CCPD

Tema U2: Concurrencia

Lab 3: Estructuras de Datos Sincronizadas

2024/2025

David Olivieri
Leandro Rodriguez Liñares
Uvigo, E.S. Informatica

lab3prob01

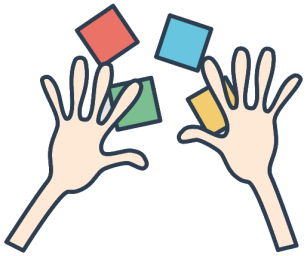
Ejemplo: Tick Tock

Ejemplo Simple: TickTock

```
class TickTock {
    String state;
    synchronized void tick (boolean running){
      if (!running) {   //stop clock
          state="ticked";
          notify(); // notify any waiting threads
          return;
      }
      System.out.print("Tick... ");
      state="ticked";
      notify();
      try {
          while(!state.equals("tocked"))
            wait(); // wait for tock to complete
      } catch (InterruptedException exc) {
          System.out.println("interrupted");
      }
    }

    synchronized void tock (boolean running){
      if (!running) { //stop clock
          state="tocked";
          notify(); // notify any waiting threads
          return;
      }
      System.out.println("Tock ");
      state="tocked";
      notify();
      try {
          while(!state.equals("ticked"))
            wait(); // wait for tock to complete
      } catch (InterruptedException exc) {
          System.out.println("interrupted");
      }
    }
}
```

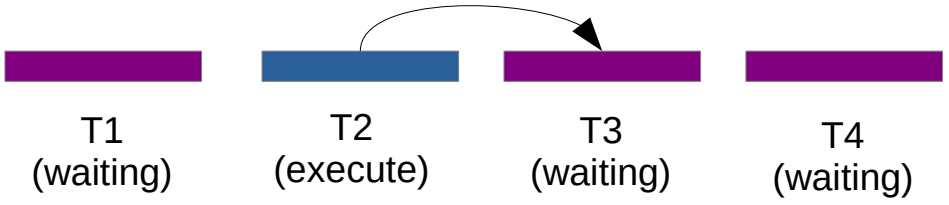
```
class MyThread implements Runnable{
    Thread thrd;
    TickTock ttOb;

    // constructor of new thread
    MyThread(String name, TickTock tt){
      thrd = new Thread(this, name);
      ttOb = tt;
    }

    public void run() {
      if(thrd.getName().compareTo("Tick") == 0){
          for (int i=0; i<=5; i++) ttOb.tick(true);
          ttOb.tick(false);
      }else {
          for (int i=0; i<=5; i++) ttOb.tock(true);
          ttOb.tock(false);
      }
    }
}
public class probTickTock{
    public static void main(String agrs[]){
      TickTock tt = new TickTock();

      MyThread mt1 = new MyThread("Tick", tt);
      MyThread mt2 = new MyThread("Tock", tt);

      mt1.thrd.start();
      mt2.thrd.start();

      try {
          mt1.thrd.join();
          mt2.thrd.join();
      } catch (InterruptedException exc){
          System.out.println("main thread interrupted");
      }
    }
}
```

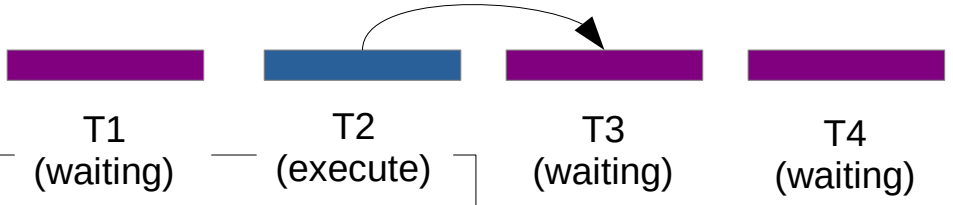
Manos-a-la-obra: Lab3Prog01



1. Ejecuta el código proporcionado y observa su comportamiento.
2. Explica el comportamiento con tus propias palabras.
3. Describe lo que ocurre cuando el programa se ejecuta varias veces.
   - ¿El resultado es siempre el mismo? ¿Por qué sí o por qué no?
4. Comprende la lógica de las estructuras notify()/wait().
5. Añade un hilo extra llamado "Toe" para que la salida sea:
   - "Tick-Tock-Toe"

lab3prob02
notify/wait “next.Thread”





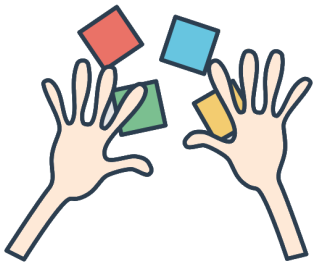

```
public class lab3prob02 {
    public static void main(String[] args) {
        int maxCount = 10;
        int startingTurn = 2;
        int numberOfThreads = 10;

        A sharedObject = new A(maxCount, startingTurn, numberOfThreads);
        B[] threadArray = new B[numberOfThreads];

        long startTime = System.nanoTime();

        for (int i = 0; i < numberOfThreads; i++) {
            int nextThreadIndex = (i + 1) % numberOfThreads;
            threadArray[i] = new B(i, sharedObject, numberOfThreads,
                  nextThreadIndex, threadArray);
        }

        for (B thread : threadArray) {
            thread.start();
        }

        synchronized (threadArray[startingTurn]) {
            threadArray[startingTurn].notify();
        }

        for (B thread : threadArray) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                System.out.println("A thread was interrupted: "
                     + e.getMessage());
                Thread.currentThread().interrupt();
            }
        }

        long estimatedTime = System.nanoTime() - startTime;
        System.out.println((float) estimatedTime / 1_000_000 + " ms");
    }
}
```

```
class B extends Thread {
    A sharedA;
    B[] threads;
    int threadId;
    private int nextThreadId;
    private int numberOfThreads;

    boolean done = false;

    B(int threadId, A sharedA, int numberOfThreads,
                int nextThreadId, B[] threads) {
        this.threadId = threadId;
        this.sharedA = sharedA;
        this.numberOfThreads = numberOfThreads;
        this.nextThreadId = nextThreadId;
        this.threads = threads;
    }


    // run method
}
```

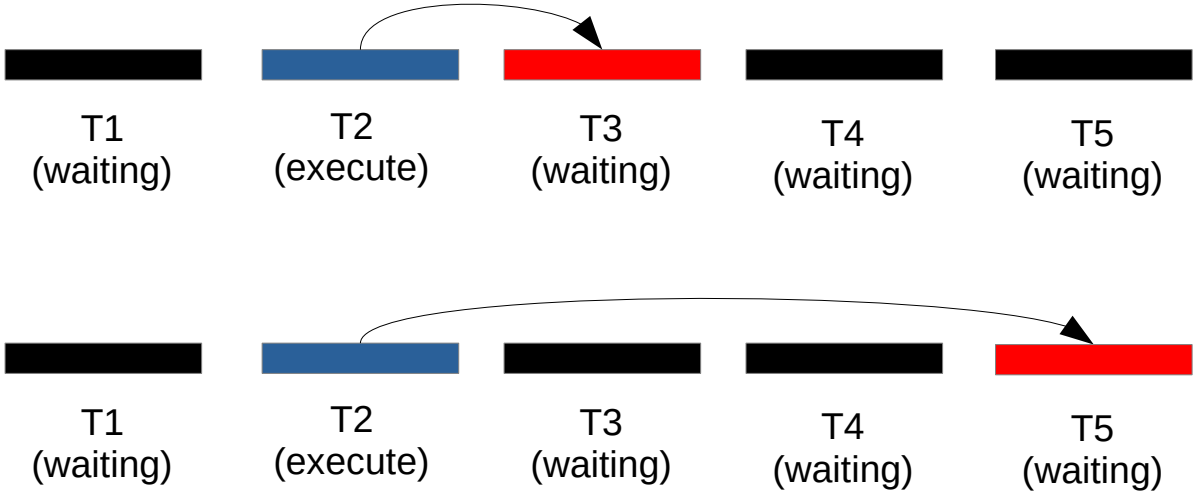
```
    public void run() {
        try {
            synchronized (this) {
                wait();
            }
            boolean flag = false;
            while (!flag) {
                if (sharedA.getDone()) {
                    synchronized (threads[nextThreadId]) {
                        threads[nextThreadId].notify();
                    }
                    flag = true;
                } else {
                    synchronized (threads[nextThreadId]) {
                        sharedA.enterAndWait(threadId);
                        if (sharedA.getCurrentCount() < 0) {
                            done = true;
                            sharedA.setDone(done);
                        }
                        threads[nextThreadId].notify();
                    }
                    synchronized (this) {
                        wait();
                    }
                }
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
```

```
class A {
    private volatile int currentCount;
    private int turn, numberOfThreads;
    private boolean done = false;

    public A(int currentCount, int turn, int numberOfThreads) {
        this.currentCount = currentCount;
        this.turn = turn;
        this.numberOfThreads = numberOfThreads;
    }

    public synchronized void enterAndWait(int threadId) {
        try {
            System.out.println("Start Thread " + threadId + " currentCount=" + currentCount);
            Thread.sleep((int) (Math.random() * 100));
            System.out.println("Finish Thread " + threadId + " currentCount" + currentCount);
            currentCount--;
            if (currentCount <= 0) {
                done = true;
            }
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }

    public boolean isFinished() {return currentCount <= 0; }
    public int getTurn() {return turn; }
    public int getCurrentCount() {return currentCount; }

    public void setTurn(int turn) {this.turn = turn;  }
    public void setCurrentCount(int currentCount) {
        this.currentCount = currentCount;
    }

    public void setDone(boolean done) { this.done = done;    }
    public boolean getDone() {return this.done; }
}
```

Manos-a-la-obra: Lab3Prog02

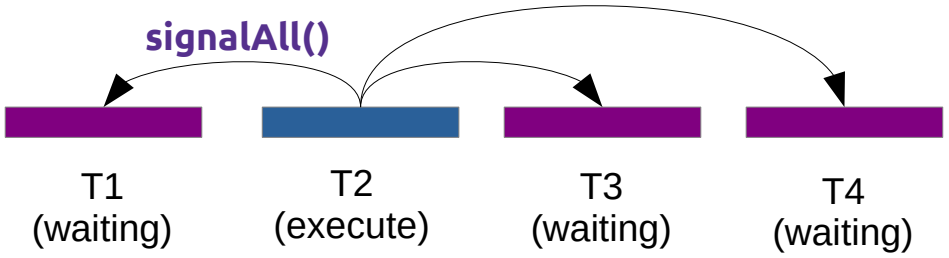


1. **Ejecuta el código** anterior y comprende cómo funciona.
2. **Modifica el código** para que el hilo que obtenga el control deba cumplir una condición:
   - Debe ser "rojo" y no "negro".
   - Si no es "rojo", imprime un mensaje de advertencia.
3. **Extra** (opcional):
   - Si el hilo no es "rojo", salta al siguiente hilo que sea "rojo"

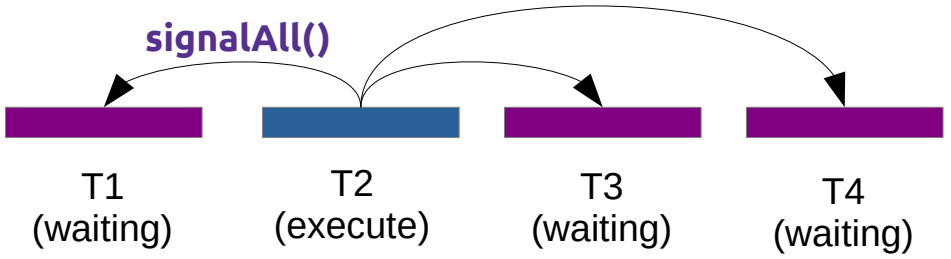


lab3prob03

lock.signalAll/await “scheduled Thread”



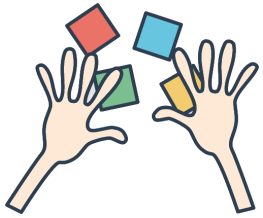
Planificación con Locks




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

public class WorkerThread implements Runnable {
    private static final ReentrantLock lock = new ReentrantLock();
    private static final Condition turnCondition = lock.newCondition();
    private static int currentMaxId;
    private final int threadId;

    public WorkerThread(int id, int maxId) {
        this.threadId = id;
        currentMaxId = maxId;
    }

    @Override
    public void run() {
        lock.lock();
        try {
            while (threadId != currentMaxId) {
                turnCondition.await();
            }
            // Critical section
            System.out.println("Thread with ID " + threadId
                    + " entering critical section.");
            Thread.sleep(1000); // Simulate work
            System.out.println("Thread with ID " + threadId +
                    " leaving critical section.");
            currentMaxId--; // Move to the next thread
            turnCondition.signalAll(); // Notify all waiting threads
                                       // to check their condition
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            lock.unlock();
        }
    }
}
```

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

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

        // Create and start threads in reverse order
        for (int i = numberOfThreads; i > 0; i--) {
            Thread thread = new
                 Thread(new WorkerThread(i, numberOfThreads));
            threads.add(thread);
            thread.start();
        }

        // Wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        System.out.println("All threads have executed.");
    }
}
```

Manos-a-la-obra: Lab3Prog03



1. Ejecuta el código anterior.
2. Explora la equidad en los Locks:
   - Comprende el impacto de la política de equidad en ReentrantLock sobre el orden de ejecución de los hilos y el rendimiento general del sistema

Idea:

- Un ReentrantLock con equidad (fairness) activada garantiza el acceso en orden FIFO (First-In-First-Out), evitando que un hilo quede sin acceso a la sección crítica (starvation).
- Sin embargo, esto puede reducir el rendimiento general, ya que se introduce una sobrecarga para gestionar la cola de hilos en espera.
- Pista: Piensa en relajar while (threadId != currentMaxId).

Resultado Esperado:

- Con equidad activada, los hilos deben ejecutarse en orden secuencial estricto, del ID más alto al más bajo, demostrando el comportamiento FIFO del lock justo.
- Activar la equidad probablemente aumentará el tiempo total de ejecución debido a la gestión adicional de la cola de espera de hilos.

1 Modificar el Lock para Ser Justo:

Actualiza la instancia estática de ReentrantLock en la clase WorkerThread para que se inicialice con el parámetro de equidad (fairness) configurado en true:

```
private static final ReentrantLock lock = new ReentrantLock(true);
```

2 Analizar el Orden de Ejecución:

- Ejecuta el programa modificado y observa el orden en que los hilos entran y salen de la sección crítica.
- deben verificar si el orden es estrictamente secuencial, desde el ID más alto hasta el más bajo, lo que indicaría que la política de equidad está siendo aplicada correctamente.

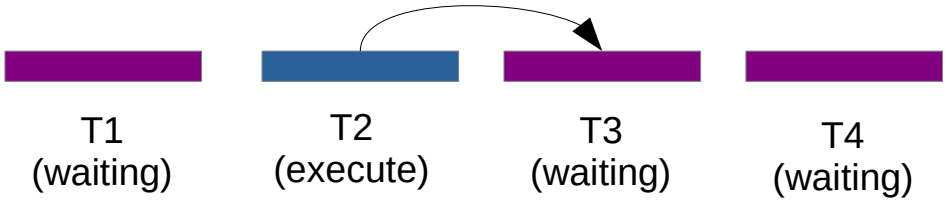
3 Medir el Impacto en el Rendimiento:

Mide el tiempo total de ejecución del programa con la política de equidad tanto activada como desactivada (la configuración original sin el parámetro de equidad).

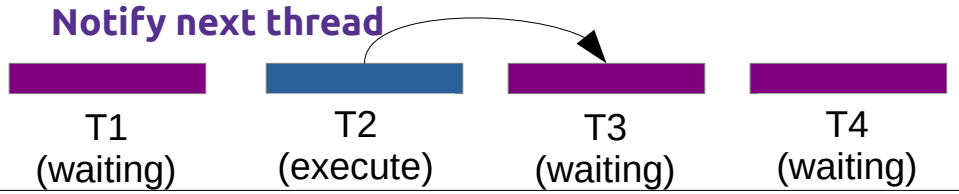
```
System.currentTimeMillis()
```

lab3prob04

lock.signal/await “next.Thread” problem



Planificación con Locks



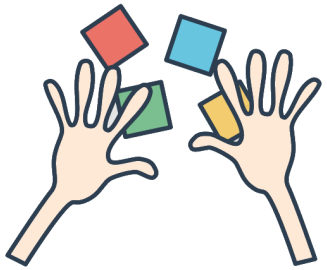

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

public class WorkerThread implements Runnable {
    private final int threadId;
    private static ReentrantLock lock = new ReentrantLock();
    private static Condition[] conditions; // Array of condition, one for each thread
    private static int currentMaxId;

    public WorkerThread(int id, int maxId, Condition condition) {
        this.threadId = id;
        currentMaxId = maxId;
        if (conditions == null) { // Initialize conditions array once
            conditions = new Condition[maxId + 1];
        }
        conditions[id] = condition; // Assign condition to this thread
    }
    // Getter for conditions
    public static Condition getCondition(int index) {
        return conditions[index];
    }
    @Override
    public void run() {
        lock.lock();
        try {
            while (threadId != currentMaxId) {
                conditions[threadId].await();
            }
            // Critical section
            System.out.println("Thread with ID "  + threadId + " entering critical section.");
            Thread.sleep(1000); // Simulate work
            System.out.println("Thread with ID "  + threadId + " leaving critical section.");
            currentMaxId--; // Move to the next thread
            if (currentMaxId > 0) {
                conditions[currentMaxId].signal(); // Notify the next thread
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            lock.unlock();
        }
    }
}
```

NOTE:

- With lock mechanism, we use the condition for scheduling;
- Contrast this with the case of wait/notify on the object

```java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.locks.ReentrantLock;
import java.util.concurrent.locks.Condition;

public class lab3prob04 {
    public static void main(String[] args) {
        final int numberOfThreads = 5;
        ReentrantLock lock = new ReentrantLock();
        List<Thread> threads = new ArrayList<>();

        // Initialize threads
        for (int i = 1; i <= numberOfThreads; i++) {
            Thread thread = new Thread(new WorkerThread(i, numberOfThreads, lock));
            threads.add(thread);
        }

        // Start threads in reverse order
        for (int i = numberOfThreads - 1; i >= 0; i--) {
            threads.get(i).start();
        }

        // Signal the first thread to start
        lock.lock();
        try {
            if (numberOfThreads > 0) {
                WorkerThread.getCondition(numberOfThreads).signal();
            }
        } finally {
            lock.unlock();
        }

        // Wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        System.out.println("All threads have executed.");
    }
}
```

Manos-a-la-obra: Lab3Prog04



1. Comprende los detalles y cómo funciona.
2. Explora diferentes algoritmos de señalización que podrían aplicarse con el array de condiciones (condition array). Elige uno de los siguientes métodos a continuación.

1

Ejecución en Round-Robin:

- Modifica la lógica de señalización para que los hilos se despierten en un orden circular (round-robin) en lugar de en orden inverso.
- Después de completar su sección crítica, cada hilo debe señalar al siguiente hilo en una secuencia circular, volviendo al primer hilo después de que el último haya terminado su trabajo.
- Esto requiere modificar cómo se decrementa currentMaxId y garantizar que el hilo correcto sea señalado a continuación.

2

Señalización de Hilo Aleatorio:

En lugar de un orden fijo, modifica el programa para señalar a un hilo aleatorio para que ingrese a la sección crítica a continuación.

Esto requerirá generar un índice aleatorio para señalar un hilo desde el array de condiciones (condition array).

3

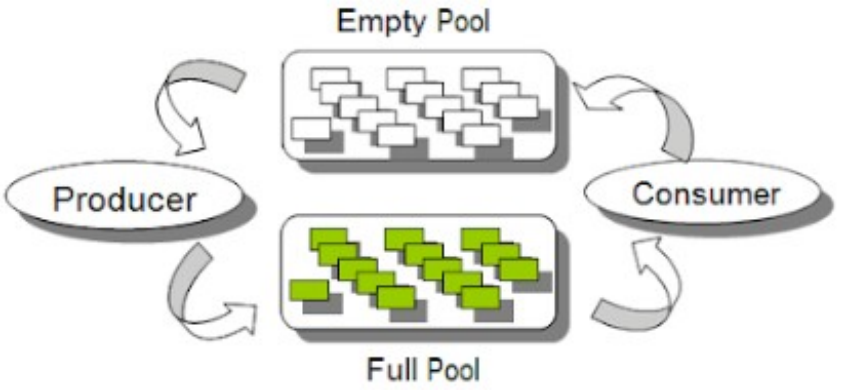
Señalización por Grupos:

Divide los hilos en dos o más grupos y modifica el programa para que todos los hilos de un grupo entren simultáneamente en sus secciones críticas (en la medida de lo posible con un solo lock).

Después de que un grupo termine, el siguiente grupo es señalado para ejecutar.

Lab3prob05

notify/wait con el “Bounded Buffer”

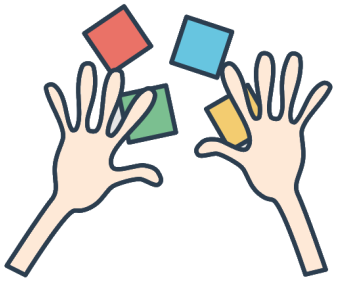


```
import java.util.LinkedList;
import java.util.Queue;

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

    public synchronized void produce(int item) throws
InterruptedException {
        while (buffer.size() == capacity) {
            wait(); // Wait when buffer is full
        }
        buffer.add(item);
        System.out.println("Produced: " + item);
        notifyAll(); // Notify all waiting consumers and
producers
    }

    public synchronized int consume() throws
InterruptedException {
        while (buffer.isEmpty()) {
            wait(); // Wait when buffer is empty
        }
        int item = buffer.remove();
        System.out.println("Consumed: " + item);
        notifyAll(); // Notify all waiting consumers and
producers
        return item;
    }
}
```

```
public class MultiProducerConsumerDemo {
    public static void main(String[] args) {
        SharedBuffer buffer = new SharedBuffer();
        int numberOfProducers = 4; // Can be adjusted
        int numberOfConsumers = 4; // Can be adjusted

        for (int i = 0; i < numberOfProducers; i++) {
            new Producer(buffer).start();
        }
        for (int i = 0; i < numberOfConsumers; i++) {
            new Consumer(buffer).start();
        }
    }
}
```

```
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++) {
            try {
                buffer.produce(i);
            } catch (InterruptedException e) {
                e.printStackTrace();
                Thread.currentThread().interrupt();
                return; // Stop the thread if interrupted
            }
        }
    }
}

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++) {
            try {
                int value = buffer.consume();
            } catch (InterruptedException e) {
                e.printStackTrace();
                Thread.currentThread().interrupt();
                return; // Stop the thread if interrupted
            }
        }
    }
}
```

Manos-a-la-obra: Lab3Prog05



1. Comprende el Funcionamiento del Productor-Consumidor con Buffer Acotado
   - Analiza cómo funciona el productor-consumidor utilizando un buffer acotado.
   - Observa el método alternativo propuesto en el código, que aprovecha la función lambda

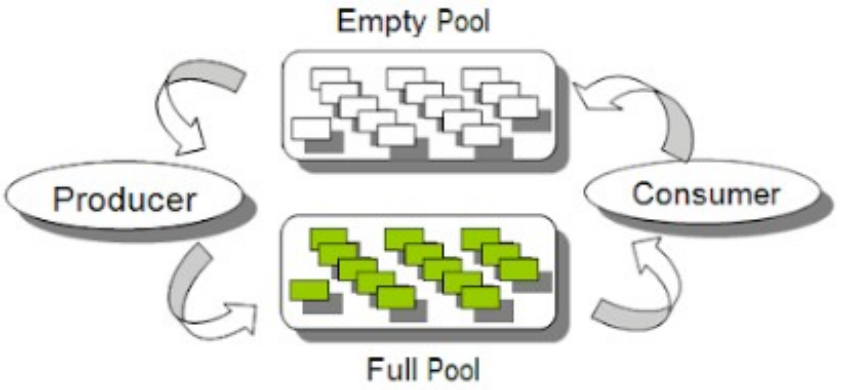
Uso de Expresiones Lambda en Java

- En Java, la estructura new Thread(() -> {}) utiliza una expresión lambda (() -> {}) para proporcionar una implementación concisa de la interfaz Runnable.

- Expresión Lambda (() -> {}):
  - Introducida en Java 8, permite tratar la funcionalidad como un argumento de método o el código como datos.
  - Es una forma compacta de representar una instancia de una interfaz funcional.
  - En el ejemplo dado, () -> {} es una expresión lambda que implementa la interfaz Runnable sin argumentos y con un cuerpo vacío.

Lab3prob06

Locks con “Bounded Buffer”



```
class BoundedBuffer {
    // Lock to control access to the buffer
    final Lock lock = new ReentrantLock();

    // Condition variables to manage the state of the buffer
    final Condition notFull = lock.newCondition();
    final Condition notEmpty = lock.newCondition();

    // The buffer and its metadata
    final Object[] items = new Object[100]; // Buffer size
    int putptr, takeptr, count; // Indices for putting and taking items, and the current number of items

    // Method to add an item to the buffer
    public void put(Object x) throws InterruptedException {
        lock.lock(); // Acquire the lock before modifying the buffer
        try {
            // Wait while the buffer is full
            while (count == items.length) {
                notFull.await();
            }
            // Add item to the buffer
            items[putptr] = x;
            if (++putptr == items.length) putptr = 0; // Circular increment of put pointer
            ++count; // Increase the buffer item count
            notEmpty.signal(); // Signal any waiting consumers that there is data
        } finally {
            lock.unlock(); // Ensure the lock is always released
        }
    }
    // Method to remove and return an item from the buffer
    public Object take() throws InterruptedException {
        lock.lock(); // Acquire the lock before modifying the buffer
        try {
            // Wait while the buffer is empty
            while (count == 0) {
                notEmpty.await();
            }
            // Remove item from the buffer
            Object x = items[takeptr];
            if (++takeptr == items.length) takeptr = 0; // Circular increment of take pointer
            --count; // Decrease the buffer item count
            notFull.signal(); // Signal any waiting producers that there is space
            return x;
        } finally {
            lock.unlock(); // Ensure the lock is always released
        }
    }
}
```

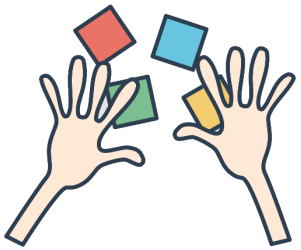
```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
```

```
public class lab3prob06 {
    public static void main(String[] args) {
        BoundedBuffer buffer = new BoundedBuffer();

        // Create and start multiple producer threads.
        int numberOfProducers = 3;
        for (int i = 0; i < numberOfProducers; i++) {
            Thread producerThread = new Thread(new Producer(buffer), "Producer-" + i);
            producerThread.start();
        }

        // Create and start multiple consumer threads.
        int numberOfConsumers = 3;
        for (int i = 0; i < numberOfConsumers; i++) {
            Thread consumerThread = new Thread(new Consumer(buffer), "Consumer-" + i);
            consumerThread.start();
        }
    }
}
```

```
class Producer implements Runnable {
    private final BoundedBuffer buffer;

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

    @Override
    public void run() {
        try {
            while (true) {
                Object item = produceItem();
                buffer.put(item);
                System.out.println(Thread.currentThread().getName()
                 + " produced " + item);
                Thread.sleep(1000); // Simulate time to produce an item
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println("Producer was interrupted.");
        }
    }

    // Simulate producing an item
    private Object produceItem() {
        return "item" + Math.random();
    }
}
```

```
class Consumer implements Runnable {
    private final BoundedBuffer buffer;

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

    @Override
    public void run() {
        try {
            while (true) {
                Object item = buffer.take();
                System.out.println(Thread.currentThread().getName()
                    + " consumed " + item);
                Thread.sleep(1000); // Simulate time to consume an item
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println("Consumer was interrupted.");
        }
    }
}
```

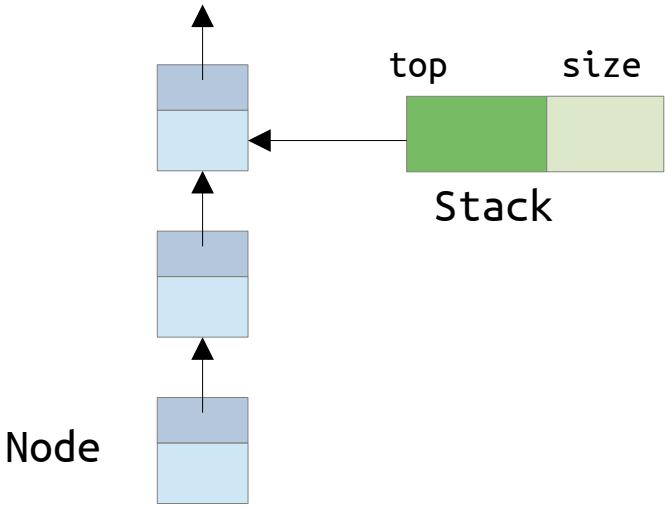
Manos-a-la-obra: Lab3Prog06



1. Ejecuta el código anterior.
2. Compara y contrasta esta implementación con una solución similar de buffer acotado que utilice los métodos tradicionales del monitor de objetos (wait() y notify()) para la comunicación y sincronización de hilos.
    - En tu comparación, considera los siguientes puntos:
        - Sintaxis y Legibilidad:
        - Analiza cómo el uso de Lock y Condition se compara con wait() y notify() en términos de legibilidad del código y claridad del mecanismo de sincronización.

Lab3prob07

Synchronized Stack



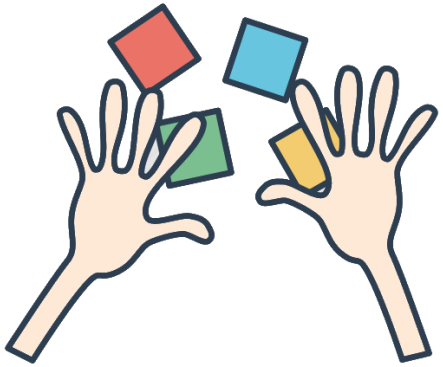
Synchronized Stack

```
class SynchronizedStack {
    private NodeInt top;
    private final ReentrantLock lock = new ReentrantLock();
    private final Condition notEmptyCondition = lock.newCondition();

    public SynchronizedStack() {
        this.top = null;
    }

    // Method to add an item to the top of the stack
    public void push(int value) {
        lock.lock();
        try {
            NodeInt newNode = new NodeInt(value);
            newNode.next = top;
            top = newNode;
            notEmptyCondition.signalAll(); // waiting pop or peek
        } finally {
            lock.unlock();
        }
    }

        // Method to remove and return the top item from the stack

        public int pop() throws InterruptedException {
            // ADD CODE
        }

        // Method to return the top item without removing it
        public int peek() throws InterruptedException {
            //  ADD CODE
        }

        // Method to check if the stack is empty
        public boolean isEmpty() {
            // ADD CODE
        }
}
```

```
class NodeInt {
    int value;
    NodeInt next;

    public NodeInt(int value) {
        this.value = value;
        this.next = null;
    }
}
```

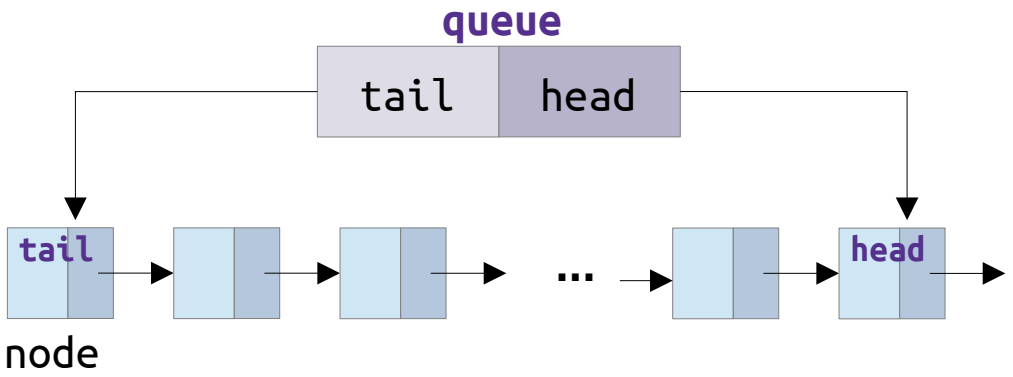
- The push method signals (notEmptyCondition.signalAll()) all waiting threads after adding a new item to the stack.
- This is to notify any threads waiting in pop or peek that they can proceed since the stack is no longer empty.

Manos-a-la-obra: Lab3Prog07



1. Ejecuta el código anterior.
2. Completa la implementación de los métodos del stack concurrente que actualmente no están implementados.
   - Consulta las notas y ejemplos de métodos de pila no concurrentes proporcionados en clase.
   - Úsalos como guía para comprender la lógica fundamental y las operaciones necesarias.

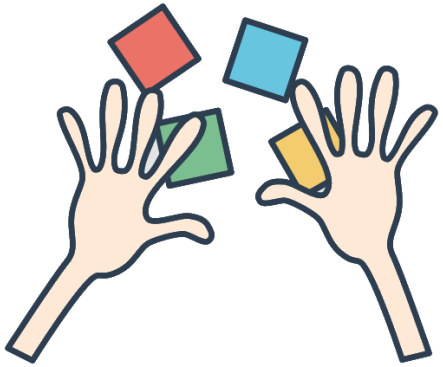
Lab3prob08
Cola Sincronizada



Synchronized Queue

```
class SynchronizedQueue {
    private Node head, tail;
    private final ReentrantLock lock = new ReentrantLock();
    private final Condition notEmpty = lock.newCondition();

    private static class Node {
        int value;
        Node next;

        Node(int value) {
            this.value = value;
            this.next = null;
        }
    }
    public SynchronizedQueue() {
        this.head = this.tail = null;
    }
    public void enqueue(int value) {
        lock.lock();
        try {
            Node newNode = new Node(value);
            if (tail == null) {
                head = tail = newNode;
            } else {
                tail.next = newNode;
                tail = newNode;
            }
            notEmpty.signal(); // Signal any waiting dequeues
        } finally {
            lock.unlock();
        }
    }
    public int dequeue() throws InterruptedException {
            // ADD CODE
    }
    public boolean isEmpty() {
        // ADD CODE
    }
    public int peek() throws InterruptedException {
        // ADD CODE
    }
}
```

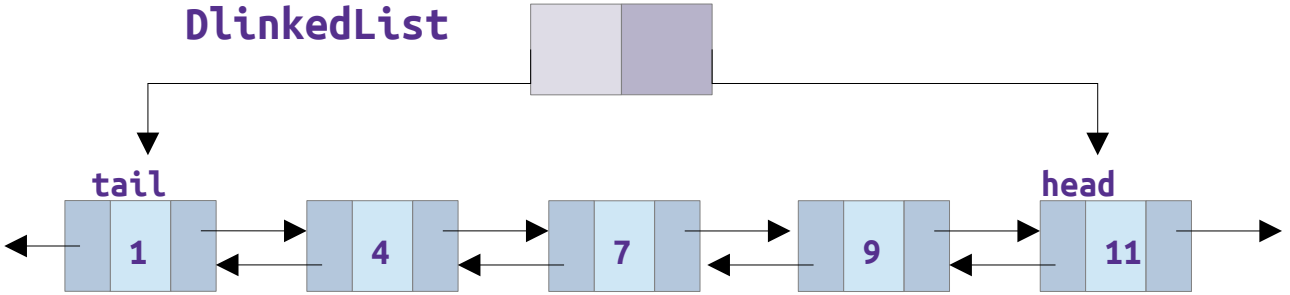
Manos-a-la-obra: Lab3Prog08



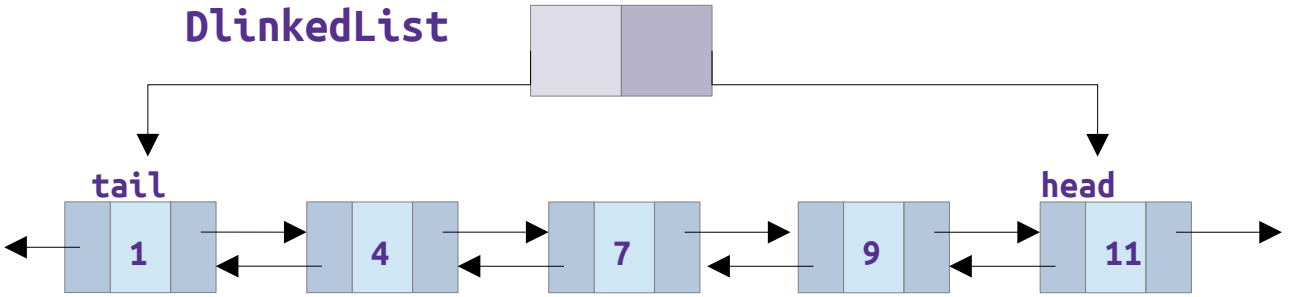
1. Completa la implementación de los métodos del stack concurrente
   - Implementa los métodos que actualmente no están desarrollados.
   - Consulta las notas y ejemplos de métodos de pila no concurrentes proporcionados en clase.
   - Úsalos como referencia para comprender la lógica fundamental y las operaciones necesarias.

Lab3prob09

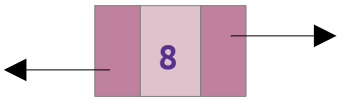
Lista Doble Enlazada Sincronizada



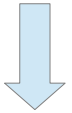
Insertar elemento en la lista doble (ordenada)

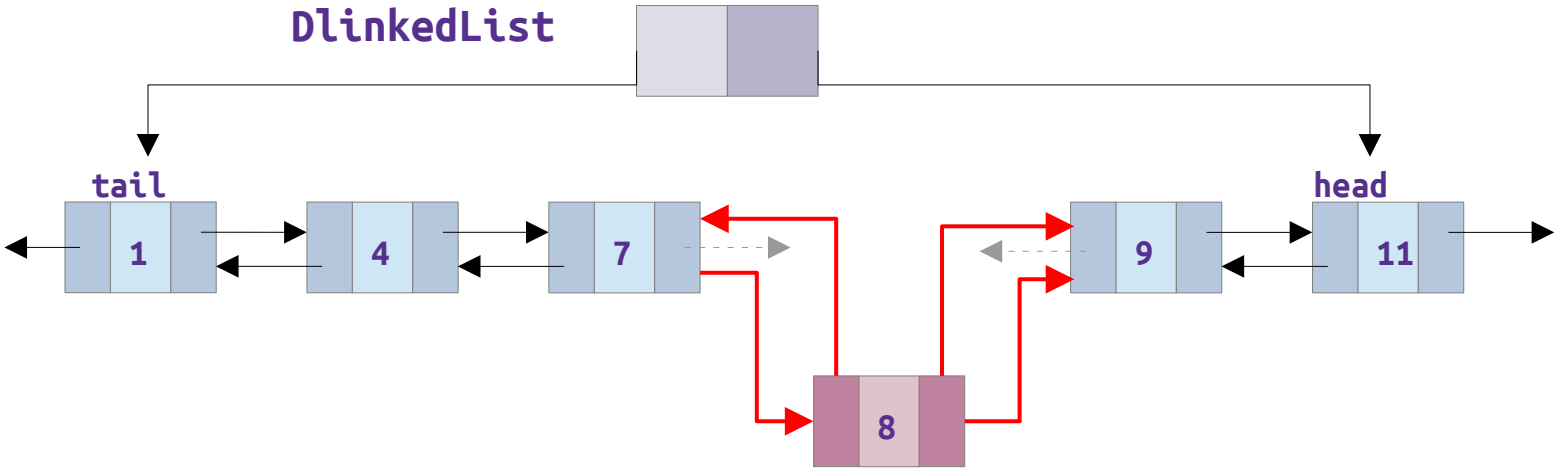


Create:
newNode

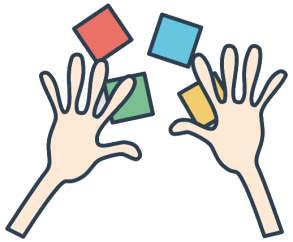


Should go between 7 and 9





Manos-a-la-obra: Lab3Prog09



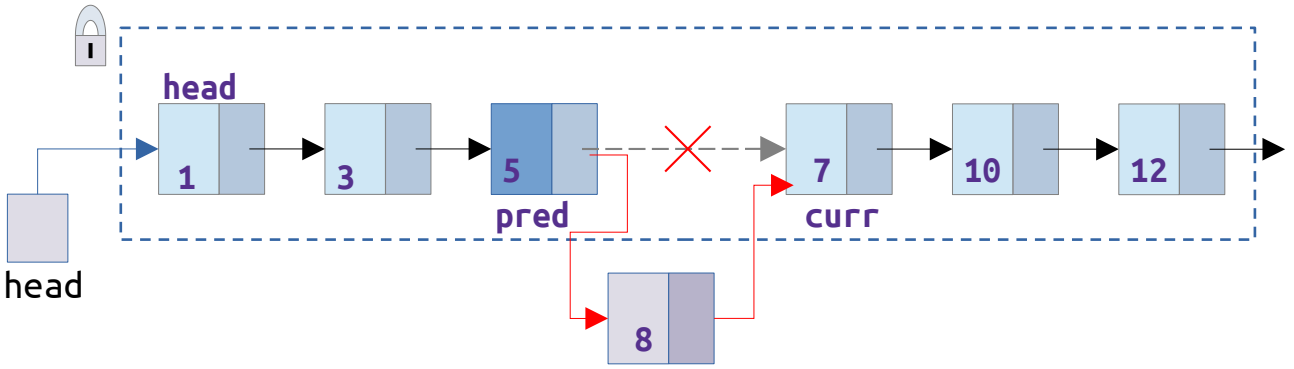
1. Basado en el código de la lista enlazada simple, implementa todo el código para la lista doblemente enlazada concurrente.
   - Asume una lista ordenada e implementa el método insert de la diapositiva anterior.

Hands-on:

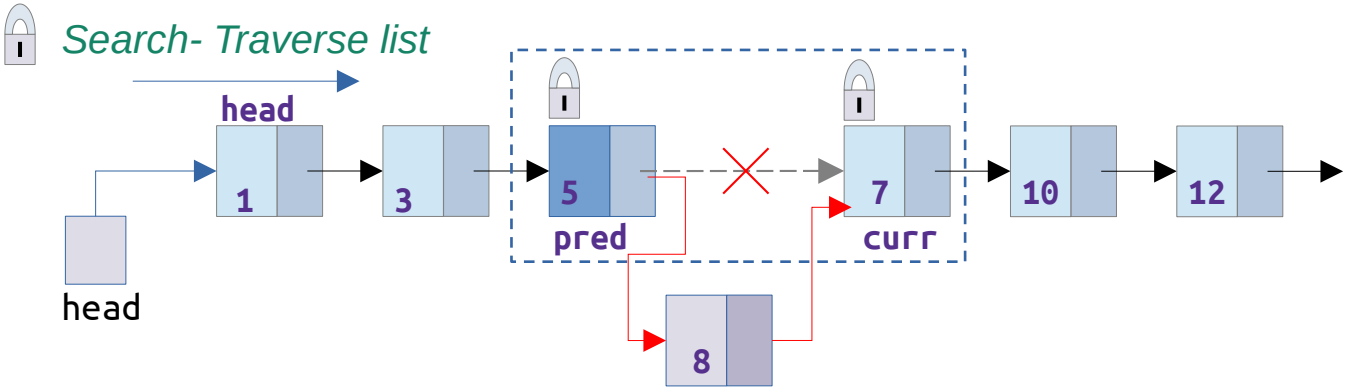
Synchronized List
Lab3Prog**10A** & Lab3Prog**10B**

Algoritmos de Listas Concurrentes

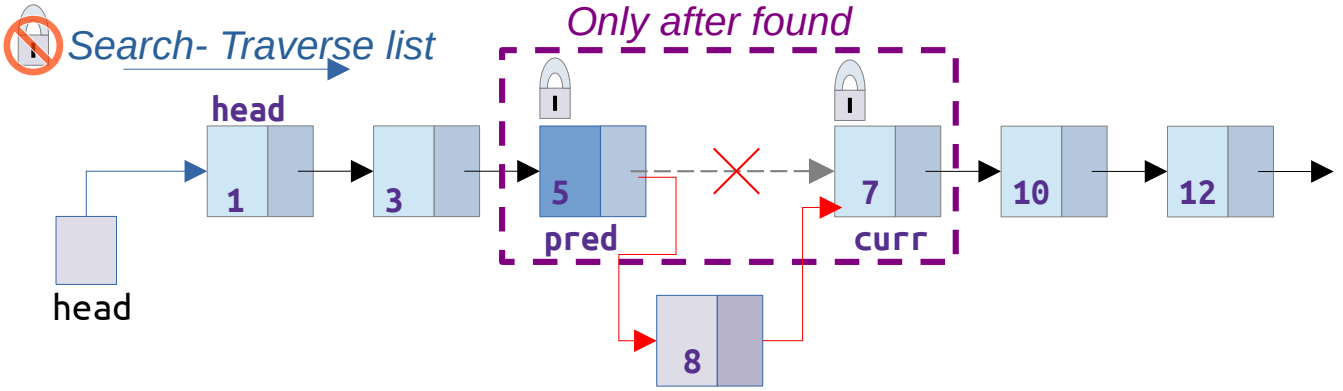
Grano Grueso
(Course Grained)



Grano Fino
(Fine Grained)



Optimista
(Optimistic)

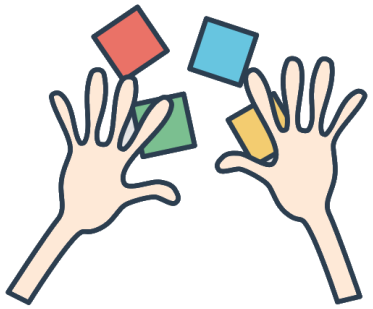


Possible structures

```
public class Node {
    int item;
    Node next;
    Lock lock = new ReentrantLock();

    // Constructor for node
    public Node(int item) {
        this.item = item;
        this.next = null;
    }

    public void lock() {
        lock.lock();
    }

    public void unlock() {
        lock.unlock();
    }
}
```

```
public class LinkedList {
    private Node head;
    public LinkedList() {
        this.head = null;
    }
    public void add(int item) {
        if (head == null) {
            head = new Node(item);
            return;
        }
        Node current = head;
        Node previous = null;
        while (current != null && current.item < item) {
            previous = current;
            current = current.next;
        }
        Node newNode = new Node(item);
        if (previous == null) {
            newNode.next = head;
            head = newNode;
        } else {
            newNode.next = current;
            previous.next = newNode;
        }
    }
    public void printList() {
          // ADD CODE
    }
}
```

Hands-on: Lab4Prog01 & 02



1. Utiliza la implementación de la Lista Enlazada "No Sincronizada" como punto de partida.
   - Añade sincronización al método add().
   - Compara los métodos de sincronización de Grano Grueso (Course Grained), Grano Fino (Fine Grained) y Optimista (Optimistic) discutidos en clase.