Concurrencia y Distribución (24/25)

Martín Pérez Pérez, Guillermo Blanco González, Noelia García Hervella

Semana 4 (24 - 28 febrero)

Práctica 4 – Sincronización de hilos

Objetivos: Diferentes formas de sincronización de hilos sobre una variable compartida.

Requisito general a todos los programas concurrentes que implementemos: El programa y todos sus subprocesos/hilos deben **terminar siempre.** Al final del programa debe mostrarse un mensaje parecido a *"Program of exercise X has terminat*ed", para garantizar que **todos los componentes** del programa concurrente terminan correctamente su ejecución.

Primera parte de la práctica

1. **El enfoque de esta práctica es desarrollar un contador concurrente y compartido por un grupo de hilos. Sigue lo siguientes pasos:**

    a. Crear la clase `Counter,` esta debe tener un método increment() que tiene como función incrementar un contador en una unidad. Así mismo debe tener un método que devuelva el valor actual del contador. El valor inicial del contador debe ser 0.

    b. Crea una clase llamada `MyTask` que implemente la interfaz `Runnable` o extienda de `Thread` la cual se construye recibiendo como parámetro un objeto de tipo `Counter.` El hilo debe dormir un tiempo aleatorio entre 0 y 100 milisegundos y a continuación invocar el método `increment()` del contador. Cada hilo debe mostrar el mensaje *"Hello world, I'm the java thread number X."* cuando comience su ejecución y *"Bye from thread number X."* cuando vaya a finalizar.

    c. Crea también una clase principal llamada `P5` en cuyo método `main` se creen varios objetos de la clase MyTask que compartan un solo objeto de tipo `Counter`. Cuando haya finalizado la ejecución de cada hilo, imprime el valor actual del contador. Ejecuta el programa varias veces y con distintos números de hilos. ¿Qué salida obtienes en las diferentes ejecuciones?

2. **`synchronized`:** Modifica la clase `MyTask` para que el incremento del contador se haga desde un bloque sincronizado (el bloque con la palabra reservada `syncrhonized` debe actuar sobre el objeto contador compartido). ¿Cómo cambia esto el comportamiento del programa principal? Familiarízate con las dos formas sintácticas que ofrece Java para tal fin

(bloque sincronizado o método sincronizado), ¿cómo modificarías tu código para implementar un método sincronizado y no un bloque?

3. **`AtomicInteger/LongAdder:`** haz las modificaciones necesarias en el código anterior para usar un `AtomicInteger` y/o `LongAdder`. Para realizar la operación del incremento, usa un método adecuado disponible para estos objetos (consulta la documentación de la API de Java). Explica lo que observas.

4. **`Lock (candado):`** En lugar de bloques sincronizados, la API de Java tiene un conjunto de objetos optimizados de alto nivel para su uso en programas concurrentes. Iremos a utilizar el paquete `java.util.concurrent.locks`, que proporciona un mecanismo de bloqueo similar que proporciona la semántica de `synchonized`: es decir, se puede reemplazar el bloque sincronizado con un uso de un `lock`, que se adquiere antes de modificar el contador y que se libera una vez haberlo hecho. Debes hacer dos programas diferentes utilizando esta técnica; la primera solución debe modificar la clase `Counter` y la segunda debe aplicarse en la clase `MyTask.`

5. **Análisis/Medidas:** modifica tu programa del apartado uno adecuadamente y ejecuta el código del contador para todos los métodos de sincronización empleados, variando la cantidad de hilos utilizados. ¿Puedes notar una diferencia en el rendimiento? Explica tus resultados.

Segunda parte de la práctica

El objetivo de este ejercicio es implementar un programa en Java que permita la gestión concurrente de un archivo, donde varios hilos puedan modificar su contenido de manera segura y sincronizada. El archivo puede ser en formato CSV u otro tipo de archivo de texto.

El programa debe funcionar de la siguiente manera:

1. El archivo contendrá datos organizados por líneas, donde cada línea representa una entrada de información.
2. Se implementará una clase **AdminFile** que tenga la funcionalidad necesaria para permitir a los hilos modificar el archivo de forma segura.
3. Cada hilo recibirá como parámetro el número de línea que desea modificar y el nuevo contenido que desea escribir en esa línea.
4. **AdminFile** debe garantizar que solo un hilo pueda acceder al archivo en un momento dado para evitar conflictos y garantizar la consistencia de los datos.
5. Cuando un hilo accede al archivo para modificar una línea, debe asegurarse de que ningún otro hilo pueda acceder al mismo tiempo.
6. Después de que un hilo haya modificado una línea, debe liberar el acceso al archivo para que otros hilos puedan realizar modificaciones.

Instrucciones:

- Implementa la clase **AdminFile** con los métodos necesarios para garantizar la sincronización de acceso al archivo.
- Crea una clase **MyThread** que implemente la interfaz Runnable o extienda de Thread. Esta clase representará cada hilo que modifica el archivo. Recibirá como parámetros el número de línea a modificar y el nuevo contenido.
- Crea un programa principal que cree varios hilos y los ejecute simultáneamente.
- Verifica que el programa funcione correctamente, asegurándote de que los cambios realizados por los hilos se reflejen correctamente en el archivo y que no haya conflictos de acceso concurrente.

Nota: Puedes utilizar **synchronized**, **Lock**, o **AtomicInteger** según consideres apropiado para sincronizar el acceso al archivo y garantizar la exclusión mutua entre los hilos.

Tercera parte de la práctica

1. Escribe un programa en Java que cree dos hilos: uno que incremente un contador y otro que lo decremente. El contador inicializa en 0. Utiliza un objeto Lock para controlar el acceso al contador y asegura que los cambios en el contador sean atómicos. Además, utiliza un método synchronized para imprimir el valor del contador después de cada operación. Asegúrate de que cada hilo realice su operación (incrementar o decrementar) 1000 veces.
2. Basándote en el programa anterior que utiliza dos hilos para incrementar y decrementar un contador atómicamente, amplía la funcionalidad de tu aplicación para introducir las siguientes características:
    a. Control de Rango: Modifica el programa para que el contador nunca exceda un valor máximo de 10 ni descienda por debajo de un valor mínimo de -10. Si un hilo intenta incrementar o decrementar el contador fuera de estos límites, el hilo debe pausar su operación y reintentar después de un breve período.
    b. Registro de Actividades: Añade una funcionalidad para llevar un registro (log) de todas las operaciones realizadas sobre el contador, incluyendo el valor resultante después de cada operación y el nombre del hilo que realizó la operación. Implementa esto utilizando un método sincronizado para añadir entradas al registro y asegura que el acceso a este registro esté bien sincronizado entre los hilos.
    c. Asegúrate de que el programa complete todas las operaciones planeadas para ambos hilos y que termine correctamente. Proporciona también un método sincronizado para imprimir el registro final de actividades una vez que ambos hilos hayan terminado su ejecución.

Cuarta parte de la práctica

Crea un sistema de gestión de tareas múltiples. En este nuevo escenario, implementarás un programa que maneje tres tipos de tareas diferentes, cada una representada por hilos que realizan operaciones distintas sobre un conjunto común de recursos compartidos:

1. Gestión de Inventario: Implementa hilos que simulan la adición y eliminación de elementos de un inventario (por ejemplo, productos en un almacén - Strings en una lista). Las adiciones y eliminaciones deben ajustarse para no caer en números negativos de productos.
2. Registro de Transacciones: Desarrolla hilos que simulen la escritura de transacciones (como ventas o devoluciones - Eliminar un string insertado). Asegúrate de que las entradas en el registro sean coherentes y que no haya conflictos entre las entradas.
3. Análisis de Datos: Crea hilos que simulen la lectura de los registros del inventario para generar informes (Muestra cuántos elementos existen). Crea dos versiones, una que muestre los datos aproximados en un momento determinado (no atómico) y otro que muestre los datos exactos (atómico)