

CCPD

Tema U2: Concurrency

LAB 2: Sincronización y Mecanismos de Coordinación de Hilos

2024/2025

David Olivieri
Leandro Rodríguez Liñares
Uvigo, E.S. Informática

lab2prob01

```
class TransThread extends Thread {
    private FinTrans ft;

    TransThread (FinTrans ft, String name) {
        super (name);    // Save thread's name
        this.ft = ft;    // Save reference to financial transaction object
    }

    public void run () {
        for (int i = 0; i < 100; i++) {
            if (getName ().equals ("Deposit Thread")){
                // Start of deposit thread's critical code section
                ft.transName = "Deposit";
                try {
                    Thread.sleep ((int) (Math.random () * 1000));
                } catch (InterruptedException e) {}

                ft.amount = 2000.0;
                System.out.println (ft.transName + " " + ft.amount);
                // End of deposit thread's critical code section
            } else {
                // Start of withdrawal thread's critical code section
                ft.transName = "Withdrawal";
                try {
                    Thread.sleep ((int) (Math.random () * 1000));
                } catch (InterruptedException e) {}
                ft.amount = 250.0;
                System.out.println (ft.transName + " " + ft.amount);
                // End of withdrawal thread's critical code section
            }
        }
    }
}
```

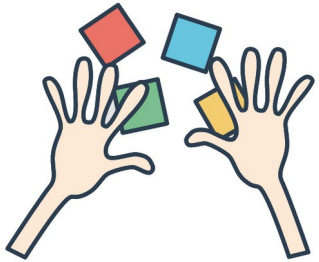
```
class FinTrans{
    public static String transName;
    public static double amount;
}
```

```
class lab2prob01 {
    public static void main (String [] args) {
        FinTrans ft = new FinTrans ();
        TransThread tt1 = new TransThread (ft, "Deposit");
        TransThread tt2 = new TransThread (ft, "Withdrawal");
        tt1.start ();
        tt2.start ();
    }
}
```

Manos-a-la-obra: Lab2Prog01

Objetivo:

- Analizar el comportamiento de un programa multihilo sin sincronización e identificar problemas de concurrencia.



1. **Ejecuta el código proporcionado** y observa su comportamiento.
2. **Explica el comportamiento** con tus propias palabras. Describe lo que ocurre cuando el programa se ejecuta varias veces. ¿El resultado es siempre el mismo?
3. **Identifica y analiza los problemas de concurrencia.**
 - Documenta varias ejecuciones del programa.
 - Identifica inconsistencias en la salida.
 - Explica por qué ocurren estas inconsistencias, centrándote en cómo los hilos acceden a los recursos compartidos sin sincronización.
4. **Propón una hipótesis** sobre cómo la sincronización podría solucionar estos problemas.

lab2prob02

```
class TransThread extends Thread {
    private FinTrans ft;

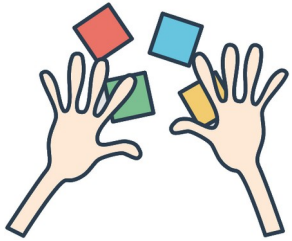
    TransThread(FinTrans ft, String name) {
        super(name); // Guardar el nombre del hilo
        this.ft = ft; // Guardar referencia al objeto de
                       // transacción financiera
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            /*
             *      AÑADIR CÓDIGO:
             *      Escribir las sentencias de control y
             *      los bloques sincronizados para proteger
             *      las secciones críticas de los hilos de
             *      Depósito y Retiro.
             */
        }
    }
}
```

```
class FinTrans {
    public static String transName; // Nombre de la transacción
    public static double amount; // Monto de la transacción
}
```

```
class lab2prob02 {
    public static void main(String[] args) {
        FinTrans ft = new FinTrans();
        TransThread tt1 = new TransThread(ft, "Depósito"); // Hilo de depósito
        TransThread tt2 = new TransThread(ft, "Retiro"); // Hilo de retiro
        tt1.start();
        tt2.start();
    }
}
```

Manos-a-la-obra: lab2prob02



Objetivo:

- Modificar el código dado para sincronizar correctamente los hilos de Depósito y Retiro, asegurando un acceso seguro a los recursos compartidos.

1. **Modifica el código** añadiendo los mecanismos de sincronización necesarios (por ejemplo, bloques o métodos synchronized).
2. **Asegura la seguridad de los hilos** para evitar que los hilos de Depósito y Retiro interfieran entre sí al acceder a recursos compartidos.
3. **Ejecuta el programa modificado varias veces** y observa su comportamiento.
4. **Documenta tus resultados:**
 - Explica cómo el código añadido garantiza la sincronización.
 - Describe la salida del programa y si permanece consistente en varias ejecuciones.
 - Compara los resultados con los del ejercicio anterior (sin sincronización).
 - Identifica cualquier problema restante o casos límite en tu implementación.

lab2prob03

```
class TransThread extends Thread {
    private FinTrans ft;
    private Object lock; // Objeto de bloqueo compartido entre los hilos

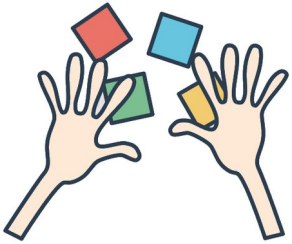
    TransThread(FinTrans ft, String name, Object lock) {
        super(name);
        this.ft = ft;
        this.lock = lock; // Usar un objeto de bloqueo compartido
    }

    public void run() {
        for (int i = 0; i < 100; i++) {
            /* AÑADIR CÓDIGO:
             *      Escribir las sentencias de control y
             *      sincronizar la sección crítica con
             *      "lock" para proteger los hilos de
             *      Depósito y Retiro.
             */
        }
    }
}
```

```
class FinTrans {
    public static String transName; // Nombre de la transacción
    public static double amount; // Monto de la transacción
}
```

```
class lab2prob03 {
    public static void main(String[] args) {
        FinTrans ft = new FinTrans();
        // Objeto de bloqueo compartido
        Object lock = new Object();
        TransThread tt1 = new TransThread(ft, "Hilo de Depósito", lock);
        TransThread tt2 = new TransThread(ft, "Hilo de Retiro", lock);
        tt1.start();
        tt2.start();
    }
}
```

Manos-a-la-obra: lab2prob03



Objetivo:

- Modificar el código dado para sincronizar correctamente los hilos de Depósito y Retiro, asegurando un acceso controlado a los datos de transacción compartidos.

1. **Añade el código** necesario para la sincronización

- Usa el objeto lock proporcionado para garantizar la ejecución segura de los hilos.
- Protege la sección crítica donde se modifican los datos de la transacción (transName y amount).

2. **Demuestra** que los datos compartidos funcionan correctamente

- Ejecuta el programa varias veces y verifica que no haya inconsistencias.
- Los hilos de Depósito y Retiro deben operar sin corromper los datos de la transacción.

3. **Elimina la sincronización y observa las inconsistencias**

- Comenta el código de sincronización y vuelve a ejecutar el programa.
- Documenta cualquier comportamiento inesperado o condiciones de carrera que ocurran.

4. **Explica** tus hallazgos

- Describe cómo la sincronización previene problemas.
- Compara los resultados con y sin sincronización.

lab2prob04

```
class CounterThread extends Thread {
    private SharedCounter sharedCounter;

    public CounterThread(SharedCounter sharedCounter) {
        this.sharedCounter = sharedCounter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            sharedCounter.increment();
        }
    }
}
```

```
class SharedCounter {
    private int count = 0;

    /*
     * AÑADIR: un método increment() sincronizado
     */

    public int getCount() {
        return count;
    }
}
```

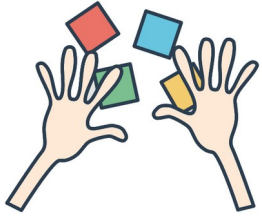
```
public class lab2prob04 {
    public static void main(String[] args) throws InterruptedException {
        SharedCounter sharedCounter = new SharedCounter();
        Thread[] threads = new Thread[10];

        // Crear y lanzar hilos
        for (int i = 0; i < threads.length; i++) {
            /*
             * AÑADIR CÓDIGO:
             *
             * Iniciar los hilos con el objeto compartido
             */
        }

        // Esperar a que todos los hilos finalicen
        for (Thread thread : threads) {
            thread.join();
        }

        System.out.println("Valor final del contador: " + sharedCounter.getCount());
    }
}
```


Manos-a-la-obra: lab2prob04



Objetivo:

- Modificar el código dado para sincronizar correctamente el contador compartido, asegurando que varios hilos puedan actualizarlo sin condiciones de carrera.

1. **Añade un método `increment()`** sincronizado en la clase `SharedCounter` para garantizar actualizaciones seguras por los hilos.
2. **Crea e inicia 10 hilos** que incrementen el contador compartido en paralelo.
3. **Espera a que todos los hilos finalicen** su ejecución utilizando `.join()`.
4. **Observa y documenta los resultados:**
 - Ejecuta el programa varias veces y verifica si el valor final del contador es siempre correcto.
 - Elimina la sincronización del método `increment()` y documenta las inconsistencias.
 - Explica por qué ocurren condiciones de carrera y cómo la sincronización las soluciona.

lab2prob05

Sincronización y Bloqueo de Objetos

```

public class SharedResource {
    /*
        AÑADIR CÓDIGO: Un bloque o método sincronizado
        methodOne()
        1. Imprime el nombre del hilo actual
        2. Duerme durante 2000 ms
        3. Imprime un mensaje indicando que el hilo actual ha terminado
    */

    /*
        AÑADIR CÓDIGO: Un bloque o método sincronizado
        methodTwo()
        1. Imprime el nombre del hilo actual
        2. Duerme durante 2000 ms
        3. Imprime un mensaje indicando que el hilo actual ha terminado
    */
}

```

```

public class MyTaskThread extends Thread {
    private SharedResource sharedResource;
    private boolean runFirstMethod;

    public MyTaskThread(SharedResource sharedResource, boolean runFirstMethod) {
        this.sharedResource = sharedResource;
        this.runFirstMethod = runFirstMethod;
    }

    @Override
    public void run() {
        if (runFirstMethod) {
            sharedResource.methodOne();
        } else {
            sharedResource.methodTwo();
        }
    }
}

```

```

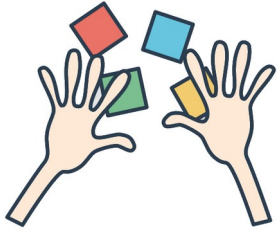
public class lab2prob05 {
    public static void main(String[] args) {
        SharedResource sharedResource = new SharedResource();

        MyTaskThread thread1 = new MyTaskThread(sharedResource, true);
        MyTaskThread thread2 = new MyTaskThread(sharedResource, false);

        thread1.start();
        thread2.start();
    }
}

```

Manos-a-la-obra: lab2prob05



Objetivo:

- Modificar el código dado para sincronizar correctamente el recurso compartido, asegurando que varios hilos puedan acceder a él sin condiciones de carrera.

1. **Añade el código necesario** para la sincronización

- Implementa bloques o métodos sincronizados en `SharedResource` para proteger `methodOne()` y `methodTwo()`.
- Asegura que solo un hilo pueda ejecutar una sección crítica a la vez.

2. **Demuestra la corrección de la sincronización**

- Ejecuta el programa con y sin sincronización.
- Observa y documenta cualquier inconsistencia (condiciones de carrera) cuando la sincronización está ausente.
- Compara los resultados y explica cómo la sincronización soluciona estos problemas.

3. **Generaliza la solución** para múltiples hilos

- Modifica el código para manejar un mayor número de hilos dinámicamente en lugar de solo dos.
- Implementa un bucle en `main()` para crear e iniciar múltiples instancias de `MyTaskThread`.

4. **Documenta tus hallazgos**

- Explica cómo la sincronización mejora la gestión de hilos.
- Describe cómo tu solución generalizada escala con múltiples hilos.

Lab2prob06:

Bloques Sincronizados vs. Métodos Sincronizados

```

class ListOperationThread extends Thread {
    private Object list;
    private boolean useMethod;

    public ListOperationThread(Object list, boolean useMethod) {
        this.list = list;
        this.useMethod = useMethod;
    }

    @Override
    public void run() {
        if (useMethod && list instanceof SynchronizedMethodList) {
            SynchronizedMethodList methodList = (SynchronizedMethodList) list;
            methodList.addElement(1);
            methodList.getElement(0);
        } else if (!useMethod && list instanceof SynchronizedBlockList) {
            SynchronizedBlockList blockList = (SynchronizedBlockList) list;
            blockList.addElement(1);
            blockList.getElement(0);
        }
    }
}

```

```

public class lab2prob06 {
    public static void main(String[] args) {
        SynchronizedMethodList methodList = new SynchronizedMethodList();
        SynchronizedBlockList blockList = new SynchronizedBlockList();

        // Crear hilos para la sincronización con métodos
        for (int i = 0; i < 5; i++) {
            new ListOperationThread(methodList, true).start();
        }

        // Crear hilos para la sincronización con bloques
        for (int i = 0; i < 5; i++) {
            new ListOperationThread(blockList, false).start();
        }
    }
}

```

```

// Versión 1
class SynchronizedMethodList {
    private List<Integer> list = new ArrayList<>();

    /*
        AÑADIR CÓDIGO:
        Agregar los métodos sincronizados
    */

}

```

```

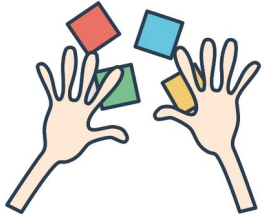
class SynchronizedBlockList {
    private List<Integer> list = new ArrayList<>();
    private final Object lock = new Object();

    /*
        AÑADIR CÓDIGO:
        Agregar los métodos con bloques sincronizados
    */

}

```

Manos-a-la-obra: lab2prob06



Objetivo:

- Modificar el código dado para explorar dos enfoques de sincronización:
 - Sincronización a nivel de método (sincronizando métodos completos).
 - Sincronización a nivel de bloque (sincronizando solo secciones críticas).

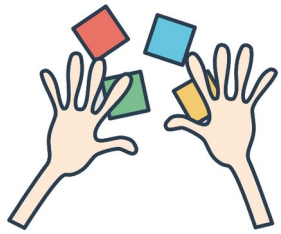
1. **Implementa métodos** sincronizados en SynchronizedMethodList
 - Agrega la palabra clave synchronized en la declaración del método para garantizar que solo un hilo pueda ejecutarlo a la vez.
2. **Implementa bloques sincronizados** en SynchronizedBlockList
 - Usa un objeto de bloqueo para proteger solo la sección crítica dentro de los métodos, en lugar de bloquear todo el método.
3. **Ejecuta el programa** y compara los resultados
 - Primero, ejecuta el programa con sincronización y verifica que todos los hilos accedan a la lista de manera segura.
 - Luego, elimina la sincronización y observa condiciones de carrera o inconsistencias.
 - Compara rendimiento y precisión entre los dos enfoques de sincronización.
4. **Generaliza** la solución para manejar más hilos
 - Modifica el código para crear y manejar dinámicamente un mayor número de hilos.
 - Asegúrate de que la sincronización siga funcionando de manera eficiente con una alta concurrencia.
5. **Documenta** tus hallazgos
 - Explica las diferencias entre métodos sincronizados y bloques sincronizados.
 - Discute las ventajas y desventajas de cada enfoque en términos de rendimiento y flexibilidad.

Lab2prob07:

Variables atómicas y CAS

Variables atómicas y CAS

```
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;
public class CompareAndSwapLock {
    private AtomicBoolean locked = new AtomicBoolean(false);
    public void lock() {
        while (!this.locked.compareAndSet(false, true)) {
            // espera activa - hasta que compareAndSet() tenga éxito
        }
    }
    public void unlock() {
        this.locked.set(false);
    }
    public static void main(String[] args) {
        final CompareAndSwapLock lock = new CompareAndSwapLock();
        final AtomicInteger counter = new AtomicInteger(0);
        final int numberOfThreads = 10;
        Thread[] threads = new Thread[numberOfThreads];
        for (int i = 0; i < threads.length; i++) {
            threads[i] = new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int j = 0; j < 1000; j++) {
                        lock.lock();
                        try {
                            // Sección crítica
                            // - solo un hilo puede incrementar el contador a la vez
                            counter.incrementAndGet();
                        } finally {
                            lock.unlock();
                        }
                    }
                }
            });
            threads[i].start();
        }
        for (Thread t : threads) {
            try {
                t.join(); // Esperar a que todos los hilos terminen
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Valor final del contador: " + counter.get());
    }
}
```



Manos-a-la-obra: Lab2prob07

Objective:

- Analyze the behavior and efficiency of a Compare-And-Swap (CAS) lock compared to a blocking synchronized approach in a multithreaded environment.

1. Execute the given CAS-based code
 - Observe its behavior and explain in your own words how the lock mechanism works.
 - Describe how the `compareAndSet()` method prevents multiple threads from modifying the counter simultaneously.
2. Implement a blocking version using synchronization
 - Modify the code to use a synchronized block instead of the CAS lock (`CompareAndSwapLock`).
 - Ensure that only one thread at a time can access the critical section.
3. Compare performance between CAS and synchronized versions
 - Measure the execution time of both implementations.
 - Run the program multiple times and document the results.
 - Consider why the performance difference might be small in this case.
4. Analyze CPU usage for both approaches
 - Monitor CPU usage while running the CAS and synchronized versions.
 - Discuss how CPU-intensive spinning in CAS might consume more resources under high contention.
5. Vary the number of threads to test contention levels
 - Increase the number of threads from low to high contention (e.g., 2, 10, 50, 100).
 - Compare which method performs better in high contention vs. low contention scenarios.
 - Explain why CAS might be more efficient with low contention but synchronized blocks may perform better when contention is high (due to reduced CPU spinning).
6. Document your findings
 - Explain how CAS-based locking differs from blocking synchronization.
 - Discuss the trade-offs between the two approaches regarding performance, CPU usage, and contention levels.
 - Draw conclusions on when each method is preferable in real-world applications.

Lab2prob08:

Hacer café con dos hilos

```

class SharedResource {
    final Object lock = new Object();
    boolean flag = false;
}

```

synchronized access to shared resources; The EnterAndWait pattern

```

class FirstThread extends Thread {
    static int iInteger;
    private final SharedResource sharedResource;

    public FirstThread(SharedResource sharedResource) {
        this.sharedResource = sharedResource;
    }

    @Override
    public void run() {
        System.out.println("Dentro de FirstThread");
        while (true) {
            firstCoffee();
            try {
                Thread.sleep(2000); // Simular trabajo
            } catch (InterruptedException e) {
                // Manejar interrupción
            }
        }

        void firstCoffee() {
            synchronized (sharedResource.lock) {
                while (sharedResource.flag) {
                    try {
                        sharedResource.lock.wait();
                    } catch (InterruptedException e) {
                        // Manejar interrupción
                    }
                }

                System.out.println(
                    Thread.currentThread().getName()
                    + " preparando café, iInteger=" + iInteger);

                sharedResource.flag = true;
                sharedResource.lock.notifyAll();
            }
        }
    }
}

```

```

class SecondThread extends Thread {
    private final SharedResource sharedResource;

    public SecondThread(SharedResource sharedResource) {
        this.sharedResource = sharedResource;
    }

    @Override
    public void run() {
        System.out.println("Dentro de SecondThread");
        while (true) {
            secondCoffee();
        }
    }

    void secondCoffee() {
        synchronized (sharedResource.lock) {
            while (!sharedResource.flag) {
                try {
                    sharedResource.lock.wait();
                } catch (InterruptedException e) {
                    // Manejar interrupción
                }
            }

            System.out.println(
                Thread.currentThread().getName() + " "
                + (++FirstThread.iInteger));
            sharedResource.flag = false;
            sharedResource.lock.notifyAll();
        }
    }
}

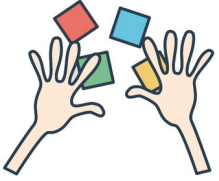
```

```

public class prob1 {
    public static void main(String[] args) {
        SharedResource sharedResource = new SharedResource();
        FirstThread a = new FirstThread(sharedResource);
        SecondThread b = new SecondThread(sharedResource);
        a.start();
        b.start();
    }
}

```

Manos-a-la-obra:Lab2prob08:



Objetivo:

- Analizar el comportamiento de un programa multihilo sincronizado en el que dos hilos coordinan la preparación de café usando un recurso compartido.

1. Ejecuta el código proporcionado

- Observa y analiza la salida para comprender cómo interactúan los hilos.
- Identifica cómo funciona el mecanismo de espera y notificación entre los hilos.
- Explica el comportamiento de la sincronización con tus propias palabras.

2. Refactoriza el código en una sola clase de hilos

- Fusiona las clases FirstThread y SecondThread en una única clase que pueda definir dinámicamente su comportamiento.
- Usa un parámetro en el constructor para indicar si el hilo ejecutará la primera o la segunda etapa del proceso de hacer café.
- Asegúrate de que el mecanismo de sincronización siga funcionando correctamente.

3. Prueba y verifica tu código refactorizado

- Ejecuta la nueva versión y confirma que el comportamiento de la salida sigue siendo correcto.
- Compara la estructura y legibilidad del código original y el refactorizado.
- Reflexiona sobre los beneficios de usar una sola clase para definir dinámicamente el comportamiento de los hilos en lugar de clases separadas.

Lab2prob09 (10)

Entrar y Esperar

EnterAndWait() sin sincronización

```
import java.util.ArrayList;
import java.util.List;

class TaskExecutor {
    public void enterAndWait(int threadNumber){
        try {
            System.out.println("Iniciando Hilo " + threadNumber);
            Thread.sleep((int) (Math.random() * 100));
            System.out.println("Finalizando Hilo " + threadNumber);
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

```
class TaskRunnable implements Runnable {
    private final TaskExecutor taskExecutor;
    private final int threadNumber;

    TaskRunnable(int threadNumber, TaskExecutor taskExecutor) {
        this.threadNumber = threadNumber;
        this.taskExecutor = taskExecutor;
    }

    @Override
    public void run() {
        taskExecutor.enterAndWait(threadNumber);
    }
}
```

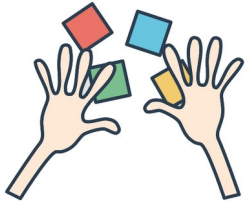
```
public class lab2prob10 {
    public static void main(String[] args) {
        TaskExecutor taskExecutor = new TaskExecutor();
        int numberOfThreads = 5;
        List<Thread> threads = new ArrayList<>();

        for (int i = 0; i < numberOfThreads; i++) {
            TaskRunnable taskRunnable
                = new TaskRunnable(i, taskExecutor);
            Thread thread = new Thread(taskRunnable);
            threads.add(thread);
            thread.start();
        }

        try {
            for (Thread t : threads) {
                t.join(); // Esperar a que todos
                        // los hilos terminen
            }
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }

        System.out.println("Finalizado.");
    }
}
```

Manos-a-la-obra:Lab2prob10



Objetivo:

- Modificar el código dado para sincronizar los hilos utilizando `wait()` y `notify()`, asegurando que solo un hilo pueda ejecutar la sección crítica a la vez.

1. Ejecuta el código proporcionado

- Observa cómo múltiples hilos ejecutan el método `enterAndWait()` simultáneamente.
- Explica el comportamiento con tus propias palabras, describiendo cómo interactúan los hilos.

2. Modifica `TaskExecutor` para sincronizar los hilos

- Usa bloques sincronizados junto con `wait()` y `notify()` para garantizar que solo un hilo pueda entrar en la sección crítica a la vez.
- No es necesario imponer un orden específico de ejecución entre los hilos.

3. Prueba y compara los resultados

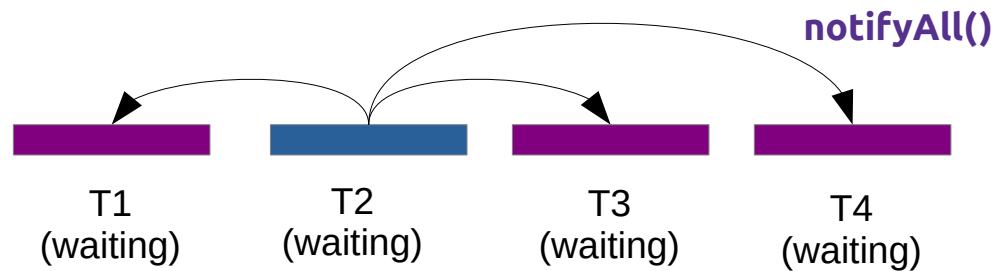
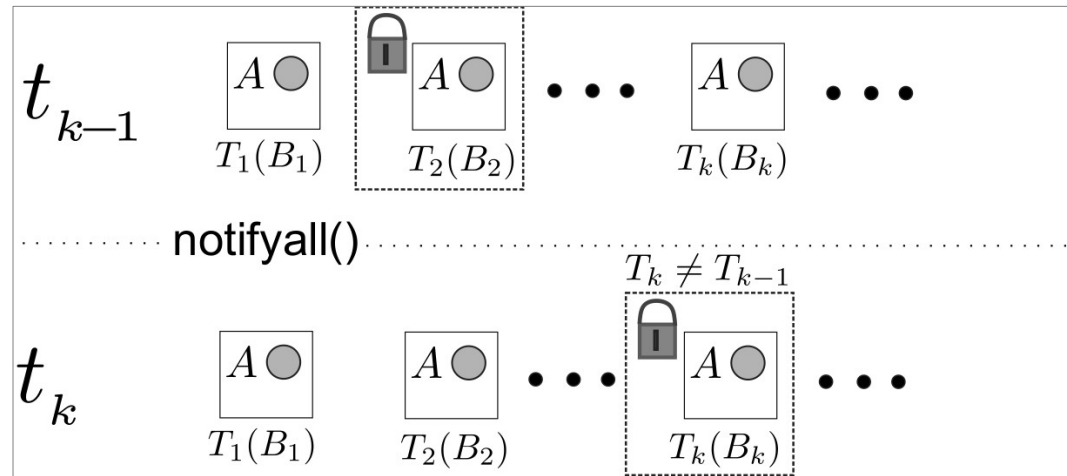
- Ejecuta el código modificado y observa la diferencia en la ejecución.
- Compara el comportamiento antes y después de la sincronización, explicando las diferencias clave.

Lab2prob11:

Entrar y Esperar: Con condición de planificación

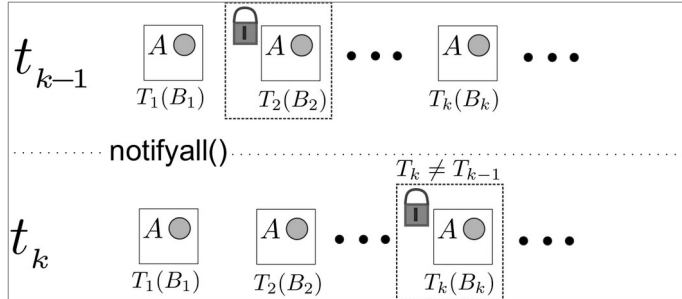
Problema Entrar/Esperar

Problema 2: Sincronización de Hilos con Acceso Condicional



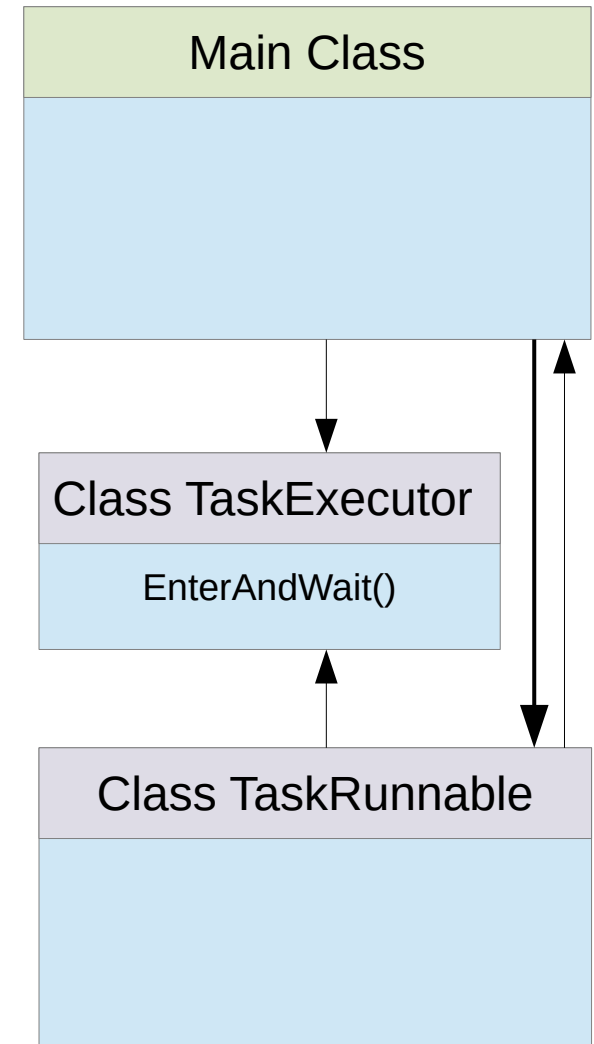
Problema Entrar/Esperar

Problema 2: Sincronización de Hilos con Acceso Condicional



Modificaciones de ClassA:

- Agregar un atributo entero llamado counter, inicializado en el constructor, y decrementarlo en uno cada vez que se invoque `enterAndWait()`.
- Implementar un método `isFinished()` para comprobar si counter es 0 (devolviendo true) o no (devolviendo false).
- Agregar un atributo `Set<Long>` llamado `threadIds` para almacenar los IDs de los hilos que han ejecutado `enterAndWait()`, junto con un método para recuperar este conjunto.



```

import java.util.ArrayList;

class TaskExecutor {
    public void enterAndWait(int threadId) {
        try {
            System.out.println("Iniciando Hilo " + threadId);
            Thread.sleep((int) (Math.random() * 100));
            System.out.println("Finalizando Hilo " + threadId);
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
}

```

```

class Task implements Runnable {
    private final TaskExecutor taskExecutor;
    private final int threadId;

    Task(int threadId, TaskExecutor taskExecutor) {
        this.threadId = threadId;
        this.taskExecutor = taskExecutor;
    }

    @Override
    public void run() {
        taskExecutor.enterAndWait(threadId);
    }
}

```

```

public class SchedulerDemo {
    public static void main(String[] args) {
        TaskExecutor taskExecutor = new TaskExecutor();
        int numberOfTasks = 5;
        ArrayList<Thread> threads = new ArrayList<>();

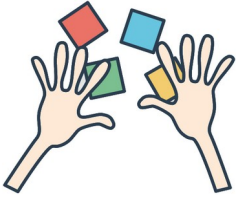
        for (int i = 0; i < numberOfTasks; i++) {
            Thread thread = new Thread(new Task(i, taskExecutor));
            threads.add(thread);
            thread.start();
        }

        for (Thread thread : threads) {
            try {
                thread.join(); // Esperar a que todos los hilos terminen
            } catch (InterruptedException e) {
                System.out.println(e.getMessage());
            }
        }

        System.out.println("Finalizado.");
    }
}

```

Manos-a-la-obra: Lab2prob11



Objetivo:

- Modificar el código lab2prob11 para controlar el orden de ejecución de los hilos en TaskExecutor, implementando un algoritmo de planificación.

1. Define una estrategia de planificación

- Elige una lógica para la planificación, como:
 - Orden ascendente de IDs de los hilos (los hilos con ID más bajo ejecutan primero).
 - Selección aleatoria (ejecución en orden aleatorio).
 - Ejecución basada en prioridad (asignar prioridades personalizadas).
 - Orden basado en tiempo de ejecución (simular tiempos de ejecución diferentes).

2. Implementa la planificación en SchedulerDemo

- Modifica el código para aplicar el algoritmo de planificación antes de iniciar los hilos.
- Considera ordenar los hilos o usar una PriorityQueue para gestionar el orden de ejecución de manera dinámica.

3. Modifica TaskExecutor para reforzar el orden de ejecución

- Asegúrate de que los hilos se ejecuten en la secuencia planeada.
- Puedes utilizar mecanismos de sincronización para gestionar la coordinación entre los hilos.

4. Prueba y verifica el algoritmo de planificación

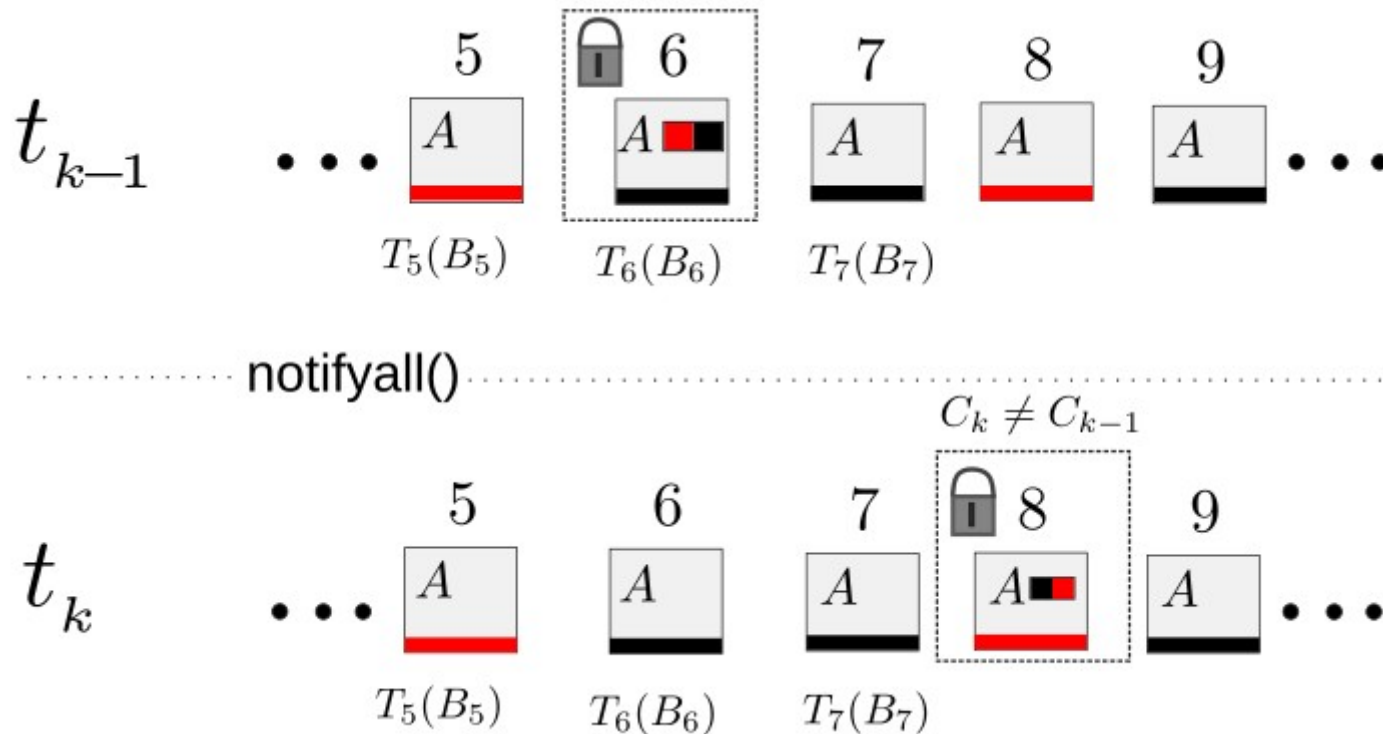
- Ejecuta múltiples pruebas con diferentes estrategias de planificación.
- Compara el orden de ejecución y documenta cómo cambia con cada enfoque.

Lab2prob11B:

Entrar y Esperar: Con condición de planificación

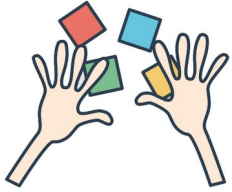
Problema de rojo/negro Entrar/Esperar

Problema 3: Sincronización de Hilos con Acceso Condicional



- **Anteriormente:** Los hilos competían continuamente y el SO decidía el siguiente.
- **Último problema:** La entrada se basaba en un número.
- **Ahora:** Añadir condiciones “Rojo” y “Negro”.

Manos-a-la-obra: Lab2prob11



Objetivo:

- Modificar lab2prob11 para incorporar un algoritmo de planificación que controle el orden en que los hilos ejecutan su método `enterAndWait` en `TaskExecutor`, añadiendo además la restricción explícita de hilos Rojo y Negro.

1. Define una estrategia de planificación con hilos Rojo y Negro

- Cada hilo debe ser asignado como Rojo o Negro.
- La planificación debe alternar o priorizar la ejecución basada en el color del hilo.
- Se pueden implementar diferentes estrategias, como:
 - Alternancia estricta: Rojo → Negro → Rojo → Negro
 - Priorización: Ejecutar primero todos los hilos Rojos, luego los Negros
 - Selección aleatoria: Elegir aleatoriamente entre los hilos disponibles

2. Implementa la planificación usando `wait()` y `notify()`

- Usa el mecanismo de espera y notificación de Java para controlar el orden de ejecución.
- Asegúrate de que solo los hilos del color permitido puedan ejecutarse en cada momento.
- Implementa la sincronización en `TaskExecutor` para gestionar las restricciones de ejecución.

3. Modifica `SchedulerDemo` para aplicar el orden de ejecución

- Usa una `PriorityQueue`, mecanismo de ordenación o lógica personalizada para planificar los hilos.
- Asegúrate de que los hilos Rojo y Negro se ejecuten según la lógica definida.

4. Prueba y verifica la implementación

- Ejecuta diferentes estrategias de planificación y compara el orden de ejecución.
- Observa y documenta cómo el mecanismo de espera y notificación afecta la ejecución.
- Compara los resultados con y sin planificación para resaltar las mejoras.

Lab2prob12

Problema Productor-Consumidor

- Seguro para hilos, pero puede llevar a espera activa.
- Enfoque de bloque sincronizado

```
import java.util.LinkedList;

class SharedBuffer {
    private LinkedList<Integer> buffer = new LinkedList<>();
    private int capacity = 10;

    /*
     * AÑADIR CÓDIGO: Escribir el método "add"
     * que recibe 'value'.
     * - Usar un bloque sincronizado.
     * - Debe esperar si el búfer está lleno
     *   (ha alcanzado su capacidad).
     * - En caso contrario, debe agregar el valor
     *   al búfer y notificar al resto de los hilos.
     *
     */

    /*
     * AÑADIR CÓDIGO: Escribir el método "remove".
     * - Usar un bloque sincronizado.
     * - Debe esperar si el búfer está vacío.
     * - En caso contrario, debe eliminar el primer
     *   elemento del búfer.
     * - Después de eso, debe notificar a otros
     *   hilos que ha finalizado.
     */
}
```

```
class Producer extends Thread {
    private SharedBuffer buffer;

    public Producer(SharedBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            buffer.add(i);
            System.out.println("Producido: " + i);
        }
    }
}

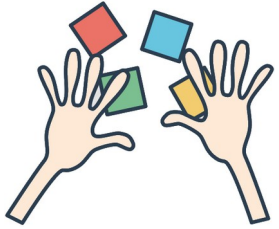
class Consumer extends Thread {
    private SharedBuffer buffer;

    public Consumer(SharedBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            int value = buffer.remove();
            System.out.println("Consumido: " + value);
        }
    }
}
```

```
public class lab2prob12 {
    public static void main(String[] args) {
        SharedBuffer buffer = new SharedBuffer();
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);
        producer.start();
        consumer.start();
    }
}
```

Manos-a-la-obra: Lab2prob12



Objective:

- Complete the implementation of the Producer-Consumer problem using Java's wait-notify mechanism to handle synchronization between threads.

1. Implement the missing methods in SharedBuffer

- `add(int value)`:
 - Use a synchronized block to ensure thread-safe access.
 - If the buffer is full, the producer should wait until space is available.
 - Otherwise, add value to the buffer and notify the consumer.
- `remove()`:
 - Use a synchronized block to ensure thread-safe access.
 - If the buffer is empty, the consumer should wait until new items are available.
 - Otherwise, remove the first element from the buffer and notify the producer.

2. Explain your implementation

- Describe how your solution prevents race conditions.
- Explain how `wait()` and `notify()` help coordinate thread execution.

3. Demonstrate correct execution of Producer-Consumer

- Run your completed code and verify that produced values are correctly consumed.
- Observe the output to ensure that the buffer does not overflow (overproduction) or underflow (overconsumption).

4. Experiment with different buffer sizes

- Modify the buffer capacity and observe how synchronization behavior changes.
- Try different production and consumption speeds to analyze performance.