Programación de Procesos y Servicios

UT2.3: Bloqueos Explícitos y Avanzados. Colecciones Seguras
2º DAM

Limitaciones de Synchronized

El uso de *synchronized* en Java es sencillo y conveniente, pero tiene algunas limitaciones en comparación con las clases de bloqueo más avanzadas

- Falta de control manual sobre la adquisición y liberación del bloqueo.
- No permite interrupciones durante la espera de un bloqueo.
- No permite probar el bloqueo sin bloquear el hilo.
- Solo ofrece un tipo de bloqueo (exclusión mutua).
- No permite asignar diferentes monitores a distintos bloques.

Bloqueos Explícitos: Lock

En Java, la clase ***Lock*** es una interfaz que forma parte del paquete *java.util.concurrent.locks* que proporciona los mecanismos básicos para controlar el acceso a secciones críticas del código. Es como un candado que los hilos deben adquirir antes de ejecutar un bloque de código.

Método	Descripción
lock()	Adquiere el bloqueo. Si otro hilo ya tiene el bloqueo, el hilo actual esperará hasta que lo libere.
unlock()	Libera el bloqueo para que otros hilos puedan adquirirlo.
tryLock()	Intenta adquirir el bloqueo sin bloquear al hilo si no está disponible.
lockInterruptibly()	Adquiere el bloqueo pero puede ser interrumpido mientras espera.

Bloqueos Explícitos: ReentrantLock

ReentrantLock es una implementación de la interfaz *Lock*. Es llamado "reentrante" porque permite que un mismo hilo adquiera el mismo bloqueo varias veces sin quedarse bloqueado. Cada vez que el hilo adquiere el bloqueo, debe liberarlo tantas veces como lo haya adquirido.

- El uso de *ReentrantLock* es sencillo pero más detallado que el de synchronized, porque debes adquirir el bloqueo manualmente y también liberarlo dentro de un bloque *finally* para asegurarte de que siempre se libere.

Bloqueos Explícitos: ReentrantLock

```
public class EjemploReentrantLock {
    private final Lock lock = new ReentrantLock();  // Crear un ReentrantLock
    public void metodoCritico() {
        lock.lock();  // Adquirir el bloqueo
        try {
            // Código crítico que debe estar sincronizado
            System.out.println("Ejecutando código protegido por el bloqueo.");
        } finally {
            lock.unlock();  // Asegurarse de liberar el bloqueo en el bloque finally
        }
    }
```

Bloqueos Explícitos: ReadWriteLock

En Java, la clase ***ReadWriteLock*** es una interfaz que permite gestionar bloqueos de lectura y escritura de manera más eficiente cuando se tiene un recurso compartido.

Este tipo de bloqueo es útil cuando muchos hilos necesitan leer datos al mismo tiempo, pero solo unos pocos necesitan escribir, ya que permite múltiples lecturas concurrentes siempre que no haya ninguna escritura en curso.

Bloqueos Explícitos: ReadWriteLock

ReadWriteLock presenta dos tipos de bloqueos:

- Lock de Lectura (readLock.lock()): Permite que múltiples hilos adquieran el bloqueo simultáneamente si solo necesitan leer datos, siempre y cuando no haya ningún hilo que esté escribiendo. De esta manera, varios hilos pueden acceder al recurso al mismo tiempo si solo están realizando operaciones de lectura.
- Lock de Escritura (writeLock.lock()): Solo un hilo puede adquirir el bloqueo de escritura a la vez. Cuando un hilo está escribiendo, ningún otro hilo (ni de lectura ni de escritura) puede acceder al recurso, garantizando que los datos no sean modificados simultáneamente por múltiples hilos.

Bloqueos Explícitos: ReentrantReadWriteLock

ReentrantReadWriteLock es la implementación más común de la interfaz *ReadWriteLock*. Este bloqueo es "reentrante", lo que significa que el mismo hilo puede adquirir el bloqueo varias veces sin bloquearse.

- Múltiples hilos pueden obtener el readLock simultáneamente, permitiendo lecturas concurrentes.
- Solo un hilo puede obtener el *writeLock*, y mientras lo tenga, ningún otro hilo puede leer ni escribir.
- Si un hilo tiene el *writeLock*, no puede ser adquirido por otro hilo hasta que se libere.

Bloqueos Explícitos: ReentrantReadWriteLock

```
class RecursoCompartido {
    private int datos = 0;
    private final ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

    public void leer() {
        lock.readLock().lock();  // Adquirir el bloqueo de lectura
        try {
            System.out.println(Thread.currentThread().getName() + " está leyendo: " + datos);
        } finally {
            lock.readLock().unlock();  // Liberar el bloqueo de lectura
        }
    }
    public void escribir(int valor) {
        lock.writeLock().lock();  // Adquirir el bloqueo de escritura
        try {
            System.out.println(Thread.currentThread().getName() + " está escribiendo: " + valor);
            datos = valor;
        } finally {
            lock.writeLock().unlock();  // Liberar el bloqueo de escritura
        }
    }
}
```

Bloqueos Explícitos: Condition

En Java, la gestión de señales entre hilos se puede realizar mediante la interfaz **Condition** del paquete java.util.concurrent.locks. Esta interfaz proporciona un mecanismo más flexible y poderoso que los métodos tradicionales de *wait()*, *notify()*, y *notifyAll()*.

```
private final Lock bloqueo = new ReentrantLock();
private final Condition condicion = bloqueo.newCondition();
```

Bloqueos Explícitos: Condition

Una **Condition** se utiliza en combinación con un Lock para coordinar la comunicación entre hilos. Funciona como una cola de espera para hilos que están bloqueados esperando que ocurra alguna condición particular. Los métodos principales que proporciona son:

Método	Descripción
await()	Pone al hilo actual en espera hasta que se le notifique que puede continuar. Es similar a wait() en términos de funcionalidad.
signal()	Despierta un hilo que esté esperando en la condición. Es equivalente a notify(), pero más controlado.
signalAll()	Despierta a todos los hilos que estén esperando en la condición, lo que es equivalente a notifyAll().

Bloqueos Explícitos: Condition

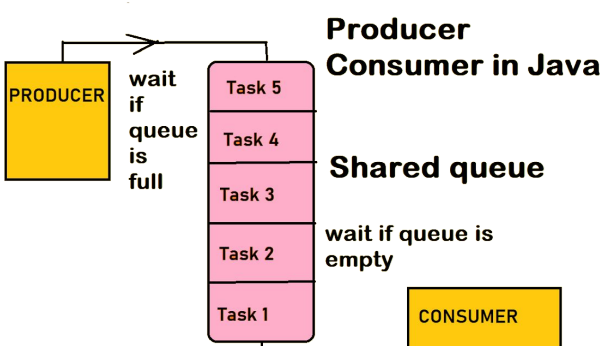
```
class Cola {
    private int contenido;
    private boolean disponible = false;
    private final Lock bloqueo = new ReentrantLock();
    private final Condition condicion = bloqueo.newCondition();

    public void producir(int valor) throws InterruptedException {
        bloqueo.lock();  // Adquirir el lock
        try {
            while (disponible) {
                condicion.await();  // Esperar hasta que el contenido sea consumido
            }
            contenido = valor;
            disponible = true;
            System.out.println("Producido: " + contenido);
            condicion.signal();  // Notificar al consumidor
        } finally {
            bloqueo.unlock();  // Liberar el lock
        }
    }
```

Bloqueos Explícitos: Condition

```
    public int consumir() throws InterruptedException {
        bloqueo.lock();
        try {
            while (!disponible) {
                condicion.await();  // Esperar hasta que haya contenido disponible
            }
            disponible = false;
            System.out.println("Consumido: " + contenido);
            condicion.signal();  // Notificar al productor
            return contenido;
        } finally {
            bloqueo.unlock();
        }
    }
}
```

Bloqueos Explícitos: Condition

- Un productor crea valores y los pone en una cola, mientras que un consumidor los toma. Ambos hilos usan un Lock y una Condition para coordinar sus acciones, asegurando que no haya interferencias.
- El productor espera hasta que el consumidor haya tomado el valor antes de producir un nuevo elemento, y viceversa.
- *await()* pone el hilo en espera de manera eficiente, liberando el lock, y signal() despierta al hilo que estaba esperando para reanudar su trabajo.



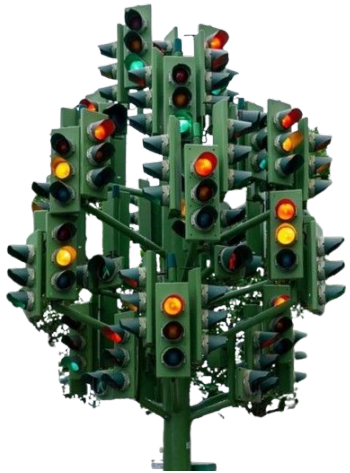
Bloqueos Explícitos: Condition

Comparación Condition vs wait/notify

Característica	wait/notify	Condition
Asociación	Con el monitor del objeto	Con un Lock
Número de condiciones	Una única cola de espera	Múltiples condiciones con Condition
Notificación	*notify()* y *notifyAll()*	*signal()* y *signalAll()*
Uso común	Simple y básico para sincronización básica	Más flexible para sincronización avanzada

Bloqueos Avanzados: Semáforos

Los **semáforos** en Java son herramientas de sincronización que limitan la cantidad de hilos que pueden acceder a un recurso compartido a la vez. Son útiles cuando se quiere controlar el acceso a un número limitado de recursos, como conexiones a una base de datos o acceso a un pool de hilos.



Bloqueos Avanzados: Semáforos

Semaphore

La clase *Semaphore* de Java del paquete java.util.concurrent funciona como un contador que regula cuántos hilos pueden acceder a una sección crítica o a un recurso.

- Un semáforo tiene un número limitado de permisos. Cada vez que un hilo quiere acceder a un recurso, debe adquirir un permiso (*acquire()*), y una vez que ha terminado, debe devolver el permiso (*release()*).
- Un semáforo con un solo permiso es un semáforo binario, que funciona como un bloqueo (similar a un *Lock*).
- Al instanciar un semáforo se especifica el número de permisos disponibles que controlan cuántos hilos pueden acceder simultáneamente a un recurso.

```
Semaphore semaforo = new Semaphore(int permisos);
```

Bloqueos Avanzados: Semáforos

Semaphore

Además de definir el número de permisos, también puedes decidir si quieres que el semáforo sea justo o no justo.

- Si un semáforo es justo, los permisos se otorgan a los hilos en el orden en que fueron solicitados (FIFO)
- En el modo no justo, el orden es indefinido y depende del sistema.

```
Semaphore semaforoJusto = new Semaphore(int permisos, boolean isFair);
```

Bloqueos Avanzados: Semáforos

Semaphore

Método	Descripción
acquire()	Intenta adquirir un permiso. Si no hay permisos disponibles, el hilo se bloquea hasta que uno esté disponible.
release()	Libera un permiso, permitiendo que otro hilo lo adquiera.
tryAcquire()	Intenta adquirir un permiso de manera no bloqueante, devolviendo *true* si tuvo éxito, y *false* si no.
availablePermits()	Devuelve la cantidad de permisos disponibles.

Bloqueos Avanzados: Semáforos

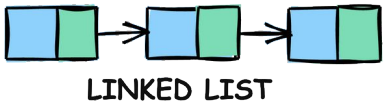
Semaphore

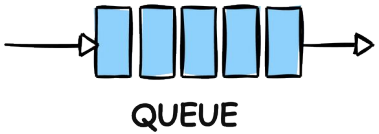
```
class Estacionamiento {
    private Semaphore plazas;

    public Estacionamiento(int numPlazas) {
        plazas = new Semaphore(numPlazas);
    }

    public void entrar(String coche) {
        try {
            System.out.println(coche + " está esperando para entrar.");
            plazas.acquire();  // Adquiere un permiso si hay espacio
            System.out.println(coche + " ha entrado.");
            Thread.sleep(2000);  // Simula el tiempo que el coche está estacionado
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            System.out.println(coche + " ha salido.");
            plazas.release();  // Libera el permiso al salir
        }
    }
}
```

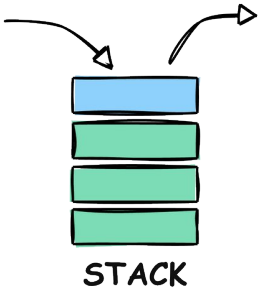
Estructuras de Datos Concurrentes

Las **estructuras de datos concurrentes** son colecciones o estructuras diseñadas específicamente para ser utilizadas de manera segura y eficiente en entornos multihilo. Garantizan que los datos no se corrompan ni haya condiciones de carrera, todo sin necesidad de que el programador implemente la sincronización manualmente.

- Acceso seguro
- Eficiencia
- Uso de bloqueos granulares o no bloqueantes: Algunas utilizan bloqueos finos (en lugar de bloquear toda la estructura), mientras que otras emplean técnicas sin bloqueos, como el algoritmo CAS







Estructuras de Datos Concurrentes

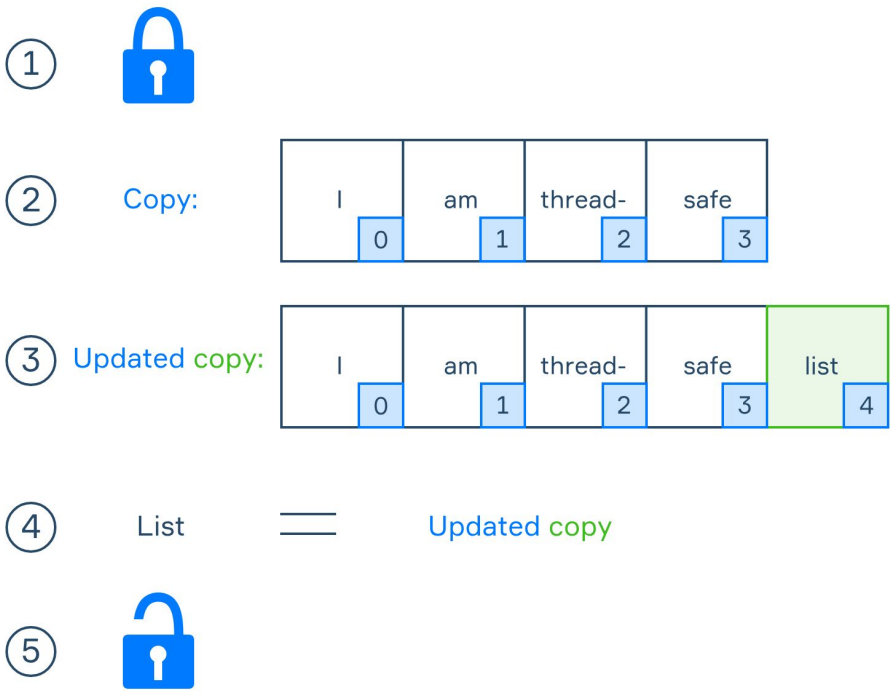
CopyOnWriteArrayList

Una lista donde las operaciones de escritura (*add*, *remove*, *etc*.) crean una nueva copia de la lista para evitar conflictos, mientras que las operaciones de lectura pueden suceder simultáneamente sin bloqueo.

```
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
list.add("elemento1");
```

Estructuras de Datos Concurrentes

CopyOnWriteArrayList



Estructuras de Datos Concurrentes

ConcurrentLinkedQueue

Una cola no bloqueante basada en nodos enlazados que usa una estructura sin bloqueos (lock-free), lo que la hace ideal para situaciones de alta concurrencia.

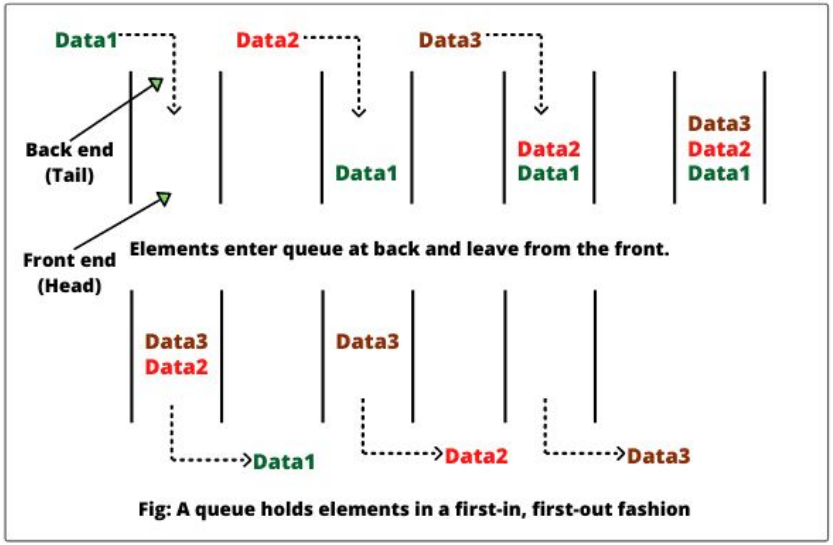


Fig: A queue holds elements in a first-in, first-out fashion

Estructuras de Datos Concurrentes

BlockingQueue

Una cola que puede bloquear las operaciones de inserción o extracción hasta que la operación sea posible. Útil en patrones como productor-consumidor, donde el productor espera si la cola está llena y el consumidor espera si la cola está vacía.

- ArrayBlockingQueue: Cola fija con bloqueo.
- LinkedBlockingQueue: Cola enlazada con bloqueo.
- PriorityBlockingQueue: Cola de prioridad con bloqueo.

Estructuras de Datos Concurrentes

BlockingQueue

Una cola que puede bloquear las operaciones de inserción o extracción hasta que la operación sea posible. Útil en patrones como productor-consumidor, donde el productor espera si la cola está llena y el consumidor espera si la cola está vacía.

- ArrayBlockingQueue: Cola fija con bloqueo.
- LinkedBlockingQueue: Cola enlazada con bloqueo.
- PriorityBlockingQueue: Cola de prioridad con bloqueo.

Estructuras de Datos Concurrentes

ConcurrentHashMap

Una versión concurrente del *HashMap* que permite acceso simultáneo a múltiples hilos sin necesidad de sincronización externa. Se divide en segmentos, lo que permite que varios hilos lean y escriban en diferentes partes del mapa a la vez.