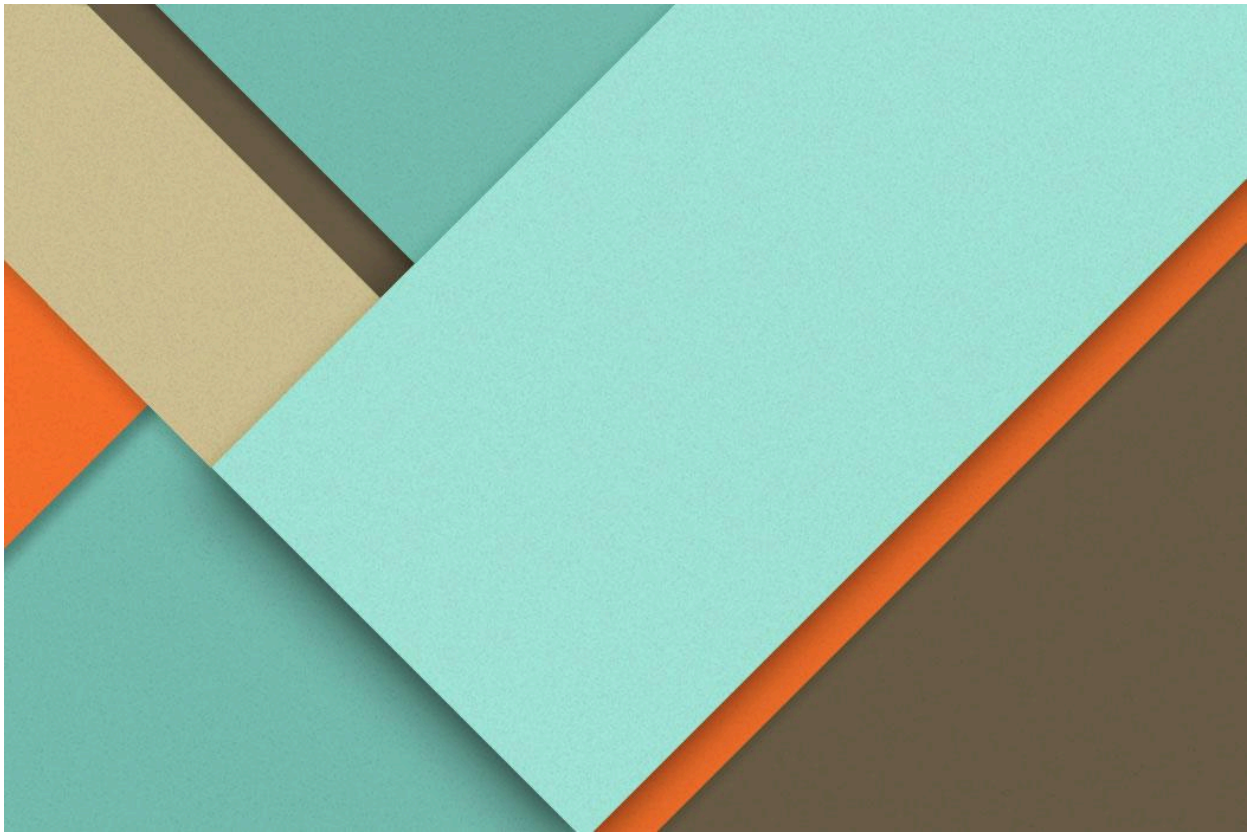

A3- Actividad Hilos

05/09/2025

Ester Cubero Chaves

Daniel Beltrán Ruíz

José Manuel Sánchez Rosal

2 DAM

IES Antonio Gala

14700 Palma del Río (Córdoba)

Visión general

Investigar, demostrar y resolver problemas reales de CONCURRENCIA relacionados con:

- **Exclusión mutua**: (control de accesos a recursos compartidos).
- **Interbloqueo (deadlock)**: cómo reproducirlo, detectarlo y evitarlo.

Objetivos

Cada grupo presentará una demo ejecutable, una explicación técnica y una breve presentación.

1- Investigación

A continuación explicaremos los conceptos clave para poder entender esta actividad.

1.1- Exclusión mutua

Es un concepto fundamental en la **programación concurrente** que se utiliza para **evitar que múltiples hilos** o procesos **accedan simultáneamente a un recurso compartido**, como una **variable**, un **archivo** o una **sección crítica de código**.

1.2- Interbloqueo o Deadlock

El interbloqueo o deadlock **ocurre cuando 2 o más hilos o procesos se bloquean mutuamente**, cada uno **esperando que el otro libre un recurso que ambos necesitan para continuar su ejecución**. Esto crea un ciclo en el que ninguno de los hilos puede avanzar.

¿Cómo ocurre un interbloqueo? Para que ocurra un deadlock, se deben cumplir las siguientes condiciones:

- Exclusión mutua: Un recurso solo puede ser utilizado por un hilo a la vez.
- Retención y espera: Un hilo tiene un recurso y está esperando obtener otro recurso que está retenido por otro hilo.
- No apropiación: Los recursos no se pueden forzar a liberarse; solo el hilo que los retiene puede liberarlos.
- Esperas circulares: Se produce un ciclo de hilos, donde cada hilo está esperando un recurso que otro hilo del ciclo tiene.

Para evitar el interbloqueo, si se garantiza que todos los hilos adquieran los bloqueos en el mismo orden es una de las formas más sencillas de evitar el deadlock.

Evitar la Retención y Espera, un hilo debería liberar los bloqueos que ya ha adquirido si no puede obtener todos los recursos que necesita.

Tiempo de Espera y Reintentos, se implementa un límite de tiempo para que los hilos intenten adquirir un bloqueo. Si el bloqueo no se puede adquirir dentro de ese tiempo, el hilo puede intentar nuevamente o liberar los recursos que tiene.

Detección de Deadlock, para ello monitorizar el estado de los hilos y detectar si están bloqueados en un ciclo de espera. Si se detecta un deadlock, se pueden tomar medidas correctivas como terminar un proceso, reinicio de la aplicación en un Sistema Java, etc...

1.3- Mecanismos en Java para la exclusión mutua e interbloqueos

En JAVA, se usan métodos o bloques con la palabra **"syncronized"**. Éstos sirven para **limitar el acceso a un recurso compartido** por dos o más hilos o procesos. De esta manera, **sólo uno de ellos accede** al recurso compartido (fragmento de código sensible, variables, etc...) **al mismo tiempo**. La **diferencia entre método y bloque con syncronized**:

- **Método**: se protege todo el código que tengamos en el mismo, ejemplo:
    - public void syncronized producir() {
        - Todo el código protegido

      }
- **Bloque**: se protege sólo el fragmento de código de alcance de ese bloque syncronizer:
    - public void producir(){
        - syncronized(this){
            - código protegido}

      Resto de código sin proteger

      }

Dentro del bloque syncronized se usan las siguientes funciones para ponerse de acuerdo a la hora de acceder a nuestra zona crítica de código:

- **wait()**: **El hilo actual se suspende** y libera el candado del monitor o recurso compartido para que otro pueda acceder al mismo. **Este espera** hasta que otro hilo lo despierte con notify().
- **wait(long timeout)**: igual que wait() pero con un tiempo máximo de espera en ms.
- **wait(long timeout, int nanos)**: versión más precisa indicando milisegundos y nanosegundos.
- **notify()**: Despierta sólo un hilo que esté esperando con wait() sobre el mismo monitor o recurso.
- **notifyAll()**: Despierta todos los hilos que estén esperando con wait() sobre el mismo monitor o recurso (sólo uno podrá entrar al monitor, los demás seguirán esperando).

Otro método para evitar interbloqueos es el uso de la interfaz LOCK. El ReentrantLock es su implementación más común.

Son mecanismos de control de interbloqueos más avanzados que nos permiten bloquear y desbloquear recursos compartidos:

- **lock()**: el hilo actual intenta adquirir el lock o bloqueo de un recurso compartido, si otro hilo lo tiene, significa que está bloqueado hasta que se libere. El hilo actual queda a la espera hasta que consiga el lock del monitor o recurso (no responde a interrupt().
- **unlock()**: libera el bloqueo del recurso compartido, únicamente el hilo dueño del recurso en este momento puede liberar el mismo. Si otro hilo lanza el unlock(), se lanza una excepción (IlegalMonitorStageException).
- **reentrancia**: definición: cuando el hilo adquiere varias veces el lock, incrementa un contador interno y deberá llamar a unlock el mismo número de veces para poder liberarlo.
- **tryLock()**: el hilo intenta adquirir de forma inmediata el lock del recurso, devuelve TRUE si lo consiguió y FALSE si no (porque otro hilo posee el lock del recurso). Usaremos el tryLock() cuando queramos evitar el deadlock
- **tryLock(long time, TimeUnit unit)**: intenta adquirir el lock del monitor durante el tiempo que le digamos.
- **newCondition()**; crea un **Condition** asociado al Lock del recurso. Es el equivalente más sofisticado del wait() y notify(). Un **Condition sirve para coordinar hilos:**
    - **await()**: el hilo que llama a await() se suspende y libera el lock temporalmente. Queda esperando hasta que otro hilo lo despierte con signal() o signalAll().
    - **await(long time, TimeUnit unit)**: igual que el await pero con un tiempo de espera máximo indicado.
    - **signal()**: despierta un solo hilo que esté esperando en esa Condition. Todos compiten por el lock y uno a uno entra a la sección crítica.
    - Siempre se debe tener el lock cuando se llame a await(), signal() o signalAll().
    - Se puede tener notFull (esperando productores) o notEmpty (esperando consumidores.

2. Elección de Caso

“El Problema del barbero Durmiente”

El problema del barbero durmiente es un problema de sincronización en ciencias de la computación, éste describe un escenario de barbería:

Un barbero con un sillón y sillas de espera; si no hay clientes, el barbero duerme; si llega un cliente, éste lo despierta y espera en una de ellas; si llegan más clientes y hay sillas libres se sientan, si no hay sillas libres se van.

La barbería tiene los siguientes elementos y reglas:

- **Un barbero**: que solo puede atender a un cliente a la vez.
- **Un sillón**: para el corte de pelo.
- **Un número determinado de sillas**: en la sala de espera.

Cuando llega un cliente:

- Si el barbero está dormido, el cliente lo despierta.
- Si hay sillas libres, el cliente se sienta y espera su turno.
- Si todas las sillas de espera están ocupadas, el cliente se va de la barbería.

El problema surge en la gestión de la concurrencia, pues el barbero y los clientes son procesos diferentes que pueden ejecutarse simultáneamente, llegando en momentos aleatorios.

Por otro lado, el sillón del barbero y las sillas son recursos compartidos y como tal, se deben gestionar para que sólo un proceso acceda a los mismos. Para ello hay que sincronizar las acciones del barbero y de los clientes para que no entren en conflicto entre sí a la hora de acceder al monitor o recurso compartido.

3. Demo Deadlock

```java
public class DemoDeadlock {  new *
    public static void main(String[] args) {  new *
        Sillon sillon = new Sillon();

        // El primer cliente intenta acceder a una silla y luego al sillón del barbero
        Thread primCliente = new Thread(new Runnable() {  new *
            @Override  new *
            public void run() {
                //el método synchronized controla el acceso a un recurso compartido con varios hilos --> es un bloqueo
                synchronized (sillon.getSillas()) {
                    System.out.println("El primer cliente ha ocupado las sillas de espera");
                    try {
                        //El hilo espera 100ms para ejecutarse
                        Thread.sleep( millis: 100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("El primer cliente ha ocupado el sillón del barbero");
                    //con synchronized sillon se pone un bloqueo al sillon
                    //al estar los dos synchronized en el mismo hilo, se produce el bloqueo al estar los
                    //dos candados en distinto orden
                    synchronized (sillon) {
                        System.out.println("El primer cliente está siendo atendido");
                    }
                }
            }
        });
```

```java
        // Un segundo cliente intenta acceder al sillón del barbero y luego a las sillas
        Thread clienteB = new Thread(new Runnable() {  new *
            @Override  new *
            public void run() {
                synchronized (sillon) {
                    System.out.println("El segundo cliente ha accedido al sillón del barbero");
                    try {
                        Thread.sleep( millis: 100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("El segundo cliente ha accedido a las sillas de espera");
                    synchronized (sillon.getSillas()) {
                        System.out.println("El segundo cliente está ocupando las sillas de espera");
                    }
                }
            }
        });

        // Iniciamos los hilos
        primCliente.start();
        clienteB.start();

        // Aunque iniciemos al barbero y al cliente reales, los sillones están ocupados
        // Cliente A y B están esperando a usar recursos del otro, asi ninguno avanza en el proceso. Eso es un deadlock.
        Barbero barbero = new Barbero(sillon);
        Cliente cliente = new Cliente(sillon);
        barbero.start();
        cliente.start();
    }
}
```

4. Uso Monitores (Syncronized)

4.1 Class Barbero

```java
// Clase barbero que extiende de HILO
public class Barbero extends Thread {  2 usages
    private Sillon sillon;  2 usages

    public Barbero(Sillon sillon) { this.sillon = sillon; }
    @Override
    public void run(){
        // Hacemos bucle infinito para que trabaje el barbero (24/7/365)
        while(true){
            try {
                sillon.CortarPelo();
                Thread.sleep( millis: 2000);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

4.2 Class Cliente:

```java
// Clase Cliente que extiende de HILO
public class Cliente extends Thread{
    private Sillon sillon;  2 usages

    public Cliente(Sillon sillon) { this.sillon = sillon; }
    @Override
    public void run(){
        // Enviamos a 7 clientes al sillón:
        for (int i = 0; i <= 6; i++){
            try {
                // Esperamos un segundo entre cliente y cliente:
                Thread.sleep( millis: 1000);
                sillon.SentarseSillon();
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

4.3 Class Main:

```java
public class Main {
    public static void main(String[] args) {
        // Instanciamos la clase sillón para pasarla por parámetro como RECURSO COMPARTIDO a los 2 procesos
        Sillon sillon = new Sillon();
        Barbero b1 = new Barbero(sillon);
        Cliente c1 = new Cliente(sillon);
        // Lanzamos los hilos....
        c1.start();
        b1.start();
    }
}
```

4.4. Class Sillón:

```java
// El sillon es el recurso compartido (también llamado monitor).
public class Sillon {  6 usages

    // Declaramos el ArrayList de sillas disponibles en nuestra Barberia y una variable de control de barbería vacía.
    private ArrayList<Integer> sillas = new ArrayList<>();  13 usages
    private boolean barberVacia = true;  2 usages

    // Getters y Setters
    public ArrayList<Integer> getSillas() {  no usages
        return sillas;
    }
    public void setSillas(ArrayList<Integer> sillas) {  no usages
        this.sillas = sillas;
    }

    // Bloque  para Cortar Pelo
    public synchronized void CortarPelo() throws InterruptedException {  1 usage
        // Si no hay clientes....el barbero se queda dormido:
        while (sillas.size() == 0) {
            System.out.println("No hay clientes, el barbero sigue durmiendo...");
            wait();
        }
```

```
        // Si entra un cliente y hay sitio se sienta y el barbero lo atiende:
        System.out.println("Barbero atiende al cliente " + sillas.get(0) + " y se marcha...");
        // Simulamos el tiempo de corte de pelo:
        Thread.sleep( millis: 2000);
        // Cliente satisfecho se va.....
        sillas.remove( index: 0);
        // Mostramos sillas ocupadas:
        System.out.print("Sillas ocupadas: " + sillas.size() + "     ");
        System.out.println(sillas);
        System.out.println("----------------------------------------------------------------");
        notifyAll();
        // Si se queda el sillón y la pelu vacías.....el barbero se queda dormido:
        if (barberVacia == true && sillas.size() == 0) {
            System.out.println("El barbero se queda dormido...");
        }
    }

    // Bloque syncronized para Sentarse en Sillón:
    public synchronized void SentarseSillon() throws InterruptedException {  1 usage
        while (sillas.size() == 3) {
            System.out.println("Llega un cliente > no hay sillas disponibles > Y SE MARCHA");
            wait();
        }
```

```
        // Si hay sitio libre, el cliente se sienta...
        // Vamos creando numeros y los asociamos a los clientes que van entrando:
        Random random = new Random();
        int cliente = random.nextInt( bound: 50);
        sillas.add(cliente);
        // Sacamos por pantalla el cliente que llega y se sienta:
        System.out.println("Llega el cliente " + cliente + " y SE SIENTA");
        // Al sentarse el cliente, mostramos las sillas ocupadas con el número de cliente:
        System.out.print("Sillas ocupadas: " + sillas.size() + "     ");
        System.out.println(sillas);
        System.out.println("----------------------------------------------------------------");
        //PARA REPRODUCIR UN DEADLOCK >> BORRAR EL NOTIFY.
        notifyAll();
    }
    // ToString:
    @Override
    public String toString() {
        return "Sillon{" +
                "sillas=" + sillas +
                ", peluVacia=" + barberVacia +
                '}';
    }
}
```

5. Uso semáforo (Semaphore)

5.1 Class Barbero

```java
class Barberoo extends Thread { //este hilo accede al monitor de las sillas  2 usages
    // y los semaforos (la variable sillon)
    // funciona a la vez que el hilo de los clientes
    private SillonBarberoSemaforo sillon;  2 usages

    //la variable sillon sirve para que el barbero pueda acceder al metodo de cortar el pelo
    public Barberoo(SillonBarberoSemaforo sillon) { this.sillon = sillon; }

    @Override
    public void run() {
        try {
            while (true) {
                sillon.cortarPelo();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

5.2 Class Cliente

```java
class Clientee extends Thread {  2 usages
    //este hilo de cliente espera un tiempo aleatorio antes de sentarse en el sillon
    private SillonBarberoSemaforo sillon;  2 usages

    public Clientee(SillonBarberoSemaforo sillon) { this.sillon = sillon; }
    @Override
    public void run() {
        try {
            Random random = new Random();
            int tiempoEspera = random.nextInt( bound: 5000);
            Thread.sleep(tiempoEspera);
            sillon.sentarseSillon();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

5.3 Class Sillón

```java
// La clase semáforo se importa con java.util.concurrent.Semaphore; (es un objeto como podría ser ArrayList)
public class SillonBarberoSemaforo {
    private final int NUM_SILLAS = 3;  1 usage
    private int sillasDisponibles = NUM_SILLAS;  5 usages
    private Semaphore clientes = new Semaphore( permits: 0);  // semaforo para clientes esperando  2 usages
    private Semaphore barbero = new Semaphore( permits: 0);   // para barbero despierto o no  2 usages
    private Semaphore mutex = new Semaphore( permits: 1);     // para exclusión mutua=mutex  5 usages
    private int contadorClientes = 0;  2 usages

    //Método para cortar el pelo:
    public void cortarPelo() throws InterruptedException {  1 usage
        clientes.acquire(); //espera a que haya un cliente
        mutex.acquire(); //accede a la *sección crítica* - semaforo en rojo
        sillasDisponibles++;
        System.out.println("El barbero está cortando el pelo. Sillas libres: " + sillasDisponibles);
        barbero.release(); //accede al cliente en espera
        mutex.release();//sale de esa sección crítica - semaforo en verde
        Thread.sleep( millis: 2000); //2000ms es lo que tarda en hacer el servicio
        System.out.println("Cortando el pelo...Corte hecho\n");
    }
```

```
    //Método para espera:
    public void sentarseSillon() throws InterruptedException {  1 usage
        mutex.acquire();//accede a la sección crítica
        contadorClientes++;
        int id = contadorClientes;
        if (sillasDisponibles > 0) {
            sillasDisponibles--;
            System.out.println("El cliente " + id + " pasa. Sillas libres: " + sillasDisponibles);
            clientes.release(); //un cliente más
            mutex.release(); //semaforo en verde: deja sitio libre
            barbero.acquire(); //entonces el barbero atiende
            System.out.println("El cliente " + id + " se está pelando");
            Thread.sleep( millis: 1500); //atiende durante ese tiempo
        } else {
            System.out.println("Está el cliente " + id + " pero no hay sillas libres. (Se va).");
            mutex.release();
        }
    }

    public static void main(String[] args) {
        SillonBarberoSemaforo sillon = new SillonBarberoSemaforo();
        Barberoo barbero = new Barberoo(sillon);
        barbero.start();
        for (int i = 0; i < 10; i++) {
            Clientee cliente = new Clientee(sillon);
            cliente.start();
            try {
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

4. Bibliografía

IBM i. (2025, septiembre 29). Ibm.com. https://www.ibm.com/docs/es/i/7.5.0?topic=threads-mutexes

Nivardo. (2024, septiembre 11). Deadlock entre Hilos en Java. Oregoom.com; Oregoom. https://oregoom.com/java/deadlock-entre-hilos/