Universidad de Murcia
INGENIERÍA INFORMÁTICA

PROGRAMACIÓN CONCURRENTE Y DISTRIBUIDA

Boletín de Prácticas

Autor: Victorio J. Molina Bermejo
DNI: 48632380-F Grupo: 1.3
Profesor: Francisco Javier Marín-Blázquez Gómez

FACULTAD DE INFORMÁTICA Junio 2020

1. Ejercicio 1

1.1. Recursos no compartibles

Los recursos compartidos no compartibles, y que por tanto necesitan ser manejados en exclusión mutua son:

1. Pantalla
2. Suma Total donde cada hilo añade su suma parcial

1.2. Pseudocódigo

```
1  Recursos compartidos no compartibles
2  Pantalla
3  Suma_Total (inicializada a 0)
4
5  Recursos compartidos si compartibles
6  Buffer de 5000 elementos enteros (Solo lectura)
7
8  main() {
9      Creo Cerrojo ST asociado a Suma_Total
10     Creo Cerrojo P asociado a la Pantalla
11     Buffer = InicializarBufferAleatoriamente()
12     Imprimir por pantalla el Buffer en formato correcto
13     Creo, inicializo y lanzo los 20 hilos(id, Buffer) // id toma el valor de 0 a 19
14     Esperar a que todos los hilos terminen
15     Mostrar por pantalla la suma total valores pares
16 }
17
18
19 Hilo sumadorPares(id, Buffer)
20 Variable Contador_Analizados (inicializada a 0)
21 Variable Suma_Parcial (inicializada a 0)
22 Variable Indice que hace de apuntador a la ultima celda tratada correspondiente al proceso
23 (inicializada a 250*id)
24 Repetir {
25     Generar un valor aleatorio X <= 250
26
27     Recorrer Buffer desde Indice mientras Contador_Analizados != X
28     y para cada elemento Hacer
29         Si el elemento es Par Entonces
30             Suma_Parcial += elemento
31         Fin_Si
32         Contador_Analizados += 1
33         // Movemos el apuntador una celda a la derecha
34         Indice += 1
35     Fin_Recorrer
36 } Hasta que Contador_Analizados sea igual a 250
37
38 // Comienzo Seccion Critica de Suma_Total
```

```
39 ST.lock()
40 Suma_Total += Suma_Parcial
41 // Final Seccion Critica de Suma_Total
42 ST.unlock()
43
44 // Comienzo Seccion Critica de Pantalla
45 P.lock()
46 Imprimir Resultados (Suma_Parcial) Por Pantalla
47 // Final Seccion Critica de Pantalla
48 P.unlock()
49 Fin_Hilo
```

1.3. Cuestiones planteadas

Tras una primera versión de pseudocódigo, a la hora de ponerme a programarlo, me he planteado la creación de dos clases 'Screen' y 'SharedVariable' que me permitiesen facilitar la lectura y compresión del código.

1.4. Código

```
1  package Exercise1;
2
3  import java.util.Random;
4  import java.util.concurrent.locks.Lock;
5  import java.util.concurrent.locks.ReentrantLock;
6
7
8  // Class in charge of accumulating the sum of the pairs
9  class PairsAdder extends Thread {
10     private static final int ANALYZED_LIMIT_COUNT = 250;
11
12     private int id;
13     private int [] buffer;
14     private SharedVariable totalSum;
15     private int parcialSum;
16     private int analyzedCounter;
17     private int bufferPointer;
18
19     public PairsAdder(int id, int [] buffer, SharedVariable totalSum) {
20         this.id = id;
21         this.buffer = buffer;
22         this.totalSum = totalSum;
23         parcialSum = 0;
24         analyzedCounter = 0;
25         bufferPointer = id * 250;
26     }
27
28     public void run() {
29         while (analyzedCounter < ANALYZED_LIMIT_COUNT) {
30             // Setting the  number of elements to analyze
31             int analyzeNumber = new Random(System.currentTimeMillis())
```

```
32                                      .nextInt(ANALYZED_LIMIT_COUNT) + 1;
33
34              while (analyzedCounter < analyzeNumber) {
35                  // If the element is pair
36                  if(buffer[bufferPointer] %2 == 0) {
37                      // Recalculate the parcial sum
38                      parcialSum += buffer[bufferPointer];
39                  }
40
41                  // Increase the number of analyzed elements
42                  analyzedCounter++;
43
44                  // Increase the buffer pointer
45                  bufferPointer++;
46              }
47          }
48
49          // Increasing the total sum
50          totalSum.increase(parcialSum);
51
52          // Show the result on screen
53          String result = String.format("%s\n%s\n%s\n"
54                  ,"Thread " + id + " -----"
55                  , "Parcial sum = " + parcialSum
56                  , "End thread " + id + " -----"
57          );
58
59          Screen.print(result);
60      }
61  }
62
63
64  // Class for creating incremental shared variables
65  class SharedVariable {
66      private final Lock l = new ReentrantLock();
67
68      private int value;
69
70      public SharedVariable(int value) {
71          this.value = value;
72      }
73
74      public void increase(int increment){
75           int temp;
76
77           l.lock();
78           try{
79               temp = value;
80               try {
81                   Thread.sleep((int) Math.round(Math.random()));
82               } catch (InterruptedException e) {
83                   e.printStackTrace();
84               }
85               temp += increment;
86               value = temp;
87           } finally{
88               l.unlock();
```

```
89              }
90          }
91
92          public int getValue(){
93              return value;
94          }
95      }
96
97
98      // Class that represents the screen
99      class Screen {
100         private static final Lock l = new ReentrantLock();
101
102         public static void print(String str) {
103             l.lock();
104             System.out.println(str);
105             l.unlock();
106         }
107     }
108
109
110     // Main class
111     public class AddPairs {
112         /**
113          * Main method that runs when the program is started.
114          */
115         public static void main(String[] args) {
116             // Generating the random buffer
117             int [] buffer = new Random().ints(5000, 0, 10).toArray(); // 5000
                    elements from 0 to 10
118
119             // Printing the buffer with a good visual format
120             System.out.println("Array -----");
121             int sum = 0;
122             for (int i = 0; i < buffer.length; i++) {
123                 if (buffer[i] %2 == 0)
124                     sum += buffer[i];
125
126                 System.out.print(buffer[i] + " ");
127                 if ((i+1) %20 == 0)
128                     System.out.println();
129             }
130             System.out.println("Total sum of pairs: " + sum);
131             System.out.println();
132
133             // Creating the shared variable
134             SharedVariable totalSum = new SharedVariable(0);
135
136             // Creating and starting 20 Threads of PairsAdders
137             PairsAdder[] pairsAdders = new PairsAdder[20];
138             for (int i = 0; i < pairsAdders.length; i++) {
139                 pairsAdders[i] = new PairsAdder(i, buffer, totalSum);
140                 pairsAdders[i].start();
141             }
142
143             // Wait all threads to finish
144             for (PairsAdder pairsAdder : pairsAdders) {
```

```
145                 try {
146                     pairsAdder.join();
147                 } catch (InterruptedException e) {
148                     e.printStackTrace();
149                 }
150             }
151 
152             // Printing the total sum os pairs calculated by our threads
153             System.out.println("Calculated total sum of pairs: " +
                    totalSum.getValue());
154         }
155 }
```

1.5. Preguntas

a) (1 punto sobre 10) ¿Qué acciones deben realizar los hilos en exclusión mutua? Justifica la respuesta.

En exclusión mutua, los hilos deben de imprimir los resultados por pantalla (recurso compartido no compartible), para que la salida de un hilo no se vea sobrepuesta por la de otro. Y, además, cada hilo también tiene que incrementar a la suma total (otro recurso compartido no compartible) el resultado de su suma parcial.

b) (1 punto sobre 10) Indica qué acciones para depurar el programa puedes llevar a cabo para comprobar que no se producen problemas debido a la concurrencia en el programa desarrollado. Justifica la respuesta.

Existen muchas formas de depurar el programa para comprobar que no se producen problemas debido a la concurrencia. En mi caso, yo lo que haría sería, una vez imprimido el buffer completo y la suma total de sus valores pares, para cada hilo ejecutar el numero de elementos que analizado y la suma parcial recien calculada para cada iteración del mismo. También, para una mejor visualización de la salida los pondría a dormir tras cada impresión.

Algo más sencillo sería coger el recurso compartido no compartible "Suma Total", el cual solo es incrementado, y mostrarlo por pantalla tras cada actualización. En caso de que este solo coja valores positivos, podríamos deducir que nuestro programa seguramente sea correcto en cuanto a concurrencia.

c) (1 punto sobre 10) Si no usaras ningún mecanismo para sincronización ¿Cómo podría ser la salida en pantalla del programa anterior?

Totalmente concurrente, con la información entremezclada, ya que la pantalla la cogería otro proceso sin haber terminado de imprimir la información el proceso que la estaba usando, y lo que es peor aún, si no existiese un mecanismo de sincronización se produciría un resultado erróneo. Un ejemplo de un trozo de una posible salida sería el siguiente:

Total sum of pairs: **9902**
Thread 6 ——
...
Thread 5 ——
Parcial sum = 448
End thread 5 ——
Calculated total sum of pairs: **448**

2. Ejercicio 2

2.1. Recursos no compartibles

Los recursos compartidos no compartibles, y que por tanto necesitan ser manejados en exclusión mutua son:

Para el apartado A)

1. Pantalla

Para el apartado B)

1. Pantalla
2. Buffer de Cajas Libres con 10 elementos enteros

2.2. Pseudocódigo

Apartado A

```
1  Recursos compartidos no compartibles
2  Pantalla
3
4  Recursos compartidos si compartibles
5  Buffer Cajas[0..9] de 10 cajas
6
7  main() {
8      Random r = new Random(1)
9      Creo Buffer C[0..9] de 10 semaforos (mutex) que gestionan cada caja (inicializados a 1)
10     Creo Semaforo P (mutex) asociado a la pantalla (inicializado a 1)
11     Creo, inicializo y lanzo los 50 hilos asociados a los clientes(id, r.nextInt(10000) + 1,
   r.nextInt(10000) + 1)
12     Esperar a que todos los hilos terminen
13     Calcular y mostrar por pantalla el tiempo medio de los clientes en checkout
14 }
15
16
17 Hilo cliente(id, X, Y, seleccion)
18 Variable Tiempo_Compra inicializado a X
19 Variable Tiempo_Caja inicializado a Y
20 Variable Tiempo_Espera_Cola inicializado a 0
21 Variable Caja_Seleccionada
22
23 Realizar compra en el tiempo Tiempo_Compra
24
25 Seleccionar una caja aleatoriamente y dirigirse a ella
26
27 // Esperar en la cola de la caja seleccionada
28 Iniciar cronometro Tiempo_Espera_Cola
```

```
29  wait(C[Caja_Seleccionada])
30  Parar cronometro Tiempo_Espera_Cola
31  Entrar a caja seleccionada
32  Pagar los productos a comprar en el tiempo Tiempo_Caja
33  signal(C[Caja_Seleccionada])
34
35  /* En este momento puede haber entrado otro cliente a la caja */
36
37  Meter la compra en bolsa // Tiempo no relacionado con la atencion de la caja
38  Salir de la caja
39  Abandonar el supermercado
40
41  // Comienzo Seccion Critica Pantalla
42  wait(P)
43  Mostrar tiempos por pantalla
44  // Final Seccion Critica Pantalla
45  signal(P)
46
47  Fin_Hilo
```

Apartado B

```
1   Recursos compartidos no compartibles
2   Pantalla
3   Buffer CajasLibres de 10 elementos enteros
4
5   main() {
6       Random r = new Random(1)
7       Creo un buffer CajasLibres de 10 elementos enteros (inicializado de 0 a 10)
8       Creo un Semaforo C general que gestiona el acceso a cajas (inicializado a 10)
9       Creo un Semaforo mutex asociado a "cajas libres" (inicializao a 1)
10      Creo Semaforo P (mutex) asociado a la pantalla (inicializado a 1)
11      Creo, inicializo y lanzo los 50 hilos asociados a los clientes(id, r.nextInt(10000) + 1,
    r.nextInt(10000) + 1)
12      Esperar a que todos los hilos terminen
13      Calcular y mostrar por pantalla el tiempo medio de los clientes en checkout
14  }
15
16
17  Hilo cliente(id, X, Y, seleccion)
18  Variable Tiempo_Compra inicializado a X
19  Variable Tiempo_Caja inicializado a Y
20  Variable Tiempo_Espera_Cola inicializado a 0
21  Variable Caja_Seleccionada no inicializada
22
23  Realizar compra en el tiempo Tiempo_Compra
24
25  // Esperar en la cola de la caja seleccionada
26  Iniciar cronometro Tiempo_Espera_Cola
27  wait(C)
28  Parar cronometro Tiempo_Espera_Cola
29
30  // Comienzo Seccion Critica de CajasLibres
31  wait(mutex)
32  Seleccionar caja libre del buffer CajasLibres y dirigirse a ella
```

```
33  // Final Seccion Critica de CajasLibres
34  signal(mutex)
35
36  Pagar los productos a comprar en el tiempo Tiempo_Caja
37  signal(C)
38
39  // Comienzo Seccion Critica de CajasLibres
40  wait(mutex)
41  Anadir la caja actual al buffer CajasLibres
42  // Final Seccion Critica de CajasLibres
43  signal(mutex)
44
45  /* En este momento puede haber entrado otro cliente a la caja */
46
47  Meter la compra en bolsa // Tiempo no relacionado con la atencion de la caja
48  Salir de la caja
49  Abandonar el supermercado
50
51  // Comienzo Seccion Critica de Pantalla
52  wait(P)
53  Mostrar tiempos por pantalla
54  // Final Seccion Critica de Pantalla
55  signal(P)
56
57  Fin_Hilo
```

2.3. Cuestiones planteadas

En este apartado he tenido dudas en si realizar o no una jerarquía de clases, pero al final he optado por hacerlo. Otra cuestion clave para la generación de resultados invariable ha sido sembrar la semilla de Random en el método main, facilitándome así la decisión de cual de las dos implementaciones es más eficaz.

2.4. Código

Client.java

```
1
2   package Exercise2;
3
4   public abstract class Client extends Thread {
5       protected long id;
6       protected int buyTime; // Time in milliseconds
7       protected int boxTime; // Time in milliseconds
8       protected int selectedBox;
9       protected long queueTimeStart;
10      protected long queueTimeEnd;
11
12      public Client(int id, int buyTime, int boxTime) {
13          this.id = id;
14          this.buyTime = buyTime;
```

```
15          this.boxTime = boxTime;
16          queueTimeStart = 0;
17          queueTimeEnd = 0;
18      }
19
20      public abstract void selectBox();
21
22      public void buy() {
23          try {
24              Thread.sleep(buyTime);
25          } catch (InterruptedException e) {
26              e.printStackTrace();
27          }
28      }
29
30      public abstract void waitInQueue();
31
32      public void payInBox() {
33          try {
34              // The client leaves the queue and enter to the box...
35              queueTimeEnd = System.currentTimeMillis();
36              Thread.sleep(boxTime);
37          } catch (InterruptedException e) {
38              e.printStackTrace();
39          }
40      }
41
42      public abstract void run();
43
44      public int getBoxTime() {
45          return boxTime;
46      }
47
48      public int getQueueTime() {
49          return (int) (queueTimeEnd - queueTimeStart);
50      }
51  }
```

ClientA.java

```
1   package Exercise2;
2
3   public class ClientA extends Client {
4       public ClientA(int id, int buyTime, int boxTime) {
5           super(id, buyTime, boxTime);
6       }
7
8       @Override
9       public void selectBox() {
10          selectedBox = (int) (Math.random() * SupermarketA.NUMBER_OF_BOXES);
11      }
12
13      @Override
14      public void waitInQueue() {
```

```
15          try {
16              // The client want to enter the box...
17              queueTimeStart = System.currentTimeMillis();
18              SupermarketA.boxSemaphores[selectedBox].acquire();
19          } catch (InterruptedException e) {
20          }
21      }
22
23      @Override
24      public void payInBox() {
25          super.payInBox();
26
27          // The client leaves the box...
28          SupermarketA.boxSemaphores[selectedBox].release();
29      }
30
31      @Override
32      public void run() {
33          // The client is buying...
34          buy();
35
36          // The client select a random box...
37          selectBox();
38
39          // The clients waits in his selected box queue
40          waitInQueue();
41
42          // The client is paying the bill in the respective box...
43          payInBox();
44
45          // Print the results on screen
46          Screen.print(String.format("%s\n%s\n%s\n%s\n\n", "Customer " + id + "
                served in box " + selectedBox + " -----",
47                  "Buy time: " + buyTime + " milliseconds", "Time in box: " +
                        boxTime + " milliseconds",
48                  "Waiting time in the box queue: " + getQueueTime() + "
                        milliseconds"));
49      }
50  }
```

ClientB.java

```
1   package Exercise2;
2
3   public class ClientB extends Client {
4       public ClientB(int id, int buyTime, int boxTime) {
5           super(id, buyTime, boxTime);
6       }
7
8       @Override
9       public void selectBox() {
10          selectedBox = SupermarketB.FreeBoxesManager.popFirsFreeBox();
11      }
12
```

```
13      @Override
14      public void waitInQueue() {
15          try {
16              // The client want to enter the box...
17              queueTimeStart = System.currentTimeMillis();
18              SupermarketB.freeBoxSemaphore.acquire();
19          } catch (InterruptedException e) {
20          }
21      }
22
23      @Override
24      public void payInBox() {
25          try {
26              // The client leaves the queue and enter to the selected box...
27              queueTimeEnd = System.currentTimeMillis();
28              Thread.sleep(boxTime);
29          } catch (InterruptedException e) {
30              e.printStackTrace();
31          }
32
33          // The client leaves the box...
34          SupermarketB.FreeBoxesManager.pushFreeBox(selectedBox);
35          SupermarketB.freeBoxSemaphore.release();
36      }
37
38      @Override
39      public void run() {
40          // The client is buying...
41          buy();
42
43          // The clients waits in the queue
44          waitInQueue();
45
46          // The client selects the free box
47          selectBox();
48
49          // The client is paying the bill in the respective box...
50          payInBox();
51
52          // Print the results on screen
53          Screen.print(String.format("%s\n%s\n%s\n%s\n\n", "Customer " + id + "
                served in box " + selectedBox + " -----",
54                  "Buy time: " + buyTime + " milliseconds", "Time in box: " +
                        boxTime + " milliseconds",
55                  "Waiting time in the box queue: " + getQueueTime() + "
                        milliseconds"));
56      }
57  }
```

Supermarket.java

```java
package Exercise2;

import java.util.Random;

public abstract class Supermarket {
    protected static final int NUMBER_OF_BOXES = 10;
    protected Client[] clients;

    // Random seed
    protected Random r;

    protected abstract void work();
}
```

SupermarketA.java

```java
package Exercise2;

import java.util.Random;
import java.util.concurrent.Semaphore;

public class SupermarketA extends Supermarket {
    public static Semaphore[] boxSemaphores = new Semaphore[NUMBER_OF_BOXES];

    public SupermarketA(Random r) {
        this.r = r;
    }

    @Override
    public void work() {
        // Creating NUMBER_OF_BOXES semaphores
        for (int i = 0; i < NUMBER_OF_BOXES; i++) {
            boxSemaphores[i] = new Semaphore(1, true);
        }

        // Creating and starting 50 Threads of Clients
        clients = new Client[50];
        for (int i = 0; i < clients.length; i++) {
            int buyTime = r.nextInt(10000) + 1; // From 1 to 1000 millisecs.
            int boxTime = r.nextInt(10000) + 1; // From 1 to 1000 millisecs.

            clients[i] = new ClientA(i, buyTime, boxTime);
            clients[i].start();
        }

        // Wait all threads to finish
        for (Client client : clients) {
            try {
                client.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
36              }
37          }
38
39          // Calculate the client average waiting time in box
40          int totalWaitingTime = 0;
41          for (Client client : clients) {
42              totalWaitingTime += client.getQueueTime() + client.getBoxTime();
43          }
44
45          System.out.println(String.format("%s\n%s\n\n", "Main thread -----",
46                  "Average waiting time at customers' checkout: " +
                        totalWaitingTime / clients.length + " milliseconds"));
47      }
48  }
```

SupermarketB.java

```
1   package Exercise2;
2
3   import java.util.LinkedList;
4   import java.util.Random;
5   import java.util.concurrent.Semaphore;
6
7   public class SupermarketB extends Supermarket {
8       public static Semaphore freeBoxSemaphore = new Semaphore(NUMBER_OF_BOXES);
9       private static LinkedList<Integer> freeBoxes = new LinkedList<>();
10
11      public SupermarketB(Random r) {
12          this.r = r;
13      }
14
15      // Class to manage the free boxes
16      static class FreeBoxesManager {
17          private final static Semaphore mutex = new Semaphore(1, true);
18
19          public static void pushFreeBox(int box) {
20              try {
21                  mutex.acquire();
22              } catch (InterruptedException e) {
23              }
24
25              freeBoxes.add(box);
26              mutex.release();
27          }
28
29          public static int popFirsFreeBox() {
30              try {
31                  mutex.acquire();
32              } catch (InterruptedException e) {
33              }
34
35              // Pop the first free box of the list
36              int freeBox = freeBoxes.removeFirst();
37              mutex.release();
```

```
38
39              return freeBox;
40          }
41      }
42
43      public void work() {
44          // Initializing the "free boxes" queue
45          for (int i = 0; i < NUMBER_OF_BOXES; i++) {
46              freeBoxes.add(i);
47          }
48
49          // Creating and starting 50 Threads of Clients
50          clients = new Client[50];
51          for (int i = 0; i < clients.length; i++) {
52              int buyTime = r.nextInt(10000) + 1; // From 1 to 1000 millisecs.
53              int boxTime = r.nextInt(10000) + 1; // From 1 to 1000 millisecs.
54
55              clients[i] = new ClientB(i, buyTime, boxTime);
56              clients[i].start();
57          }
58
59          // Wait all threads to finish
60          for (Client client : clients) {
61              try {
62                  client.join();
63              } catch (InterruptedException e) {
64                  e.printStackTrace();
65              }
66          }
67
68          // Calculate the client average waiting time in box
69          int totalWaitingTime = 0;
70          for (Client client : clients) {
71              totalWaitingTime += client.getQueueTime() + client.getBoxTime();
72          }
73
74          System.out.println(String.format("%s\n%s\n\n", "Main thread -----",
75                  "Average waiting time at customers' checkout: " +
                        totalWaitingTime / clients.length + " milliseconds"));
76      }
77  }
```

Screen.java

```
1   package Exercise2;
2
3   import java.util.concurrent.Semaphore;
4
5   public class Screen {
6       private static final Semaphore screenSemaphore = new Semaphore(1, true);
7
8       public static void print(String str) {
9           try {
10              screenSemaphore.acquire();
```

```
11          } catch (InterruptedException e) {}
12          System.out.println(str);
13          screenSemaphore.release();
14      }
15  }
```

OpenSupermarkets.java

```
1   package Exercise2;
2
3   import java.util.Random;
4
5   public class OpenSupermarkets {
6       /**
7        * Main method that runs when the program is started.
8        */
9       public static void main(String[] args) {
10          Random r;
11
12          // Supermarket A
13          System.out.println("---------- Supermarket A ----------");
14          r = new Random(1); // Initializating r with seed = 1
15          SupermarketA a = new SupermarketA(r);
16          a.work();
17
18          System.out.println();
19
20          // Supermarket B
21          System.out.println("---------- Supermarket B ----------");
22          r = new Random(1); // Re-initializating r with seed = 1
23          SupermarketB b = new SupermarketB(r);
24          b.work();
25      }
26  }
```

2.5. Preguntas

a) (0.5 puntos sobre 10) ¿Qué acciones pueden realizar simultáneamente los hilos?

Antes de empezar, me gustaría recalcar que algunas de estas acciones son redundantes para la resolución del problema, pero permiten adquirir una visión del mismo un poco más completa.

En el apartado A, las acciones que pueden realizar los clientes simultaneamente son:

1. Realizar una compra
2. Seleccionar una caja aleatoriamente y dirigirse a ella
3. Esperar en la cola de la caja seleccionada
4. Meter la compra en bolsa // Tiempo no relacionado con la atencion de la caja
5. Salir de la caja
6. Abandonar el supermercado

Mientras tanto, en el apartado B, las acciones que pueden realizar los clientes simultaneamente son:

1. Esperar en la única cola del supermercado
2. Seleccionar una caja libre y dirigirse a ella (Semáforo General)
3. Meter la compra en bolsa // Tiempo no relacionado con la atencion de la caja
4. Salir de la caja
5. Abandonar el supermercado

b) (0.5 puntos sobre 10) Explica el papel de los semáforos que has usado para resolver el problema.

Para el recurso no compartible Pantalla he usado un semáforo binario mutex que garantice el uso exclusivo de la misma por un único hilo. Además, he establecido su fairness a true, de tal forma que cualquier proceso que la solicite y no pueda usarla entre en una cola FIFO (First In First Out), o lo que es lo mismo: tendrá acceso al recurso cuando le toque su turno

En el apartado A he creado 10 semáforos del mismo tipo que el anterior, uno para cada caja, de tal forma que al acceso a las mismas por cada cliente sea exclusivo (un cliente por caja, pero pueden haber varios esperando en las colas)

Cabe destacar que la inicialización de estos semáforos a 1 simboliza que en el estado inicial de nuestro escenario estos recursos se encuentran desocupados.

En el apartado B, idem de lo mismo, hay un semaforo binario mutex para garantizar la exclusión mutua al recurso no compartible de Cajas Libres. En cambio, al haber una

única cola y estar todas las colas vacías en un principio, como en el apartado A, en este caso en puesto de tener un semáforo binario por cada caja, tenermos un semáforo general inicializado a 10, que se va decrementando conforme una caja es ocupada, e incrementando conforme es desocupada. Al igual que los otros semáforos, cuando su valor cambia a cero, el próximo proceso que intente usar el recurso que este mismo protege pasará a su cola de espera.

c) (0.5 puntos sobre 10) Realizada la medida del tiempo medio de espera en las colas de caja de los clientes en las situaciones a y b. ¿Qué planteamiento es más efectivo en base al tiempo?

Al generar los tiempos predefinidos de todos los hilos en el principal, las ejecuciones, con la misma semilla de las dos versiones, genrarán los mismos valores. Es por esto que si cambio la semilla un par de veces y comparo los resultados llego a la conclusión de que **la implementaciñon B es mas efectiva en base al tiempo**.

3. Ejercicio 3

3.1. Recursos no compartibles y condiciones de sincronización

Los recursos compartidos no compartibles, y que por tanto necesitan ser manejados en exclusión mutua son:

1. Pantalla

Además, existen 3 condiciones de sincronización:

1. Condición 1: El numero de operaciones de tipo 1 completadas no debe de ser mayor que el de operaciones de tipo 3 para poder ejecutar una operación de tipo 1.
2. Condición 2: El numero de operaciones de tipo 2 completadas no debe de ser mayor o igual que el de operaciones de tipo 1 para poder ejecutar una operación de tipo 2.
3. Condición 3: El numero de operaciones de tipo 3 completadas no debe de ser mayor o igual que el de operaciones de tipo 2 para poder ejecutar una operación de tipo 3.

3.2. Pseudocódigo

```
1  Recursos compartidos no compartibles
2  Pantalla
3
4  main() {
5      Crear Secuenciador Ternario ST
6      Crear 4 Hilos Operadores para cada operacion y lanzarlos(id, operacion, ST)
7  }
8
9
10 Monitor SecuenciadorTernario // Encargado de sincronizar las operaciones
11 Variable RentrantLock l
12 Variable nPrimeraOperacionCompletadas inicializada a 0
13 Variable nSegundaOperacionCompletadas inicializada a 0
14 Variable nTerceraOperacionCompletadas inicializada a 0
15 Variable Condicion primeraOperacion = l.newCondition()
16 Variable Condicion segundaOperacion = l.newCondition()
17 Variable Condicion terceraOperacion = l.newCondition()
18
19 procedimiento operacion1(id)
20 Adquirir Cerrojo l
21 Mientras nPrimeraOperacionCompletadas > nTerceraOperacionCompletadas
22 Hacer
23     primeraOperacion.await()
24 Fin_Mientras
25 Imprimir datos por Pantalla
26 Despertar hilos dormidos del conjunto wait de segundaOperacion
27 Desbloquear Cerrojo l
28
29 procedimiento operacion2(id)
30 Adquirir Cerrojo l
31 Mientras nSegundaOperacionCompletadas >= nPrimeraOperacionCompletadas
32 Hacer
33     segundaOperacion.await()
34 Fin_Mientras
35 Imprimir datos por Pantalla
36 Despertar hilos dormidos del conjunto wait de terceraOperacion
37 Desbloquear Cerrojo l
38
39 procedimiento operacion3(id)
40 Adquirir Cerrojo l
41 Mientras nTerceraOperacionCompletadas >= nSegundaOperacionCompletadas
42 Hacer
43     terceraOperacion.await()
44 Fin_Mientras
45 Imprimir datos por Pantalla
46 Despertar hilos dormidos del conjunto wait de primeraOperacion
47 Desbloquear Cerrojo l
48
49 Fin_Monitor
50
51
52 Hilo Operador(id, operacion, secuenciadorTernario)
53 Variable id inicializada a id
54 Variable operacion inicializada a operacion
55 Variable secuenciadorTernario inicializada a secuenciadorTernario
```

```
56
57  Repetir 3 veces
58      Dormir Hilo durante un tiempo aleatorio
59      Ejecutar operacion correspondiente desde secuenciadorTernario
60
61  Fin_Hilo
```

3.3. Cuestiones planteadas

El desarrollo de este ejercicio se puede resumir en las siguientes cuestiones: ¿Es una buena opción usar ReentrantLock junto Condition para implementar el Monitor? ¿Qué condiciones debería establecer?

3.4. Código

TernarySequencer.java

```
1   package Exercise3;
2
3   import java.util.concurrent.locks.Condition;
4   import java.util.concurrent.locks.ReentrantLock;
5
6   public class TernarySequencer {
7       private ReentrantLock l = new ReentrantLock();
8       private int nFirstOperationsCompleted;
9       private int nSecondOperationsCompleted;
10      private int nThirdOperationsCompleted;
11      private Condition firstOperation = l.newCondition();
12      private Condition secondOperation = l.newCondition();
13      private Condition thirdOperation = l.newCondition();
14
15      public TernarySequencer() {
16          nFirstOperationsCompleted = 0;
17          nSecondOperationsCompleted = 0;
18          nThirdOperationsCompleted = 0;
19      }
20
21      public void operation1(int id) throws InterruptedException {
22          l.lock();
23          try {
24              while (nFirstOperationsCompleted > nThirdOperationsCompleted)
25                  firstOperation.await();
26
27              nFirstOperationsCompleted++;
28              System.out.println(String.format("%s\n%s\n%s\n\n", "Thread " + id +
                    " -----", "Executes operation 1",
29                      "End of Thread " + id + " -----"));
30
31              secondOperation.signalAll();
32          } finally {
33              l.unlock();
34          }
```

```
35      }
36
37      public void operation2(int id) throws InterruptedException {
38          l.lock();
39          try {
40              while (nSecondOperationsCompleted >= nFirstOperationsCompleted)
41                  secondOperation.await();
42
43              nSecondOperationsCompleted++;
44              System.out.println(String.format("%s\n%s\n%s\n\n", "Thread " + id +
                    " -----", "Executes operation 2",
45                      "End of Thread " + id + " -----"));
46
47              thirdOperation.signalAll();
48          } finally {
49              l.unlock();
50          }
51      }
52
53      public void operation3(int id) throws InterruptedException {
54          l.lock();
55          try {
56              while (nThirdOperationsCompleted >= nSecondOperationsCompleted)
57                  thirdOperation.await();
58
59              nThirdOperationsCompleted++;
60              System.out.println(String.format("%s\n%s\n%s\n\n", "Thread " + id +
                    " -----", "Executes operation 3",
61                      "End of Thread " + id + " -----"));
62
63              firstOperation.signalAll();
64          } finally {
65              l.unlock();
66          }
67      }
68  }
```

Operator.java

```
1   package Exercise3;
2
3   public class Operator extends Thread {
4       private int id;
5       private int operation;
6       private TernarySequencer ternarySequencer;
7
8       public Operator(int id, int operation, TernarySequencer ternarySequencer) {
9           this.id = id;
10          this.operation = operation;
11          this.ternarySequencer = ternarySequencer;
12      }
13
14      public void run() {
15          for (int i = 0; i < 3; i++) {
```

```
16              try {
17                  Thread.sleep((int) Math.random() * 10);
18              } catch (InterruptedException e) {
19                  e.printStackTrace();
20              }
21
22              switch (operation) {
23              case 1:
24                  try {
25                      ternarySequencer.operation1(id);
26                  } catch (InterruptedException e) {
27                      e.printStackTrace();
28                  }
29                  break;
30              case 2:
31                  try {
32                      ternarySequencer.operation2(id);
33                  } catch (InterruptedException e) {
34                      e.printStackTrace();
35                  }
36                  break;
37              case 3:
38                  try {
39                      ternarySequencer.operation3(id);
40                  } catch (InterruptedException e) {
41                      e.printStackTrace();
42                  }
43                  break;
44              default:
45                  break;
46              }
47          }
48      }
49  }
```

TernarySequencerMonitor.java

```
1   package Exercise3;
2
3   public class TernarySequencerMonitor {
4       /**
5        * Main method that runs when the program is started.
6        */
7       public static void main(String[] args) {
8           TernarySequencer ternarySequencer = new TernarySequencer();
9
10          // Create 4 operators of each operation
11          for (int i = 0; i < 12; i++) {
12              Operator o = new Operator(i, (i %3 + 1), ternarySequencer);
13              o.start();
14          }
15      }
16  }
```

3.5. Preguntas

Cuestiones relativas al guión 3 de prácticas:

a) (0.5 puntos sobre 10) Respecto al código al que se refiere el Ejercicio 1, en lugar de la invocación a notifyAll(), ¿se podría haber usado una invocación a notify()? Justifica la respuesta

Si se puede usar un notify() porque el hilo que decrementa está en espera y será activado para decrementar cuando haya sido incrementada la variable, es decir, cuando se asegura que el decremento no será negativo. Como únicamente hay un proceso en espera, da igual hacer notify() o notifyAll(), pues con notify() pasamos al estado 'listo' a un proceso arbitrario del conjunto de espera, mientras que con notifyAll() todos los del conjunto de espera pasan a estar listos. Como solo hay uno esperando el resultado será el mismo.

b) (0.5 puntos sobre 10) Si añadimos a dicho código de forma correcta otro hilo que decrementa y otro que incrementa, en lugar de la invocación a notifyAll(), ¿se podría haber usado una invocación a notify()? Justifica la respuesta.

Se terminan ejecutando el mismo número de incrementos que de decrementos, por lo que al final siempre se desbloquearán todos los hilos de la cola de espera, asegurándose siempre que los decrementos no sean negativos

c) (0.5 puntos sobre 10) Respecto al código del Ejercicio 4, ¿se debe usar notify() o notifyAll()? Justifica la respuesta

Se pueden usar las dos, ya aunque hagamos notifyAll() y liberemos todos los procesos del conjunto de procesos en espera, como solo uno puede usar el monitor se terminaría produciendo el mismo efecto que con el notify().

d) (0.5 puntos sobre 10) Considerando el tiempo computacional, ¿Consideras que es más eficiente resolver el problema planteado en el Ejercicio 4 con el monitor synchronized o con el monitor reentrantlock? Justificar la respuesta.

ReentrantLock es más eficiente en cuanto a nivel de CPU, además de ser más flexible que el monitor synchronized. Un ReentrantLock, a diferencia de las construcciones sincronizadas, no necesita usar una estructura de bloque para bloquear e incluso puede mantener un bloqueo en todos los métodos.

e) (0.5 puntos sobre 10) ¿Qué tipo de monitor Java has usado?. Justifica la respuesta.

He usado el Monitor ReentrantLock con Condition porque, como ya he aclarado antes, es más eficiente y flexible.

f) (0.5 puntos sobre 10) En el monitor diseñado, ¿has usado notify/signal o notifyAll/signalAll? Justifica la respuesta.

He dado uso de signalAll. En realidad hubiese tenido el mismo efecto usar notify, pues a fin de cuentas, al estar sometidos a una condición, siempre ejecutara la operación operación respectiva un unico proceso de todos los 'notificados'.

g) (0.5 puntos sobre 10) ¿Cómo resuelves la exclusión mutua de la pantalla? Justifica tu respuesta.

Como las operaciones del monitor están vacías y son exclusivas, es decir en cada instante determinado solo se está ejecutando una única operación, he decidido que en el cuerpo de esas operaciones, entre el await y el signal, se realice la impresión por pantalla, en puesto de gestionarla con otros mecanismos a parte.

4. Ejercicio 4

4.1. Recursos no compartibles

Los recursos compartidos no compartibles, y que por tanto necesitan ser manejados en exclusión mutua son:

1. Pantalla

4.2. Pseudocódigo

```
1  Recursos compartidos no compartibles
2  Pantalla
3
4
5  main() {
6      Crear Buzon Solicitudes Ilimitado buzonControlador asociado al Controlador del Servidor(true)
7      Crear 20 Buzones buzonesClientes asociados a cada Cliente
8      Crear y Lanzar la Pantalla
9      Crear y Lanzar 20 Clientes jugadores(id, buzonesClientes[id], buzonControlador)
10     Crear y Lanzar el Controlador del Servidor(buzonControlador, buzonesClientes)
11 }
12
13
14 Hilo Cliente(id, buzonCliente, buzonControlador)
15 Variable id inicializada a id
16 Variable nivel inicializada a 0
17 Variable buzonCliente inicializada a buzonCliente
18 Variable buzonControlador inicializada a buzonControlador
19 Variable msg inicializada a cadena vacia
20
21 Repetir Siempre
22     Generar nivel del 1 al 10
23     Enviar solicitud de rival a buzonControlador
24     recibir(buzonCliente, msg)
25     Pantalla.mutex.send(msg)
26
27 Fin_Hilo
28
29
30
31 Hilo Controlador(buzonControlador, buzonesClientes)
32 Variable buzonControlador inicializada a buzonControlador
33 Variable buzonesClientes inicializada a buzonesClientes
34 Variable solicitudesPendientes (lista vacia inicialmente) // Solicitudes de emparejamiento pendientes
35 Variable msg inicializada a nulo // Las instancias de msg tendran la forma {id, nivel}
36
37 Repetir Siempre
38     recibir(buzonControlador, msg)
39
40     A adir msg a solicitudesPendientes
41
42     Si solicitudesPendientes.size() > 1
```

```
43          Para cada mensaje m en solicitudesPendientes Hacer
44              Si m != msg Y absolute(m.getLevel() - msg.getLevel()) <= 2
45                  Enviar mensaje informacion partida a buzonesClientes[mdg.id] y buzonesClientes[m.id]
46                  Borrar m y msg de solicitudesPendientes
47          Fin_Para
48
49  Fin_Hilo
50
51
52
53  Hilo Pantalla()
54  Variable mutex inicializada a nuevo buzon
55  Variable ocupada inicializada a false
56  Variable msg inicializada a cadena vacia
57
58  Repetir Siempre
59      Si no ocupada
60          recibir(msg, mutex)
61          ocupada = true
62          Dormir hilo un tiempo aleatorio
63          Imprimir msg
64          ocupada = false
```

4.3. Cuestiones planteadas

¿Deberia el controlador realizar la busqueda de rivales en con Streams y Lambdas de Java8? ->Posible optimizacion

¿Cuantos buzones son necesarios?

¿Enfoque centralizado?

4.4. Código

Screen.java

```
1  package Exercise4;
2
3  import java.util.Random;
4
5  import messagepassing.MailBox;
6  import messagepassing.Selector;
7
8  public class Screen extends Thread {
9      public static MailBox mutex = new MailBox();
10     private boolean busy;
11     private Selector selector;
12     private String msg;
13
14     public Screen() {
15         busy = false;
16         selector = new Selector();
17         selector.addSelectable(mutex, false);
18         msg = "";
19     }
20
21     @Override
22     public void run() {
23         Random r = new Random();
24         while (true) {
25             mutex.setGuardValue(true);
26             switch (selector.selectOrBlock()) {
27             case 1:
28                 if (!busy) {
29                     msg = (String) mutex.receive();
30                     busy = true;
31                     System.out.println(msg);
32                     try {
33                         Thread.sleep(r.nextInt(2) * 500);
34                     } catch (InterruptedException e) {
35                         e.printStackTrace();
36                     }
37                     busy = false;
38                 }
39                 break;
40             default:
41                 break;
42             }
43         }
44     }
45 }
```

Server.java

```
1  package Exercise4;
2
3  import java.io.Serializable;
4  import java.util.LinkedList;
5
6  import messagepassing.MailBox;
7  import messagepassing.Selector;
8
9  class Message implements Serializable {
10     private static final long serialVersionUID = 1L;
11     private int clientId;
12     private int level;
13
14     public Message(int clientId, int level) {
15         this.clientId = clientId;
16         this.level = level;
17     }
18
19     public int getClientId() {
20         return clientId;
21     }
22
23     public int getLevel() {
24         return level;
25     }
26 }
27
28 class Controller extends Thread {
29     private MailBox queries;
30     private MailBox[] clients;
31     private Selector selector;
32     private LinkedList<Message> pendingRequests;
33
34     public Controller(MailBox queries, MailBox[] clients) {
35         this.queries = queries;
36         this.clients = clients;
37         selector = new Selector();
38         selector.addSelectable(queries, false);
39         pendingRequests = new LinkedList<>();
40     }
41
42     public void run() {
43         while (true) {
44             queries.setGuardValue(true);
45             switch (selector.selectOrBlock()) {
46             case 1:
47                 Message msg = (Message) queries.receive();
48                 pendingRequests.add(msg);
49                 if (pendingRequests.size() > 1) {
50                     for (Message m : pendingRequests) {
51                         if (m != msg && Math.abs(m.getLevel() - msg.getLevel())
                               <= 2) {
52                             // Send the found partner to the clients
53                             clients[msg.getClientId()].send("Plays with player "
                                   + m.getClientId() + " ---- with LEVEL "
```

```
54                                      + m.getLevel() + " ----");
55
56                                  clients[m.getClientId()].send("Plays with player " +
                                        msg.getClientId() + " ---- with LEVEL "
57                                          + msg.getLevel() + " ----");
58
59                                  // Delete both messages from pendingRequests
60                                  pendingRequests.remove(pendingRequests.indexOf(msg));
61                                  pendingRequests.remove(pendingRequests.indexOf(m));
62                                  break;
63                              }
64                          }
65                      }
66                      break;
67                  default:
68                      break;
69                  }
70              }
71      }
72  }
73
74  class Client extends Thread {
75      private int id;
76      private int level;
77      private MailBox partner;
78      private MailBox controllerQueries;
79      private Selector selector;
80
81      public Client(int id, MailBox partner, MailBox controllerQueries) {
82          this.id = id;
83          level = 0;
84          this.partner = partner;
85          this.controllerQueries = controllerQueries;
86          selector = new Selector();
87          selector.addSelectable(partner, false);
88      }
89
90      private void setLevel() {
91          level = (int) (Math.random() * 10) + 1;
92      }
93
94      private void requestPartner(Message msg) {
95          controllerQueries.send(msg);
96      }
97
98      public void run() {
99          while (true) {
100             // Generate level
101             setLevel();
102
103             // Sending the querie message to the Server Controller
104             Message msg = new Message(id, level);
105
106             // Requesting a partner with similar level
107             requestPartner(msg);
108
109             partner.setGuardValue(true);
```

```
110             switch (selector.selectOrBlock()) {
111             case 1:
112                 String str = (String) partner.receive();
113                 Screen.mutex.send(String.format("%s\n%s\n\n"
114                         , "Player " + id + " ---- with LEVEL " + level
115                         , str
116                 ));
117                 break;
118             default:
119                 break;
120             }
121
122         }
123     }
124 }
125
126 public class Server {
127     private static final int NUMBER_OF_CLIENTS = 20;
128
129     /**
130      * Main method that runs when the program is started.
131      */
132     public static void main(String[] args) {
133         MailBox controllerQueries = new MailBox();
134         MailBox clients[] = new MailBox[NUMBER_OF_CLIENTS]; // Clients' mailboxes
135
136         // Create and launch the screen thread
137         Screen screen = new Screen();
138         screen.start();
139
140         // Create and launch NUMBER_OF_CLIENTS Client threads
141         for (int i = 0; i < NUMBER_OF_CLIENTS; i++) {
142             clients[i] = new MailBox(1);
143             Client c = new Client(i, clients[i], controllerQueries);
144             c.start();
145         }
146
147         // Create and launch the Server Controller
148         Controller controller = new Controller(controllerQueries, clients);
149         controller.start();
150
151     }
152 }
```

4.5. Preguntas

a) (0.5 puntos sobre 10) ¿Qué tipo de enfoque se está usando para resolver el problema (centralizado, distribuido o token en anillo lógico)? Justifica la respuesta.

El enfoque que se usa para resolver el problema es centralizado, pues disponemos de un controlador que hace de "Middleware.[en]tre otros entes software. Además, antes de

entrar a una sección crítica pedimos permiso al coordinador enviándole un mensaje de solicitud.

c) (0.5 puntos sobre 10) ¿Cómo has resuelto la exclusión mutua de la pantalla?

Para resolver la exclusión mutua de la pantalla he creado un ente que la representa, y el cual dispone de un buzon para asegurar la exclusión mutua. La pantalla únicamente recibe del buzón cuando esta está disponible.