ENTREGABLE 2.
PROGRAMACIÓN AVANZADA

Documentación Comercio Electrónico

Integrantes:

Matías Senatore

Juan Ignacio Zangaro

Docente:

Ing. Rodrigo Lopez

1. Estructura General del Sistema

- **Clase Pedido:** Define pedidos con un ID único y una prioridad (urgente o normal), utilizando una cola de prioridad para asegurar que los pedidos urgentes se procesen antes que los normales.
- **Clase ProcesadorDePedidos:** Controla el flujo de procesamiento y utiliza ForkJoinPool con hasta 10 hilos asignados para gestionar tareas concurrentes. Se adapta la asignación de hilos en función de la carga de pedidos urgentes y normales.
- **Clases ProcesadorDePago, ProcesadorDeEmpaquetado y ProcesadorDeEnvio:** Simulan las etapas de pago, empaquetado y envío, imprimiendo mensajes y simulando tiempos con retardos. La implementación se mantuvo simple, sin validaciones complejas de pagos.
- **Clase Main:** Genera pedidos aleatoriamente, lo que permite observar en tiempo real la priorización de los urgentes. El sistema permite el procesamiento concurrente, gestionando eficientemente los recursos disponibles.
- **Tests de Verificación:** Se implementaron pruebas con JUnit para asegurar que el sistema funcione sin excepciones y que el pool de hilos se cierre correctamente.

2. Lógica General Aplicada

El flujo de procesamiento de pedidos se maneja secuencialmente en un hilo, asegurando que un hilo maneje todas las etapas de un pedido. PriorityBlockingQueue garantiza que los pedidos urgentes se procesen antes que los normales, asignando hilos libres a urgentes si están disponibles. La complejidad temporal total es O(n log n) debido a la cola de prioridad, y los tiempos estimados para procesar 100 pedidos son de 8 a 15 segundos.

3. Dinámica del Sistema

El sistema utiliza ForkJoinPool para manejar tareas concurrentes, permitiendo el procesamiento eficiente de pedidos incluso en alta demanda. Los pedidos urgentes tienen prioridad, y aunque los normales no se interrumpen, los hilos liberados se asignan a urgentes. La seguridad en el acceso a recursos se asegura mediante mecanismos adecuados para evitar condiciones de carrera. El sistema permite un apagado ordenado, garantizando que no queden pedidos incompletos.

4. Criterios de Diseño

- **Modularidad:** El diseño modular facilita la separación de responsabilidades y futuras expansiones.
- **Optimización de Recursos:** ForkJoinPool optimiza el uso de recursos, ajustando dinámicamente la cantidad de hilos activos en función de la carga.
- **Seguridad:** Se garantiza el acceso seguro a los recursos compartidos, protegiendo las operaciones críticas contra accesos simultáneos no controlados.

5. Dificultades en el transcurso del proyecto

5.1. Dificultades:

1) Nos aparecía el error “Cannot resolve symbol 'junit'” y después “MojoFailureException” debido a problemas con Maven.
2) En la función “shutdown” lanzaba “NullPointerException”.
3) Al principio, estábamos importando “org.testng.annotations.Test” y “org.junit.jupiter.api.Test” sin saber que se tenía que elegir una de las dos.
4) Más adelante, cuando ya solucionamos problemas básicos nos encontramos con que la clase “ProcesadorDePedidos”, no habíamos implementado el método shutdown() ni el método isShutdown() para gestionar el estado del ForkJoinPool.
5) Ya respecto al código, tuvimos problemas al detectar la creación de los pedidos, su importancia para podes testar nosotros si efectivamente estaba funcionando.
6) Nos topamos que estábamos escribiendo mucho código y nos faltaba identificar algunos criterios que debíamos definir nosotros de la letra para seguir la lógica, debido a que lo dábamos por dado y nos salteamos la parte de pactarlo en grupo.
7) Tuvimos problemas en los test para acceder al forkJoinPool.

5.2. Soluciones:

1) Solucionado con el archivo pom.xml y agregando las dependencias, abriendo IntelliJ específicamente en la carpeta del proyecto para que no diera errores.
2) Se solucionó inicializando el ExecutorService distinto de null y pasando como un argumento al método shutdown(), lo que te permite cerrar un pool existente. También, se añadió la importación de TimeUnit para que awaitTermination() funcione correctamente.
3) Elegimos Junit para los test.
4) Se incluyeron ambos métodos para garantizar que se apague correctamente y verificar el estado del ForkJoinPool dentro de la clase.
5) Se agregaron mensajes para ir corroborando el flujo y se pueda controlar en la corrección.
6) Se decidió hacer que los pedidos pares sean urgentes y los normales impares, así como no realizar mayor validación en pago, envío y empaquetado, más que mostrar mensajes ya que no era el eje principal del trabajo. Si se decidió que, aunque los pares eran urgentes, que se genere de forma aleatoria el tiempo de la generación de los pedidos aunque sea sistemático. Esto es para darle más semejanza a la realidad, de que no siempre van a venir ordenados los pedidos urgentes, sino van a estar entrelazados de forma aleatoria con los normales (esto lo hacemos con un random entre 0 y 300 ms, ya que es una medida rápida para que no estemos esperando mucho tiempo la ejecución del código). Otra gran directiva que se tomó fue que los pedidos normales no se interrumpieran para atender a los urgentes, sino que la prioridad yace en los hilos disponibles para atender los pedidos. En caso de que se usara, se estudió la posibilidad de implementar PriorityBlockingQueue. Debido a que la letra solicitaba que se pudieran realizar 100 pedidos, tomamos 10 hilos para que no tardara tanto en ejecutar.
7) Creamos el método auxiliar “esperarTerminacion” como un método público en la clase ProcesadorDePedidos que encapsule la llamada a awaitTermination(), de modo que podamos esperar correctamente a que el procesamiento termine sin acceder directamente al forkJoinPool.