Sincronización de Hilos en Java

Vamos a sumergirnos en la sincronización de hilos en Java. Cuando múltiples hilos acceden a recursos compartidos (como variables, colecciones o archivos), necesitas prevenir condiciones de carrera, donde los hilos interfieren entre sí, llevando a resultados impredecibles. Java proporciona varias herramientas para esto: `synchronized`, `Lock` y `Semaphore`. Explicaré cada una, mostraré ejemplos y las compararé para que puedas elegir la adecuada para tus necesidades.

---

1. Palabra clave `synchronized`

La palabra clave `synchronized` asegura que solo un hilo pueda ejecutar un bloque de código o método a la vez. Es la herramienta de sincronización más sencilla, integrada en el núcleo de Java.

Cómo funciona

- Bloquea el monitor de un objeto (cada objeto de Java tiene uno).
- Otros hilos esperan hasta que se libere el bloqueo.

Ejemplo: Contador con `synchronized`

```
class Contador {
    private int cuenta = 0;

    public synchronized void incrementar() {
        cuenta++;
    }

    public int obtenerCuenta() {
        return cuenta;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Contador contador = new Contador();
        Runnable tarea = () -> {
            for (int i = 0; i < 1000; i++) {
                contador.incrementar();
```

```
            }
        };

        Thread t1 = new Thread(tarea);
        Thread t2 = new Thread(tarea);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Cuenta final: " + contador.obtenerCuenta()); // Siempre 2000
    }
}
```

- Sin `synchronized`, `cuenta++` (leer-modificar-escribir) podría solaparse, dando un resultado menor a 2000.
- `synchronized` bloquea la instancia de `Contador`, asegurando actualizaciones atómicas.

Bloque sincronizado También puedes bloquear código específico en lugar de un método completo:

```
public void incrementar() {
    synchronized (this) {
        cuenta++;
    }
}
```

- Usa un objeto diferente como bloqueo si deseas un control más fino:

```
private final Object bloqueo = new Object();
public void incrementar() {
    synchronized (bloqueo) {
        cuenta++;
    }
}
```

Pros y contras

- **Pros**: Simple, integrado, bueno para uso básico.
- **Contras**: Sin flexibilidad (por ejemplo, no se puede interrumpir un hilo en espera), bloquea hasta que se libera.

2. Interfaz `Lock` (java.util.concurrent.locks)

La interfaz `Lock` (por ejemplo, `ReentrantLock`) ofrece más control que `synchronized`. Es parte del paquete `java.util.concurrent` e introdujo para abordar las limitaciones de `synchronized`.

Características clave

- Llamadas explícitas `lock()` y `unlock()`.
- Soporte para bloqueos de intento, temporizadores e interrupción.
- Opción de equidad (hilos esperan en orden).

Ejemplo: Contador con `ReentrantLock`

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class Contador {
    private int cuenta = 0;
    private final Lock bloqueo = new ReentrantLock();

    public void incrementar() {
        bloqueo.lock();
        try {
            cuenta++;
        } finally {
            bloqueo.unlock(); // Siempre desbloquear en finally
        }
    }

    public int obtenerCuenta() {
        return cuenta;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Contador contador = new Contador();
        Runnable tarea = () -> {
            for (int i = 0; i < 1000; i++) {
                contador.incrementar();
```

```
            }
        };

        Thread t1 = new Thread(tarea);
        Thread t2 = new Thread(tarea);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Cuenta final: " + contador.obtenerCuenta()); // Siempre 2000
    }
}
```

- `try-finally` asegura que el bloqueo se libere incluso si ocurre una excepción.

Características avanzadas

- **Bloqueo de intento**: Intento no bloqueante de adquirir el bloqueo:

```
if (bloqueo.tryLock()) {
    try {
        cuenta++;
    } finally {
        bloqueo.unlock();
    }
} else {
    System.out.println("No se pudo adquirir el bloqueo");
}
```

- **Temporizador**: Esperar un tiempo limitado:

```
if (bloqueo.tryLock(1, TimeUnit.SECONDS)) { ... }
```

- **Interruptible**: Permitir que los hilos en espera sean interrumpidos:

```
bloqueo.lockInterruptibly();
```

Pros y contras

- **Pros**: Flexible, soporta características avanzadas, control explícito.
- **Contras**: Más verboso, desbloqueo manual requerido (riesgo de olvidar).

3. `Semaphore`

Un `Semaphore` controla el acceso a un recurso manteniendo un conjunto de permisos. Es ideal para limitar la concurrencia (por ejemplo, máximo 5 hilos pueden acceder a un recurso).

Cómo funciona

- Los hilos adquieren permisos con `acquire()`.
- Liberan permisos con `release()`.
- Si no hay permisos disponibles, los hilos esperan.

Ejemplo: Limitando conexiones de base de datos

```
import java.util.concurrent.Semaphore;

class PoolDeConexiones {
    private final Semaphore semaforo = new Semaphore(3); // Máximo 3 conexiones

    public void conectar() {
        try {
            semaforo.acquire();
            System.out.println(Thread.currentThread().getName() + " conectado");
            Thread.sleep(1000); // Simular trabajo
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            semaforo.release();
            System.out.println(Thread.currentThread().getName() + " desconectado");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        PoolDeConexiones pool = new PoolDeConexiones();
        Runnable tarea = () -> pool.conectar();

        Thread[] hilos = new Thread[10];
        for (int i = 0; i < 10; i++) {
            hilos[i] = new Thread(tarea, "Hilo-" + i);
```

```
            hilos[i].start();
        }
    }
}
```

- Solo 3 hilos pueden “conectar”a la vez; otros esperan hasta que se libere un permiso.

Características avanzadas

- **Equidad**: `new Semaphore(3, true)` asegura orden FIFO.
- **Adquirir intento**: `semaforo.tryAcquire()` verifica sin bloquear.

Pros y contras

- **Pros**: Ideal para pools de recursos, sistema de permisos flexible.
- **Contras**: Más complejo que `synchronized` o `Lock`, no para exclusión mutua simple.

Combinando con `ExecutorService`

Aquí está cómo funcionan estas herramientas con un pool de hilos:

```
import java.util.concurrent.*;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class RecursoCompartido {
    private int valor = 0;
    private final Lock bloqueo = new ReentrantLock();

    public void incrementar() {
        bloqueo.lock();
        try {
            valor++;
        } finally {
            bloqueo.unlock();
        }
    }

    public int obtenerValor() {
```

```
        return valor;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        RecursoCompartido recurso = new RecursoCompartido();
        ExecutorService ejecutor = Executors.newFixedThreadPool(4);

        for (int i = 0; i < 10; i++) {
            ejecutor.submit(() -> {
                for (int j = 0; j < 100; j++) {
                    recurso.incrementar();
                }
            });
        }

        ejecutor.shutdown();
        ejecutor.awaitTermination(5, TimeUnit.SECONDS);
        System.out.println("Valor final: " + recurso.obtenerValor()); // Siempre 1000
    }
}
```

Comparación

Característica	`synchronized`	`Lock` (ReentrantLock)	`Semaphore`
Facilidad de uso	Alta	Media	Baja
Flexibilidad	Baja	Alta	Media
Granularidad	Método/ bloque	Explicito	Basado en permisos
Interruptible	No	Sí	Sí
Opción de equidad	No	Sí	Sí
Caso de uso	Exclusión mutua simple	Control avanzado	Limitación de recursos

---

¿Cuándo usar qué?

- `synchronized`: Rápido, exclusión mutua simple (por ejemplo, proteger una sola variable).

- `Lock`: Necesitas bloqueos de intento, temporizadores o interrupción (por ejemplo, flujos de trabajo complejos).
- `Semaphore`: Limitando el acceso concurrente a un recurso (por ejemplo, pools de conexiones, limitación de tasa).