualmente creamos nuestra pool de hilos con *"ExecutorService executor = Executors.newFixedThreadPool(4)"*; y similar al ejemplo dado, creamos una lista para guardar los objetos futures generados con la función tipo lambda *"futures.add(executor.submit(() ->accion(queue, ntask)));"*.

   Generamos la cola para ejemplificar las inconsistencias, en la cual hacemos *enq(a), enq(b), deq()* y sí suceden inconsistencias con lo que se espera de una cola, ya

que a veces se queda el valor de b en la cola, luego sucede que se queda con valor nulo o que no se desencola ningún elemento:



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

   Sí existen carreras por los datos, ya que podemos no tener actualizado el *next* de la cola en el momento en el que un hilo quiere desencolar un elemento, por lo que puede no desencolar nada o un elemento incorrecto.

   Lo mismo sucede para el método *enq*, como no controlamos cómo acceden los hilos a éste, podemos tener dos hilos queriendo encolar un elemento distinto al mismo tiempo y al final tener inconsistencia de datos.

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?

   Para nuestra solución usamos *synchronized*, en la cual modificamos la cola secuencial, creando los métodos enqSync(), deqSync(), y printSync() como synchronized para garantizar que se ejecuten de forma atómica y como un solo candado.

   Para compilar y ejecutar dicho archivo usamos:

   javac ColaConcurrenteSync.java
   java ColaConcurrenteSync

   Lo que sucede aquí es que siguen habiendo inconsistencias, ya que solo estamos sincronizando los métodos de encolar y desencolar, cuando hay elementos de la cola como los punteros al head o next, de los cuales podemos tener inconsistencias. Otra cosa que creemos que sucede es que no hat justicia, por lo que, aunque

aseguramos que los métodos son de forma atómica, puede suceder que un hilo quiera desencolar cuando todavía no termina la acción de encolar a o b, lo que lleva a un desencole nulo, incorrecto o en un orden no esperado.

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

   Recordemos que una pool de hilos crea un grupo de hilos en espera de tareas por ejecutar, una vez que existe una tarea, uno de los hilos la toma, la ejecuta, finaliza y vuelve a esperar por otra.

   Notemos que no necesariamente se cumple que si utilizamos un mayor número de hilos tendremos un mayor *throughput* ya que existen ocasiones en las que esto no pasa necesariamente, por ejemplo veamos el caso en el que contamos con una cantidad n de núcleos y un número h de hilos tal que $h > n$, si nos encontramos en este caso podriamos causar una disminución en el throughput ya que el procesador estaria realizando cambios de contexto muy seguido ya que los núcleos que "realmente" estarian trabajando son con los que cuenta nuestro sistema.

   Por lo que, hay que tomar en cuenta los recursos con los que contamos, la cantidad de núcleos con los que cuenta nuestro sistema y el número de tareas a realizarse para poder evaluar cual es la mejor opción.

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?
   - Sino es así, describe cómo podrías garantizarla.

   La implementación no cumple con la propiedad de *justicia* ya que aunque el hacer uso de un semáforo nos permite solamente tener 3 tareas ejecutandose a la vez, no precisamente se cumple que el orden de las tareas que se estan ejecutando sea de los hilos que llegan antes, es decir, no necesariamente se sigue un orden FIFO, para resolver esto *java* cuenta con la opción de agregar `justicia = true` como segundo parametro al crear un semaforo, por lo que agregando esto, aseguraremos que se cumpla la propiedad de Justicia.

Nuestra solución se encuentra en los archivos dados (Tarea.java y Scheduler.java).