# ISMAEL FERNÁNDEZ HERRERUELA

## ACTIVIDAD 1

La solución A es incorrecta ya que si ejecutan 2 hilos el método sem.acquire al mismo tiempo, el semáforo se quedara en -2 y el programa se bloqueara. Para corregirlo podemos inicializar el semáforo a 2 y que cada hilo ejecute sem.acquire y sem.release para así acabar liberando el semáforo.

La solución B es correcta. Se utiliza countdownlatch para esperar a que un numero de hilos lleguen a sincronizarse. Como aquí la barrera esta inicializada en 2, cada hilo llama a countdown y cuando acaban, la barrera se levanta y ambos pueden continuar su ejecución.

La solución C también es correcta ya que se utiliza un monitor y cuando un hilo llama a cita aumenta el valor de w y luego espera a que el valor de w sea igual a 2 mediante wait. Cuando otro hilo llama a cita también aumenta el valor de w y como ahora 2 es igual a 2 se notifica a todos los hilos que estaban en espera utilizando notify.

## ACTIVIDAD 2

1. Al crear un semáforo con valor inicial 0, todos los hilos que intenten adquirir el semáforo se bloquearan hasta que otro hilo lo libere. Además, no es seguro que los hilos B terminen antes de que el Hilo A imprima “fin”. Para corregirlo el valor inicial del semáforo debería ser 3N siendo N el numero de hilos B y el hilo A debería llamar a s.acquire(3N) antes de imprimir “fin”.
2. Al crear un semáforo con valor inicial -N deja que N hilos adquieran el semáforo antes de que el hilo A llame a s.acquire. El valor inicial del semáforo debería ser 3N, además, los hilos B deberían llamar a s.release(3) en vez de s.release para indicar que ha terminado el bucle.
3. La clase cycliccilinder se utiliza para esperar a que un numero determinado de hilos lleguen a un punto de sincronización antes de continuar ejecutándose. En este caso el objeto cyclicbarrier se crea con valoer 1, lo que implica que el hilo A esperara solo a si mismo en lugar de esperar a los hilos B. Para solucionar esto, el valor debería ser n+1 y cada hilo B debería llamar a c.countdown.
4. Se utiliza countdownlatch para esperar a que se completen cierto numero de operaciones. Además el numero de hilos que se están ejecutando no debería incluir A ya que solo estamos esperando a que los hilos B acaben su trabajo. Para solucionarlo todos los hilos podrían compartir el objeto countdownlatch cdl= new countdownlatch(n3) de modo que A realizaría cdl.await antes de imprimir su mensaje y los hilos B realizarían cdl.countdown tras terminar cada iteración del bucle. Cada hilo B debería realizar tres llamadas a cdl.countdown ya que se espera que cada hilo B escriba “trabajando” tres veces

## ACTIVIDAD 3

Las alternativas 1 y 3 son correctas.

La alternativa 2 no es correcta ya que semaphore no se puede usar para esperar a que un conjunto de hilos finalicen. Además, acquire solo disminuye el contador del semáforo pero no espera a que se libere ningún recurso lo que podría causar una condición de carrera.

La alternativa 4 tampoco es correcta ya que el uso de reetrantlock no garantiza que los hilos se completen antes de que se ejecute el código posterior a label 2.

## ACTIVIDAD 4

### A.)

MyMonitor.java

```
1  import java.util.concurrent.Semaphore;
2  import java.util.concurrent.atomic.AtomicReference;
3  public class MyMonitor {
4
5      private Semaphore mtx = new Semaphore (1);
6      private AtomicReference owner = new AtomicReference (null); // reference to Thread.
7
8      public final MyCondition newCondition () {
9          return new MyCondition (this);
10     }
11
12     void checkOwner () {
13         if (!Thread.currentThread().equals (owner.get()))
14             throw new IllegalMonitorStateException();
15     }
16     void checkNotOwner () {
17         if (Thread.currentThread().equals (owner.get()))
18             throw new IllegalMonitorStateException();
19     }
20
21     protected void enterMonitor () {
22         checkNotOwner();
23         mtx.acquireUninterruptibly ();
24         owner.set(Thread.currentThread());
25     }
26     protected void exitMonitor () { checkOwner(); owner.set(null); mtx.release (); }
27
28     protected void runMonitor ( Runnable r ) {
29         enterMonitor ();
30         try {
31             r.run();
32         } finally {
33             exitMonitor();
34         }
35     }
36
37 }
```

entermonitor(): Se obtiene el mutex y la tarea actual se establece como propietaria del monitor mediante el método enterMonitor(). Se genera una IllegalMonitorStateException si la tarea actual ya es el propietario del monitor.

exitmonitor():El mutex debe liberarse y el propietario del monitor debe establecerse en nulo mediante el método exitMonitor(). Se genera una IllegalMonitorStateException si la tarea actual no es el propietario del monitor.

Runmonitor():Los objetos ejecutables se utilizan como argumento para el método runMonitor(Runnable r), que los ejecuta dentro del monitor. Después de adquirir la exclusión

mutua llamando a enterMonitor(), se lleva a cabo la tarea y luego se llama a exitMonitor() para liberar la exclusión mutua.

newcondition():este método crea una nueva instancia de MyCondition, una clase de condición que se puede usar para detener subprocesos hasta que se cumpla una condición específica.

MyCondition.java

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

public class MyCondition {
    private MyMonitor mon;
    private Semaphore cond = new Semaphore (0);
    private int nwait = 0;

    public MyCondition (MyMonitor mon) {
        this.mon = mon;
    }

    public void await () throws InterruptedException {
        mon.checkOwner();
        nwait++;
        mon.exitMonitor();
        cond.acquire(); // Can throw InterruptedException
        mon.enterMonitor();
    }

    public void signalAll () {
        mon.checkOwner();
        if (nwait > 0) cond.release (nwait);
        nwait=0;
    }
}
```

Mycondition y Mymonitor son las únicas dos clases en el código proporcionado. La combinación de estas clases permite la implementación de Java de una variable de condición para la comunicación entre procesos.

La clase Mymonitor proporciona un monitor que puede ser adquirido y liberado por varios subprocesos. Entermonitor y Exitmonitor son los dos métodos disponibles en esta clase. Un subproceso que llama al método entermonitor () se bloquea hasta que se libera el monitor después de haberlo adquirido. El método exitmonitor () libera el monitor y desbloquea el subproceso que lo llamó.

Para bloquear y desbloquear subprocesos que esperan en un monitor, la clase Mycondition ofrece una variable de condición. Await y Signalall son dos de los métodos de esta clase. Cuando una llamada al método Signalall () libera la variable de condición, el método Await () bloquea el subproceso que lo llamó. Todos los subprocesos que están esperando en la variable de condición son desbloqueados por el método Signalall ().

La clase MyCondition utiliza un objeto Semaphore que se inicializa con un valor de 0 en el generador para implementar la variable de condición. Los subprocesos que están esperando en la variable de condición se bloquean y luego vuelven a estar disponibles utilizando el objeto semáforo.

En conclusión, Mymonitor se encarga de sincronizar y excluirse mutuamente los subprocesos, mientras que Mycondition se utiliza para permitir que los subprocesos esperen y se despierten en respuesta a una condición. Juntas, estas clases brindan a los programadores de Java una forma rápida de implementar la comunicación entre subprocesos utilizando variables de condición y monitores.

```
public class Caja extends MyMonitor{

    private int elems=0, cabeza=0, cola=0;
    private int[] datos;
    private MyCondition cond;

    public Caja (int size) {
        datos = new int [size];
        cond = newCondition ();
    }

    public int get () {
        enterMonitor();

        while (elems==0) {
            System.out.println("consumidor esperando");
            try {cond.await();} catch(InterruptedException e) {}
        }
        int x=datos[cabeza]; cabeza=(cabeza+1) % datos.length; elems--;

        cond.signalAll();
        exitMonitor();
        return x;
    }

    public void put (int x) {
        runMonitor ( () -> {
            while (elems==datos.length) {
                System.out.println("productor esperando");
                try {cond.await();} catch(InterruptedException e) {}
            }
            datos[cola]=x; cola=(cola+1)%datos.length; elems++;
            cond.signalAll();
        });
    }
}
```

El código proporcionado demuestra cómo implementar una clase de Java Box que se utiliza para almacenar elementos en una estructura de datos de tamaño fijo. Los subprocesos de productor y consumidor pueden comunicarse y sincronizarse entre sí utilizando los monitores y las variables de condición de la clase Box.

El constructor de la clase Box acepta un parámetro de tamaño y genera una nueva matriz de enteros con el tamaño dado. Utilizando el método newCondition() de Java, también inicializa una variable de condición. utilidad concurrente Cerraduras. Interfaz para condiciones.

Se puede obtener un elemento Box usando el método get(). Al llamar a enterMonitor() dentro del método, primero se obtiene el monitor. Usando un bucle while y la variable elems, que realiza un seguimiento de cuántos elementos hay en el cuadro, el método espera hasta que el cuadro esté vacío. El método espera invocando el método await() de la variable de condición si el cuadro está vacío. El subproceso se bloquea hasta que otro subproceso señala que hay elementos disponibles en el cuadro cuando se llama await() para desbloquear el monitor.

El método recupera el primer elemento del cuadro y lo elimina de la matriz cuando se activa. Luego, el método signalAll() se usa para informar a todos los subprocesos en espera que se eliminó un elemento y liberar el bloque en la variable de condición. Luego llama a exitMonitor() para liberar el monitor y devuelve el elemento devuelto.

Para agregar un elemento al cuadro, use el método put(). La adquisición y liberación automáticas del monitor se manejan dentro del método mediante el método runMonitor(). Usando un ciclo while y los elementos variables, el método espera hasta que el Box esté completamente lleno. Cuando el cuadro está lleno, el método espera invocando el método await() de la variable de condición. Cuando se llama, el método agrega el elemento al final de la matriz, señala que se ha agregado un elemento y desbloquea todos los subprocesos que están esperando la variable de condición usando el método signalAll().

En pocas palabras, la clase Box ofrece una forma de almacenar elementos en una estructura de datos de tamaño fijo utilizando monitores y variables de condición para que los subprocesos de productor y consumidor puedan interactuar y sincronizarse.

**B.)** 1. Se debe asegurar la exclusión mutua en los métodos que lo manipulan (enterMonitor, exitMonitor, checkOwner y checkNotowner) para resolver la condición de carrera en el acceso a la variable "propietario" en la clase MyMonitor. Para sincronizar el acceso a estas partes cruciales del código, se puede usar el semáforo "mtx".

2. El adagio es exacto. Una llamada a enterMonitor lanzará la excepción IllegalMonitorStateException si se realiza inmediatamente después de una llamada a exitMonitor sin llamar primero a exitMonitor. Para evitar esto, debe asegurarse de que cada llamada para ingresar al monitor tenga una llamada posterior para salir del monitor en algún momento.

3. La afirmación es precisa. Podría haber condiciones de carrera en su acceso si la variable "nwait" en la clase MyCondition se usa sin protección de pestillo, monitor o semáforo. El semáforo "cond" se puede usar para sincronizar el acceso a la variable "nwait" para evitar esto.

4. Esta afirmación es falsa. Las invocaciones cruzadas no generan interbloqueos en la implementación de MyMonitor, que utiliza semáforos para sincronizar el acceso a secciones de código cruciales. En realidad, los interbloqueos se evitan gracias a la exclusión mutua que asegura el semáforo "mtx".

5. Falso. El método "runMonitor" de la clase Box puede usar legítimamente una expresión lambda. El método "runMonitor" acepta un objeto Runnable, que puede ser una instancia de una clase que implementa la interfaz Runnable o una expresión lambda que se evalúa como un objeto Runnable.

## ACTIVIDAD 5

**A.)**El subproceso productor (p) usa el método put de la caja diez veces para poner el valor 1 en la caja.

El subproceso de consumo (c) utiliza el método get del cuadro para recuperar un valor antes de imprimirlo 10 veces en la consola. Dentro del bucle, hay dos llamadas a randomSleep

que agregan un retraso arbitrario de entre 0 y 5 milisegundos.

El subproceso principal imprime "adiós" en la consola después de que ambos subprocesos hayan terminado de ejecutarse.

**B.)** En la 2 y la 5 no es necesario corregir nada.

1. Para garantizar la exclusión mutua entre los hilos productor y consumidor al acceder a la caja compartida, se debe agregar algún mecanismo de sincronización, como un objeto Lock y un par de Condiciones. También se debe realizar un seguimiento del número de artículos en la caja, y el consumidor debe esperar hasta que haya al menos un artículo en la caja antes de intentar abrirla. Además, debe modificar la forma en que se implementa el método put para que el subproceso del productor espere si la caja está llena y no puede aceptar más elementos. Se puede realizar incluyendo una espera condicional en el caso de que la caja esté llena.

3. Si el cuadro está vacío y no puede recibir más elementos, se debe cambiar la implementación del método get para permitir que el subproceso de consumo espere. En caso de que la casilla esté vacía, se puede lograr incluyendo una espera condicional.

4. Para garantizar que el ciclo del consumidor espere si la casilla está vacía, se debe agregar una espera condicional.

## ACTIVIDAD 6

**A.)**La clase tiene tres métodos: randomSleep(), produce() y consuma(). produce() crea elementos enteros en el cuadro del valor a al b (inclusive) usando un cuadro (Box) y dos valores enteros a y b. También inserta un intervalo de sleep aleatorio (usando randomSleep()) entre cada elemento producido. consuma() consume artículos de la caja, imprime el nombre del consumidor y el artículo consumido en cada iteración y agrega una pausa aleatoria entre cada iteración. Se necesita una caja, un nombre de consumidor y una cantidad de artículos para consumir. Simplemente tome un descanso aleatorio con randomSleep().

Se crea una instancia de Box con capacidad para tres elementos en el método main(), junto con cuatro subprocesos: dos para producir elementos y dos para consumirlos. Debido a las pausas aleatorias entre la producción y el consumo de elementos, cada subproceso se ejecuta en paralelo y la ejecución completa del programa puede generar un conjunto diferente de resultados.

**B.)** 1. Para que esta afirmación sea cierta, el manejo de la excepción "InterruptedException" debe eliminarse de las ubicaciones donde se maneja actualmente. Sin embargo, esto no se recomienda, ya que la excepción podría generarse en algunas circunstancias. En general, manejar las excepciones es una buena idea incluso si es poco probable que sucedan.

2. Debido a que el programa principal imprime "adiós" y termina sin esperar a que se completen los hilos infantiles, esta declaración es precisa. Antes de imprimir "adiós", podríamos rectificar esto invocando el método "unirse" en cada hilo infantil.

3. Esta afirmación no es cierta porque los subprocesos funcionan de forma asíncrona y no hay garantía de que los mensajes recibidos se impriman en el orden correcto. Dependiendo de cómo se sincronicen los subprocesos, también se pueden mostrar mensajes de "productor en espera" o "consumidor en espera" en cualquier momento.

4. Esta declaración puede ser precisa en algunas circunstancias porque puede haber momentos en que el consumidor está esperando y el productor aún no ha terminado de crear el contenido de la caja. Sin embargo, no podemos prometer que 4 mensajes de "espera del consumidor" se imprimirán en una fila.

5. Esta afirmación es cierta, ya que "bye" es el último mensaje que se imprime en el programa después de haber iniciado los hilos secundarios.