Documentación sobre estrategias de sincronización utilizadas

Diego Flores López 2066033

1. Exclusión mutua con *synchronized*

El acceso a recursos compartidos, como el saldo de una cuenta bancaria, se protege mediante el uso de la palabra clave *synchronized*. Esto garantiza que solo un hilo a la vez pueda ejecutar métodos críticos como **depositar()** o **retirar()**. De esta forma se evita la condición de carrera (*race condition*), en la que múltiples hilos podrían modificar el saldo simultáneamente y producir resultados inconsistentes.

2. Comunicación entre hilos con wait() y notifyAll()

El patrón Productor-Consumidor se implementa mediante un buffer compartido. Para coordinar la producción y el consumo de datos, se utilizan los métodos **wait()** y **notifyAll()**:

- **wait()** suspende la ejecución de un hilo cuando el buffer está lleno (en el caso del productor) o vacío (en el caso del consumidor).
- **notifyAll()** despierta a los hilos en espera cuando el estado del buffer cambia, permitiendo que continúe la ejecución.

Así se evita tanto la sobreproducción como el consumo de datos inexistentes, garantizando un flujo ordenado de información entre productores y consumidores.

3. Manejo de *race conditions*

Las *race conditions* ocurren cuando dos o más hilos acceden a un recurso compartido sin la debida sincronización. En este proyecto, se ejemplifica con la clase CuentaBancaria. Sin *synchronized*, múltiples hilos podrían alterar el saldo de manera inconsistente. La solución aplicada fue encapsular las operaciones críticas dentro de

métodos sincronizados, asegurando que solo un hilo pueda modificar el saldo en un momento dado.

4. Manejo de *deadlocks*

Un *deadlock* se produce cuando dos hilos esperan indefinidamente por recursos que el otro posee, generando un bloqueo circular. En esta práctica, se simula un *deadlock* mediante la clase BloqueoCruzado, donde dos métodos adquieren los bloqueos en distinto orden (recursoA y recursoB). Esto permite observar cómo los hilos pueden quedar atrapados en espera mutua.

La estrategia general para evitar *deadlocks* en sistemas reales incluye:

- Mantener un orden consistente de adquisición de recursos.
- Usar mecanismos como **tryLock()** con tiempo de espera.
- Reducir al mínimo los bloqueos anidados.

En este caso, el *deadlock* se deja intencionalmente como demostración.

5. Uso de ExecutorService

En lugar de crear y gestionar manualmente cada hilo, se utiliza un pool de hilos mediante ExecutorService. Esto permite:

- Reutilizar hilos, evitando el costo de creación repetida.
- Controlar el número máximo de hilos concurrentes.
- Programar tareas de manera más flexible.

En el código de la práctica, ExecutorService se emplea para ejecutar productores y consumidores en paralelo, mostrando cómo se integran tareas concurrentes en un entorno controlado.

6. Análisis de rendimiento y logs

Para evaluar el rendimiento, se mide el tiempo total de ejecución con **System.nanoTime()**. Además, se generan *logs* en consola que muestran:

- Operaciones de depósito y retiro.
- Producción y consumo de datos en el buffer.
- Accesos a recursos que pueden generar *deadlocks*.

Estos registros permiten analizar el comportamiento del sistema bajo concurrencia y verificar que las estrategias de sincronización funcionan correctamente.