ACTIVIDADES DE AUTOCORRECCIÓN - UNIDADES 4 Y 5

ACTIVIDAD 1 – Condiciones de Coffman y soluciones a la gestión de interbloqueos

1.1 Las estrategias de prevención de interbloqueos se basan en romper alguna de las condiciones de Coffman, para así evitar la posible existencia de interbloqueos. Indique a continuación, por cada condición de Coffman, un ejemplo de solución que permita romper con dicha condición o, en su caso, qué requisitos se deberían cumplir para poder romper dicha condición.

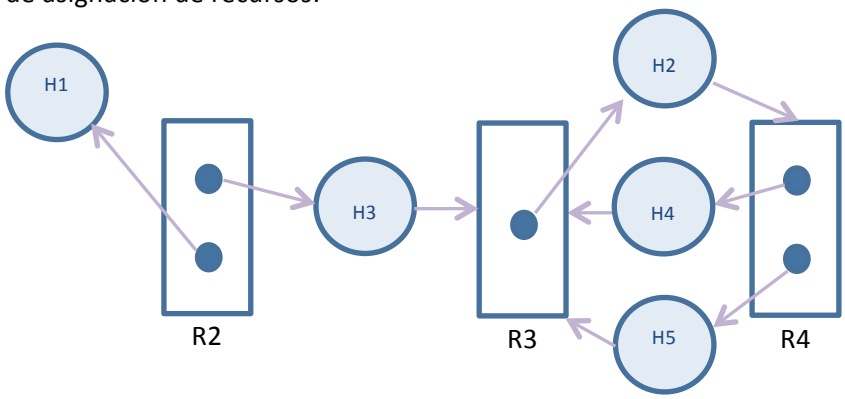
ACTIVIDAD 2 – Grafo de Asignación de Recursos

2.1 Explique qué es un *grafo de asignación de recursos*, describiendo qué elementos lo componen y qué representan.

2.2 ¿Para qué se utiliza un grafo de asignación de recursos? Indique en qué tipo de estrategias de gestión de interbloqueos se utilizan.

ACTIVIDAD 3 – GAR

Dado el siguiente grafo de asignación de recursos:



3.1 Justifique si dicho grafo presenta una situación de interbloqueo o no. Para ello, en caso de que haya interbloqueo indique qué hilos están interbloqueados y en caso de que no lo haya, indique por qué.

3.2 Si el recurso R3 tuviera dos instancias, una de ellas asignada a H2 y la otra a H3 (por lo que H3 tendría todas sus solicitudes resueltas), indique si el sistema presentaría o no una situación de interbloqueo.

ACTIVIDAD 4 – biblioteca *java.util.concurrent*

Asuma que existe una clase Buffer con capacidad para 10 elementos, cuyo método *put*() permite insertar un elemento de tipo **int** (bloqueando al invocador si no hubiera espacio) y cuyo método *get*() extrae un elemento. Ese método *get*() suspende a quien lo invoque cuando haya menos de cinco elementos. Se desea que el hilo principal del programa que se muestra a continuación escriba su mensaje cuando hayan finalizado su ejecución todos los hilos de la clase Worker utilizados en ese programa.

```
public class Main {
    public static void main(String[] args)  {
       Buffer d = new Buffer();
       Worker w1 = new Worker(d, 1);
       Worker w2 = new Worker(d, 2);
       w1.start();
       w2.start();

      //El hilo principal debe esperar

       System.out.println("Buffer item: " + d.get());
    }
}
```

```
public class Worker extends Thread {
      private Buffer buf;
      private int number;

   public Worker(Buffer c, int id) {
       buf = c;
       number = id;
    }
    public void run()  {
       for (int i = 1; i < 101; i++)
       {
         buf.put(number*100 + i);
         if (i > 3)
           System.out.println("Worker #" + number +
           " gets: " + buf.get());
       } // for
   } // run
}
```

4.1 Proporcione una solución al problema anterior, haciendo uso de la clase *Semaphore* de la biblioteca *java.util.concurrent*

4.2 Proporcione una nueva solución al citado problema, haciendo esta vez uso de la clase *CountDownLatch* de la biblioteca java.util.concurrent.

4.3 Muestre ahora cómo el hilo principal se esperará a que finalicen los hilos de la clase Worker, haciendo uso de la clase *CyclicBarrier* de la biblioteca java.util.concurrent

ACTIVIDAD 5 – biblioteca java.util.concurrent

En la actividad anterior se mencionaba un Buffer con capacidad para 10 elementos, cuyo método *put*() permite insertar un elemento de tipo **int** (bloqueando al invocador si no hubiera espacio) y cuyo método *get*() extrae un elemento. Ese método *get*() suspende a quien lo invoque cuando haya menos de cinco elementos.

Se desea implementar dicho Buffer mediante una clase que contenga los métodos *put*() y *get*() con la funcionalidad previamente indicada.

5.1 Proporcione una implementación adecuada para dicha clase Buffer, haciendo uso de las clases *ReentrantLock* y *Condition*.

5.2 Proporcione una solución adecuada para la clase Buffer, implementando la exclusión mutua y la sincronización condicional con la ayuda de la clase *Semaphore*.

SOLUCIONES A LAS ACTIVIDADES

ACTIVIDAD 1

A continuación se indican las condiciones de Coffman y un ejemplo de solución para romper con dicha condición:

Condición de Coffman	**Para romperla....**
Exclusión mutua	Se debería permitir que un recurso pudiera ser utilizado por varios hilos a la vez. Normalmente esto no se permite por las propias restricciones del problema (es decir, que los recursos suelen ser no compartibles) así que esta condición no se rompe.
Retención y espera	Opción a) Se debería pedir desde el principio todo lo que podemos llegar a necesitar. Por ejemplo, en el caso de los 5 filósofos, un filósofo coge a la vez los dos tenedores o bien no coge ninguno. Opción b) Al solicitar un recurso, no bloquear al hilo si el recurso está ocupado, sino que se le devuelve un valor informándole de esa situación. Para ello, se tendría que emplear sentencias del tipo tryLock(). Además, si el hilo no pudiera obtener todos los recursos que necesita, debería liberar los que ya hubiera obtenido.
No expulsión	Se debería permitir que un hilo se apropiara de los recursos de otro hilo. Normalmente esto no es aceptable por las condiciones del problema. Por ejemplo, en el caso de los 5 filósofos, no se permite que un filósofo le robe un tenedor a otro filósofo. Así que esta condición de Coffman no se suele romper. Además, de romperla, se puede provocar la situación de "livelock", donde los hilos se están expropiando recursos mutuamente, de modo que nunca consiguen avanzar en su ejecución.
Espera circular	Se establece un orden total entre los recursos y se obliga a solicitarlos en ese orden. O bien se aplican soluciones donde se rompa la simetría existente en la ejecución de los hilos. Por ejemplo, en el caso de los 5 filósofos, se puede establecer que todos los hilos cojan los tenedores en un determinado orden (i.e. derecho – izquierdo), excepto el último filósofo, que lo hará en el orden contrario (i.e. izquierdo – derecho). O bien, que los hilos pares cojan los tenedores en orden derecho-izquierdo, mientras que los impares lo hagan en orden contrario (i.e. izquierdo-derecho).

ACTIVIDAD 2

2.1 En un grafo de asignación de recursos se representan:

- Los recursos existentes en el sistema, indicando para cada uno de ellos las instancias que tienen. Cada recurso se dibuja mediante un rectángulo, que tiene dentro tantos puntos como instancias del recurso.
- Los procesos/tareas/hilos del sistema, dibujados mediante un círculo.
- La solicitud de instancias que realizan los hilos sobre los recursos. Si la instancia del recurso ha sido asignada al hilo, dicha asignación se representa con una flecha que va

desde la instancia hacia el hilo. En caso contrario (i.e. instancia no asignada), se dibuja una flecha desde el hilo hacia el recurso.

2.2 El grafo de asignación de recursos permite mostrar la asignación de recursos a hilos, así como las solicitudes no resueltas (de modo que los hilos deben esperar). Si existe un ciclo dirigido en dicho grafo, esto implica un riesgo de interbloqueo. Si se puede encontrar un orden de terminación de los hilos (secuencia segura), entonces no se producirá ningún interbloqueo. En caso contrario (i.e. hay un ciclo dirigido y no se encuentra ninguna secuencia segura), sí que hay interbloqueo.

Los grafos de asignación de recursos (GAR) se utilizan en las estrategias de evitación de interbloqueos. Para ello, el sistema, usando un GAR, considera cada solicitud y decide si es seguro o no atenderla en ese momento. Para ello, simula con el GAR qué pasaría si se atendiese la solicitud, de modo que si se produce un ciclo dirigido se denegará la solicitud.

ACTIVIDAD 3

3.1 Existe un ciclo dirigido, por ejemplo H2-> R4 -> H5 -> R3 -> H2, por lo que hay riesgo de interbloqueo. Si buscamos una secuencia de terminación de los hilos (secuencia segura), observamos que primero puede acabar el hilo H1 (pues tiene todos sus recursos asignados), pero ya no podría acabar ningún hilo más. El resto de hilos H3, H2, H4, H5 estarían reteniendo una instancia de un recurso y esperando otra. Por tanto, al no encontrar la secuencia segura, el sistema está en un interbloqueo. Los hilos interbloqueados son: H3, H2, H4 y H5.

3.2 En este caso, tanto H1 como H3 podrían finalizar sin problemas. Al terminar H3 liberaría una instancia de R3, que podría ser asignada a H4 o bien a H5. Si, por ejemplo, es asignada a H5, ese hilo también tendría ya todos sus recursos asignados y podría finalizar, liberando de nuevo la instancia de R3, que sería asignada a H4. Por su parte, al disponer H4 de una instancia de R3 y otra de R4, también podría finalizar. Por último, cuando terminen tanto H4 como H5, se liberará una instancia de R4, que podrá ser asignada a H2 y este hilo podrá así finalizar también.

En definitiva, si se añade una instancia al recurso R3, entonces sí que se puede encontrar una secuencia segura. Es más, existen diferentes secuencias posible. Por ejemplo:

<H1, H3, H5, H4, H2>, o bien <H1, H3, H4, H5, H2>, o incluso <H1, H3, H5, H2, H4>, o bien <H3, H1, H4, H2, H5>, etc.

ACTIVIDAD 4

4.1 Solución usando la clase Semaphore (se remarca en negrita los cambios realizados):

```
import java.util.concurrent;
public class Main {
   public static void main(String[] args)  {
      Buffer d = new Buffer();
      Semaphore c = new Semaphore(0);
      Worker w1 = new Worker(d, 1, c);
      Worker w2 = new Worker(d, 2, c);
      w1.start();
      w2.start();

     //El hilo principal debe esperar
     c.acquire();
     c.acquire();
      System.out.println("Buffer item: " + d.get());
   }
}
```

```
public class Worker extends Thread {
     private Buffer buf;
     private int number;
     private Semaphore sem;

   public Worker(Buffer c, int id, Semaphore s) {
      buf = c;
      number = id;
      sem=s;
    }
    public void run()  {
      for (int i = 1; i < 101; i++)
      {
        buf.put(number*100 + i);
        if (i > 3)
          System.out.println("Worker #" + number +
          " gets: " + buf.get());
      } // for
     sem.release();
  } // run
}
```

Nota: Java permite una solución alternativa que consiste en adquirir el semáforo indicando la cantidad de permisos que se desean. En este caso:

c.acquire(2);

4.2 Solución usando la clase CountDownLatch

```
public class Main {
   public static void main(String[] args)  {
      Buffer d = new Buffer();
      CountDownLatch c=new CountDownLatch(2);
      Worker w1 = new Worker(d, 1, c);
      Worker w2 = new Worker(d, 2, c);
      w1.start();
      w2.start();

     //El hilo principal debe esperar
     c.await();
     System.out.println("Buffer item: " + d.get());
   }
}
```

```
public class Worker extends Thread {
     private Buffer buf;
     private int number;
     private CountDownLatch cdl;

   public Worker(Buffer c, int id, CountDownLatch a ) {
      buf = c;
      number = id;
      cdl=a;
    }
    public void run()  {
      for (int i = 1; i < 101; i++)
      {
        buf.put(number*100 + i);
        if (i > 3)
          System.out.println("Worker #" + number +
          " gets: " + buf.get());
      } // for
     cdl.countDown();
  } // run
}
```

4.3 Solución usando la clase CyclicBarrier

```
public class Main {
    public static void main(String[] args)  {
       Buffer d = new Buffer();
       CyclidBarrier c=new CyclidBarrier(3);
       Worker w1 = new Worker(d, 1, c);
       Worker w2 = new Worker(d, 2, c);
       w1.start();
       w2.start();

      //El hilo principal debe esperar
       c.await();
       System.out.println("Buffer item: " + d.get());
    }
}
```

```
public class Worker extends Thread {
      private Buffer buf;
      private int number;
      private CyclidBarrier cb;

   public Worker(Buffer c, int id, CyclidBarrier a ) {
       buf = c;
       number = id;
       cb=a;
    }
    public void run()  {
       for (int i = 1; i < 101; i++)
       {
         buf.put(number*100 + i);
         if (i > 3)
           System.out.println("Worker #" + number +
           " gets: " + buf.get());
       } // for
       cb.await();
   } // run
}
```

ACTIVIDAD 5

5.1 Implementación de la clase Buffer utilizando ReentrantLock y Condition. Se ha seguido el ejemplo de las trasparencias de la Unidad 5.

```
public class Buffer {
    private int[] data;
    private int elems, head, tail, N;
   Condition notFull, notMinimum;
    ReentrantLock rl;

 public Buffer(int N){
   data= new int[N];
   this.N=N;
   head=tail=elems=0;
   rl=new ReentrantLock();
   notFull=lock.newCondition();
   notMinimum=lock.newCondition();
}
```

```
public int get()
{  int x;
   try{
      rl.lock();
     while(elems<5){   //suspende si hay menos de 5 elementos
             try { notMinimum.await();
             } catch(InterruptedException e) {} }
     x=data[head];
    head= (head+1)%N;     elems--;
    notFull.signal();
 } finally { rl.unlock();}
  return x;
}

public void put(int x) {
 try{
    rl.lock();
    while (elems==N) {
        try {notFull.await();
        }  catch(InterruptedException e) {} }
   data[tail]=x; tail= (tail+1)%N; elems++;
  if (elems>=5)
     notMinimum.signal();
 } finally {rl.unlock();}
}
```

Nota: En el método put(), la condición if (elems>=5) se podría eliminar, dejando la instrucción notMinimum.signal(). En dicho caso, el código sería un poco menos eficiente, pues si en el buffer hay menos de 5 elementos y existe algún hilo suspendido por ejecutar el método get(),

dicho hilo se reactivará cada vez que se inserte un elemento en Buffer y se volverá a suspender, hasta que el número de elementos sea igual o superior a 5.

5.2 Solución utilizando la clase Semaphore. Se ha seguido el ejemplo de las trasparencias de la unidad 5.

```
public class Buffer {
    private int[] data;
    private int elems, head, tail, N;
    private Semaphore mutex, item, slot;

 public Buffer(int N){
   data= new int[N];
   this.N=N;
   head=tail=elems=0;

    mutex=new Semaphore(1, true);
    item=new Semaphore(0,true);
    slot= new Semaphore(N, true);
}
```

```
public int get()
{ try {item.acquire();} catch (InterruptedException e) {}
  try {mutex.acquire();} catch (InterruptedException e) {}
  x=data[head];  head= (head+1)%N;     elems--;
  mutex.release();
  slot.release();
  return x;
}

public void put(int x) {
 try {slot.acquire();} catch (InterruptedException e) {}
 try {mutex.acquire();} catch (InterruptedException e) {}
 data[tail]=x; tail= (tail+1)%N; elems++;
  if (elems>=5)  item.release();
  return x;
}
```

Nota: aquí sí que es necesario realizar la condición if(elems>=5) del método put(), ya que se debe reactivar a hilos suspendidos en el semáforo *item* solamente cuando haya 5 o más elementos en el buffer.