Marquina Meseguer, Alfredo
Programación Concurrente y Distribuida
Grupo 2, Subgrupo 3
Junio 23/24
López Bernal, Sergio

PROYECTO PROGRAMACIÓN CONCURRENTE Y DISTRIBUIDA

ÍNDICE

1 Ejercicio 1

1.1 Recursos No Compartibles Y Condiciones de Sincronización

Los únicos recursos no compartibles son: el array que contiene todos los número y operaciones, llamado arrayLectura y el array que contiene los subtotales llamado arrayEscritura . El problema cuenta con tres tipos de procesos: generador, consumidor y sumador.

Los procesos de tipo generador y sumador tiene un solo proceso de cada tipo y requieren el acceso a un array completo, siendo estos arrayLectura y arrayEscritura, respectivamente.

El tipo de proceso consumidor tiene once ocurrencias. Este tipo lee del arrayLectura, calcula un subtotal y lo escribe en el arrayEscritura, aunque cada una accede a una parte de los arrays diferente, por lo que no tienen problemas con la exclusión mutua entre sí.

Sin embargo, por lo que hemos visto, dos tipos de proceso diferentes nos pueden estar ejecutandose al mismo tiempo, pues como un tipo quiere leer de un array y el otro escribir, se están rompiendo la condición de Bernstein y no se puede asegurar que estos se ejecuten correctamente, por lo que se tiene que aseguarar la exclusión mutua mediantel cerrojos.

De la misma manera, también existe un orden en el que los procesos deben ejecutarse. Primero debe ejecutarse el porceso generador, para rellenar el arrayLectura. Después, se tienen que ejecutar los procesos consumidores que leen el arrayLectura y llenan arrayEscritura con los resultados de las operacioens.Y, finalmente, el proceso sumador tiene que leer del arrayEscritura para calcular el resultado final.

En resumen, los recursos no compartibles son el arrayLectura, el arrayEscritura y la pantalla, cada uno siendo protegido por una serie de cerrojos que protegen secciones concretas, salvo por la pantalla que tiene un único cerrojo. Además, también se tiene que ejecutar en el orden generador, consumirdores y sumador.

1.2 Código

1.2.1 Pseudocódigo

```
// Variables goblales
arrayLectura: array[110] de entero
arrayEscritura: array[11] de entero
semaforoEscritura: semaphore
semaforoConsumidor: semaphore
```

```
semaforoSumador: array[10] de semaphore

process type generador():
begin
      // Calculo
      total = arrayEscritura[0];

      // El arrayLectura se divide en grupos de once (de ahí mod 11) y dentro de
esos
      // grupos, si se empieza a contar por 0, los que son pares son números y
los
      // impares operaciones (de ahí mod 2).
      for i = 0 .. len(arrayLectura)-1:
            if i%11%2 == 0 then // Comprobamos si es posición de entero
                  arrayLectura[i]   =   RandomInt(/*Rango   no   especificado   en
ejercicio*/);
            else  // Sino es de operación
                  arrayLectura[i] = RandomInt(min=1,max=3);
            finif
      finfor

      /* Versión alternativa, condición simple:
            for i=0; i<len(arrayLectura); i+=11 do
                  for j=0; j<11; j++ do
                        if j%2 == 0 then // Si par es posición de entero
                              arrayLectura[i+j]    =    RandomInt(/*Rango    no
especificado en ejercicio*/);
                        else  // Sino es de operación
                              arrayLectura[i+j] = RandomInt(min=1,max=3);
                        finif
                  finfor
            finfor
            */

      wait(semaforoEscritura); // No debería ser necesario porque es el primero
      writeln("Generador terminado.");
      signal(semaforoEscritura);

      signal(semaforoConsumidor);
end;

// También se puede calcular el inicio y final dentro del proceso,
// pero para ello tendríamos que asumir que la distancia entre inicio y final
// es de 11, de esta manera funciona para cualquier N donde el primer elemento
// es un número y la distancia enter inicio y final es impar.
// Si fuera par funcionaría pero la última operación se la compería
proccess type consumidor (id:entero, inicio:entero, final: entero)
begin
      total: entero
      operación: entero
      temp: entero
end;
begin
      // Espera a que el generador termine
      wait(semaforoConsumidor);
      // Cuando se despierta, levanta al siguiente
      signal(semaforoConsumidor);
      // Inicialización variables
      operación = 0;
```

```
      // Guardamos el valor del primer elemento pues se añade
      // al total de forma diferente al resto.
      total = arrayLectura[inicio];

      // Recorremos solo los números (posiciones impares en el grupo de 11 si se
empieza
      // por 1, y pare si se empieza por 0), y consideramos el espacio anterior
de
      // operación.
      i = inicio+2;
      while i<=final then
            temp = arrayLectura[i] // guardamos en local por buenas prácticas
            operación = arrayLectura[i-1]
            total = calcular(total, temp, operación)
            i+=2
      finwhile

      arrayEscritura[id] = total; // Guardar resultado en global
      // La sección crítica solo requiere el uso de la salida
      wait(semaforoEscritura);
      writeln(id,total); // Imprimir el mensaje
      signal(semaforoEscritura);
      signal(semaforoSumador[id]);
end;


/*    Función que realiza una operación sobre los números num1 y num2
      según el valor de op
      Si el valor de op es:
            1 - Suma
            2 - resta
            3 - Multiplicación
            otro - error
*/
entero function calcular(num1: entero, num2: entero, op: entero)
begin
      total:entero;
end;
begin
      switch temp then
            case 1:
                  total = num1 + num2;
            case 2:
                  total = num1 - num2;
            case 3:
                  total = num1 * num2;
            default:
                  // Lanzar error
      finswitch
      return total
end;

/*    Caculamos la suma de todos los valores guardados en
      el array de escritura.
*/

process type sumador():
begin
```

```
      total:entero // Esta variable puede ser global si queremos utilizar el
total en el main
end
begin
      // Cada consumidor hace un signal, cuando todos hallan terminado,
      // entonces podemos sumar.
      for i:=0.. len(semaforoSumador) do:
            wait(semaforoSumador[i]);
      // Calculo
      total = arrayEscritura[0];
      for i = 1 .. len(arrayEscritura)-1:
            total += arrayEscritura[i];

      wait(semaforoEscritura); // No debería ser necesario porque es el último
      writeln("Sumador terminado. Total:", total);
      signal(semaforoEscritura);

      // OPTIONAL: Se pueden "liberar" los semáforos
      for i:=0.. len(semaforoSumador) do:
            signal(semaforoSumador[i]);
end;



main()
begin
      procesos: array[1..11] de process; // De Process consumidor
      solución: entero;
end
begin


      writeln("Comienzo de ejecución");

      // Inicialización de semáforos
      initial(semaforoEscritura, 1)
      initial(semaforoConsumidor, 0)
      for i =0..len(semaforoSumador) do
            initial(semaforoSumador[i],0)

      // Definimos consumidores,
      for i = 0 .. len(arrayEscritura)-1 do
            procesos[i] = consumidor(i, i*11, (i+1)*11-1)

      // lanzar procesos
      cobegin
      generador; // No inicializo porque no argumentos
      for i = 0 .. len(arrayEscritura)-1 do
            procesos[i];
      sumador(); // No inicializo porque no argumentos
      coend;

      writeln ("Procesos terminados");
end;
```

1.2.2 Código Java

Generador

```
package ejercicio1;

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

public class Generador extends Thread {

   private ReentrantLock l;
   private Condition c;

   public void run() {
      Ejercicio1.cerrojo.lock();
      try {
         for (int i = 0; i < 110; i++) {
            if (i % 11 % 2 == 1)
               Ejercicio1.arrayLectura[i] = (int) (Math.random() * (3)) + 1;
            else
               Ejercicio1.arrayLectura[i] = (int) (Math.random() * (5)) + 1;
         }
         Ejercicio1.turno++;
      } finally {
         Ejercicio1.cerrojo.unlock();
      }
   }
}
```

Consumidor

```
package ejercicio1;

/**
 * Procesos que lee una porción del array de lectura y calcula un subtotal que acaba colocando en el array de escritura
 */
public class Consumidor extends Thread {
         private final int processId;
         private final int inicioArrayLectura;
         private final int finArrayLectura;

         /**
          * Constructor con los datos exclusivos del proceso concreto
          * @param processId el id del proceso
          * @param inicioArrayLectura el inicio del trozo que puede acceder
```

```
	 * @param finArrayLectura el fin del trozo que puede acceder
	 */
	public Consumidor(int processId, int inicioArrayLectura, int finArrayLectura) {
		super();
		this.processId = processId;
		this.inicioArrayLectura = inicioArrayLectura;
		this.finArrayLectura = finArrayLectura;
	}

	@Override
	public void run() {

		Ejercicio1.cerrojo.lock();
	try {
			while(Ejercicio1.turno != 1){
				Ejercicio1.cerrojo.unlock();
				Ejercicio1.cerrojo.lock();
			}
	} finally {
			Ejercicio1.cerrojo.unlock();
	}
	int operacion;
		int temp;
		int total = Ejercicio1.arrayLectura[inicioArrayLectura];

		for (int i = inicioArrayLectura + 2; i <= finArrayLectura; i+=2) {
			temp = Ejercicio1.arrayLectura[i];
			operacion= Ejercicio1.arrayLectura[i-1];
			total = calcular(total, temp, operacion);
		}
		Ejercicio1.arrayEscritura[processId] = total;

		Ejercicio1.cerrojo.lock();
		Ejercicio1.consumidoresEjecutando--;
		if (Ejercicio1.consumidoresEjecutando == 0)
			Ejercicio1.turno = 2;
		Ejercicio1.cerrojo.unlock();

		Ejercicio1.lockPrint.lock();
		System.out.println("***************\nProceso terminado: " + processId + "\nDa como
resultado: " + total
				+ "\n***************");
		Ejercicio1.lockPrint.unlock();

	}

	/**
	 * Método auxiliar que calcula una opearación entre dos número
	 * @param num1 primer número de la operación
```

```
         * @param num2 segundo número de la operación
         * @param operacion operación a realizar
         * @return el resultado
         */
        private int calcular(int num1, int num2, int operacion) {
                int total;

                try {
                        switch (operacion) {
                        case 1:
                                total = num1 + num2;
                                break;
                        case 2:
                                total = num1 - num2;
                                break;

                        case 3:
                                total = num1 * num2;
                                break;
                        default:
                                throw new IllegalArgumentException("Error calculando la operacion " +
num1 + " " + num2 + " "
                                                + operacion + " En el proceso " + processId);
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                        total = 0;
                }
                return total;
        }
}
```

SUMADOR

```
package ejercicio1;

import java.util.Arrays;

public class Sumador extends Thread {

   public void run() {
      super.run();
      int resultado = 0;
      Ejercicio1.cerrojo.lock();
      try {
         while(Ejercicio1.turno != 2){
            Ejercicio1.cerrojo.unlock();
            Ejercicio1.cerrojo.lock();
```

```
            }

            resultado = totalArrayEscritura(Ejercicio1.arrayEscritura);
        } finally {
            Ejercicio1.cerrojo.unlock();
        }

        Ejercicio1.lockPrint.lock();
        try {
            System.out.println("El total es: " + resultado + ".");
        } finally {
            Ejercicio1.lockPrint.unlock();
        }
    }

    public static int totalArrayEscritura(Integer... arrayEscritura) {
        return Arrays.stream(arrayEscritura).reduce(0, Integer::sum);
    }
}
```

Main

```
package ejercicio1;

import java.util.concurrent.locks.ReentrantLock;

/**
 * Clase principal que contine el main y las variables compartidas
 */
public class Ejercicio1 {
        public static ReentrantLock lockPrint = new ReentrantLock();
        public static ReentrantLock cerrojo = new ReentrantLock();

        public static Integer[] arrayLectura = new Integer[110];
        public static Integer[] arrayEscritura = new Integer[10];
        public static int consumidoresEjecutando = 10;
        // Como con Reentrant lock no se puede crear un semáforo nulo, que sería lo necesario para
sincronizar los procesos
        // lo sustituimos por una variable turno que se va actualizando conforme terminan los turnos.
        public static int turno = 0;

        /**
         * El problema
         * @param args lo que quieras, no se usa
         */
        public static void main(String[] args) {
                Consumidor[] procesos = new Consumidor[10];
                Generador generador = new Generador();
```

```
                Sumador sumador = new Sumador();

                System.out.println("Comienzo");

                System.out.println("Creando procesos");
                for (int i = 0; i < 10; i++) {
                        procesos[i] = new Consumidor(i, i * 11, (i + 1) * 11 - 1);
                }

                System.out.println("Ejecutando procesos");
                sumador.start();
                for(Consumidor p: procesos) p.start();
                generador.start();

                try {
                        generador.join();
                        for (Consumidor p: procesos) p.join();
                        sumador.join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                System.out.println("Fin de ejecución de procesos.");

                System.out.println("Array Lectura:[");
                for (Integer i : arrayLectura)
                        System.out.print(i + ", ");
                System.out.print("].\nArray Escritura:[");
                for (Integer i : arrayEscritura)
                        System.out.print(i + ",");
                System.out.println("].");
        }
}
```

1.3 CUESTIONES

1.3.1 APARTADO A

¿Qué acciones pueden realizar los hilos concurrentemente? Justifica la respuesta.

Los diferentes procesos pueden realizar las acciones concurrentemente siempre que no estén utilizando un recurso global no compartible. Estos recursos pueden ser tanto software (una variable) como hardware (pantalla), los problemas suelen surgir cuando los procesos escriben en ellos. En este caso, en cada proceso podemos identificar tres recursos que utilizan todos los procesos, el array de lectura, el array de escritura y la pantalla.

El array de lectura (si lo consideramos un solo recurso) cumple la condición de Bernstein ya que ninguno de los procesos, salvo el método main donde no hay concurrencia, edita dicho array, por lo que no hay que preocuparse de su uso.

El array de escritura en un principio parece no cumplir dicha condición pues todos los procesos escriben en él. Sin embargo, si miramos fijamente al comportamiento de los hilos, estos tiene una sola posición donde pueden escribir, sin compartirla con ningún otro proceso. Esto provoca que este recurso no sea, de hecho, compartido, por lo que no hace falta defenderlo. De la misma manera se podría analizar el array de lectura, sin embargo, como nunca se llega a escribir en este no es necesario.

Finalmente, tenemos la pantalla, que es accedida mediante la llamada a sistema `System.out.println`. Este sí que se trata de un recurso no compartible, pues si bien cada proceso es diferente, todos comparten la misma consola de salida estándar.

Con esto llegamos a la conclusión de que todas las acciones se pueden realizar concurrentemente salvo las impresiones por pantalla que tienen que ser protegidas.

1.3.2 Apartado B

Las impresiones que hacen los hilos, ¿son consecutivas o están desordenadas con las de los demás hilos? ¿Cuál de las opciones consideras que es la correcta? Justifica la respuesta.

Se escriben de forma ordenada según el tipo: primero generador, luego consumidores y finalmente sumador, pero se realizan de forma desordenada entre los consumidores. Las impresiones realizadas por los diferentes hilos se realizan de forma desordenada según cada proceso llegue a la sección crítica. Esto es porque los procesos se ejecutan en un orden parcial, es decir, que los procesos se pueden ejecutar en un orden diferente al llamado. Sin embargo, es probable que la primera llamada sí que sigan dicho orden y se vallan cambiando mientras las instrucciones de los procesos se vayan entremezclando.

En este ejercicio, parece que los procesos se están ejecutando de forma síncrona, en un orden total, porque el primer hilos en terminar es el primero y sigue el orden hasta el final. No obstante, es probable que esto se deba a que la cantidad de código a ejecutar es muy pequeña y es igual de larga en todos ellos. Es posible, que para poder observar la diferencia sea necesario un problema de mayor magnitud y/o que la ejecución varíe según la entrada. Si se quisiera observar en este problema, se podría pausar la función del proceso una cantidad aleatoria de segundos (por ejemplo, entre 0 y 3).

1.3.3 Apartado C

Si no usaras ningún mecanismo para sincronización, ¿cómo podría ser la salida en pantalla del programa anterior?

Sería desordenada y estaría mal, pues como los programas accederían a partes del array que todavía no han sido rellenadas, los número utilizados para los cálculos serían los incorrectos.

2 Ejercicio 2

2.1 Recursos No Compartibles Y Condiciones de Sincronización

En este problema tenemos una serie de transeúntes (coches y peatones) que quieren pasar por un cruce. El cruce tiene tres direcciones, dos para los coches (Norte-Sur y Este-Oeste) y una para los peatones, y solo se puede utilizar una a la vez, por lo que se tiene que compartir es el cruce en sí. La manera en la que se controlaría en la vida real sería una semáforo, este semáforo controlaría el turno de cada dirección.

De la misma manera, podríamos crear tres tipos de procesos, uno coche, otro peatones y finalmente un semáforo que controle los turnos. Si seguimos inspirándonos en la vida real, podríamos pensar en los semáforos que tiene cada transeúnte, que le dicen si puede o no pasar, y crear una variable booleana por dirección y que cada proceso comprobase si puede pasar según su valor. Si bien esta solución podría ser viable, tener que manejar tres variables puede resultar complicado, y como sabemos que estas solo representan un turno, podemos crear una variable entera turno y que cada proceso comprueba si es su turno para poder pasar. Esta variable también podría indicar el final de la ejecución si le pone un valor como -1. Turno también podría ser una variable de enumerado, aunque esto no se ha implementado.

Además, en caso de que un proceso no pudiese pasar directamente (porque no es su turno o el cruce se encuentra ocupado), tenemos que tener una manera de saber cuantos procesos se encuentran esperando y despertarlos. Para solucionar este problema podemos inspirarnos en la solución del problema clásico de lectores y escritores. En esta solución tenemos una variable compartida para contar el número de procesos de un tipo esperando y un semáforo binario que los despierta cuando es su momento de ser ejecutados. Esto es similar a las variables booleanas propuestas anteriormente. Para mayor comodidad y evitar código repetido en coche, que tiene dos direcciones, se ha representados los semáforos binarios de nuestro problema en un array. De esta manera, cada proceso tiene que despertar a su compañero y cuando se termina su turno, despiertan al siguiente.

También tenemos que tener en cuenta, que no todos los transeúntes de una dirección pueden pasar al mismo tiempo, pues el tamaño del cruce es limitado. Por lo que se tiene que seguir el número de transeúntes pasando y limitar de acuerdo al máximo. Es decir, si un transeúnte quiere pasar, normalmente despertaría a otro proceso antes de hacerlo,

pero no lo debería en caso de que el número de transeúntes en su misma dirección pasando esté al máximo. De la misma manera, cuando un transeúnte termine de pasar, deberá despertar a otro de su misma dirección si sigue siendo su turno o en caso de que no lo sea y él sea el último transeúnte pasando, despertar al del siguiente turno, pues el cruce solo se puede pasar una dirección a la vez. Si no es su turno, pero no es el último, no hará nada.

En conclusión, los recursos a compartir van a ser el turno, un array con el número de procesos esperando y un array con el número de procesos cruzando (un elemento en cada array por dirección), la exclusión mutua de estos elementos se asegurará con el uso de un semáforo binario “mutex”. La sincronización se va a asegurar mediante un array de semáforos binarios (al igual que con los arrays anteriores un elemento por dirección), donde los procesos esperarán en el semáforo correspondiente a su turno, siendo despertados por el proceso anterior y despertando al siguiente turno cuando el suyo acabe.

Como en la implementación se ha impreso por pantalla, esta también es un recurso no compartido y también está gestionado por el semáforo mutex.

2.2 CÓDIGO

2.2.1 PSEUDOCÓDIGO

```
// La variable turno informa a qué grupo de le toca pasar.
// Siendo 0 calle norte-sur, 1 calle este-oeste y 3 peatones.
// Cuando turno se pone a -1 significa que la simulación ha terminado.
// NOTA: Turno podría ser enum.
turno:int;
mutex:semaphore;
semaforosTurnos: array[0..2] semaphore;
esperando, pasando:array[0..2] of int;

// Este proceso se encarga de actualizar la variable turno.
// Simula los semáforos de la carretera cambiando
process type semaforoCarretera()
begin
      // Estas líneas se hacen en el constructor en Java para que sea menos
lioso
      wait(mutex);
      turno:= 0;
      signal(mutex);
      // Aquí se pone un número arbitrario de turnos que queremos que dure el
programa
      for i:=0 to 100 do
      begin
            wait(mutex);
            turno := (turno+1) mod 3;
            signal(mutex);
```

```
      // La función de dormir de Java pero parámetro en segundos
            sleep(5);
      end;
      wait(mutex);
      turno := -1;
      for i:= 0 .. 2 do // Despertamos a todos los procesos esperando para que
salgan
            signal(semaforosTurnos[i])
      signal(mutex);
end;

process type coche(int calleInicial)
var
      turnoPasar:int;
begin
      turnoPasar:= calleInicial
      wait(mutex);
      while(turno != -1)
      begin
            /* Las condiciones son que no puede pasar si:
            1. no es su turno
            2. hay gente pasando del anterior
            3. hay demasiados pasando en este turno
            */
            if turno != turnoPasar || pasando[(turnoPasar-1) mod 3] !=  0 ||
pasando[turnoPasar] >= MAX_PASAR_COCHES then
            begin
                  esperando[turnoPasar]++;
                  signal(mutex);
                  wait(semaforosTurnos[turnoPasar]);
                  esperando[turnoPasar]--;
            end;
            if turno == -1: // Si el turno es el final salir
                  break;

            pasando[turnoPasar]++;

            // Liberamos si el cuace no está lleno
            if esperando[turnoPasar] != 0 and pasando[turnoPasar] <
MAX_PASAR_COCHES then
                  signal(semaforosTurnos[turnoPasar]); // Cesión exclusión mutua
al siguiente
            else // Si está lleno liberamos el mutex
                  signal(mutex);

            // pasar
            sleep(0.5);

            wait(mutex);
            pasando[turnoPasar]--;

            // Liberamos del mismo proceso si (todas ciertas):
```

```
            // 1. es nuetro turno
            // 2. hay otros procesos esperando
            if turno == turnoPasar && esperando[turnoPasar] != 0 then
                  signal(semaforosTurnos[turnoPasar]);
            // Liberamos del siguiente si (todas ciertas);
            // 1. no hay ninguno de nuestro turno pasando
            // 2. no es nuestro turno
            // 3. hay procesos del siguiente turno esperando
            else if pasando[turnoPasar] == 0 and turnoPasar != turno and
esperando[turnoPasar+1] > 0 then
                  signal(semaforoTurnos[turnoPasar + 1]);
            else // Si ninguna de las anteriores entonces liberamos mutex
                  signal(mutex);

            sleep(7);
            turnoPasar := (turnoPasar+1) mod 2;
            wait(mutex);
      end;
      signal(mutex);
end;


TURNO_PEATONES = 2: int;
SIGUIENTE_TURNO = 0: int;
ANTERIOR_TURNO = 1: int;
process type peaton()
begin
      wait(mutex);
      while(turno != -1)
      begin
            if turno != TURNO_PEATONES || pasando[ANTERIOR_TURNO] != 0 then
            begin
                  esperando[TURNO_PEATONES]++;
                  signal(mutex);
                  wait(semaforosTurnos[TURNO_PEATONES]);
                  esperando[TURNO_PEATONES]--;
            end;
            pasando[TURNO_PEATONES]++;
            if esperando[TURNO_PEATONES] > 0  and pasando[turnoPasar] <
MAX_PASAR_PEATONES then
                  signal(semaforosTurnos[TURNO_PEATONES]);
            else
                  signal(mutex);

            // pasar
            sleep(3);

            wait(mutex);
            pasando[TURNO_PEATONES]--;

            if turno == TURNO_PEATONES then
                  signal(semaforosTurnos[TURNO_PEATONES]);
```

```
            else if pasando[TURNO_PEATONES] = 0 and turno != TURNO_PEATONES and
esperando[turno] > 0 then
                  signal(semaforoTurnos[turno]);
            else
                  signal(mutex);

            sleep(8);
            wait(mutex);
      end;
      signal(mutex);
end;

main()
begin
      procesosCoches: array[0..100] de process; // De Process consumidor
      procesosPeatones: array[0..50] de process; // De Process consumidor
      solución: entero;
end
begin


      writeln("Comienzo de ejecución");

      // Inicialización de semáforos
      initial(semaforoEscritura, 1)
      initial(semaforoConsumidor, 0)
      for i =0..len(semaforoSumador) do
            initial(semaforoSumador[i],0)

      // Definimos consumidores,
      for i = 0 .. len(arrayEscritura)-1 do
            procesos[i] = consumidor(i, i*11, (i+1)*11-1)

      // lanzar procesos
      cobegin
      generador().launch(); // No inicializo porque no argumentos
      for i = 0 .. len(arrayEscritura)-1 do
            procesos[i].launch();
      sumador().launch(); // No inicializo porque no argumentos
      coend;

      writeln ("Procesos terminados");
end;
```

2.2.2 CÓDIGO JAVA

SEMÁFORO CARRETERA

```
package ejercicio2;

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

```java
/**
 * La Clase que se encarga principalmente de cambiar de turno, también contiene todas las variables compartidas
 */
public class SemaforoCarretera extends Thread {
    public static final int NUM_SEMAFOROS = 3;
    private static final int NUM_TURNOS = 3;
    private Semaphore mutex;
    private int turno;
    // Semáforos binarios que se activan cuando es el turno del indice correspondiente
    private Semaphore[] semaforosTurnos;
    // Número de procesos esperando que pertenecen al turno del indice correspondiente
    private int[] esperando;
    // Número de procesos pasando que pertenecen la turno del indice correspondiente
    private int[] pasando;

    /**
     * Constructor de la Clase, como todas las variables compartidas empiezan siempre en el mismo valor, no es necesario
     * tener ningún parámetro.
     */
    public SemaforoCarretera() {
        super();
        this.mutex = new Semaphore(1);
        this.turno = 0;
        this.semaforosTurnos = new Semaphore[NUM_SEMAFOROS];
        this.esperando = new int[NUM_SEMAFOROS];
        this.pasando = new int[NUM_SEMAFOROS];

        for (int i = 0; i < NUM_SEMAFOROS; i++) {
            this.semaforosTurnos[i] = new Semaphore(0);
            this.esperando[i] = 0;
            this.pasando[i] = 0;
        }
    }

    /**
     * Obtener la variable compartida mutex
     *
     * @return el mutex
     */
    public Semaphore getMutex() {
        return mutex;
    }

    /**
     * Obtener la variable compartida turno
     *
```

```
     * @return el turno
     */
    public int getTurno() {
        return turno;
    }

    /**
     * Obtener el array de semáforos semaforosTurno
     *
     * @return semaforosTurnos
     */
    public Semaphore[] getSemaforosTurnos() {
        return semaforosTurnos;
    }

    /**
     * Obtener el array de transeúntes esperando
     *
     * @return esperando
     */
    public int[] getEsperando() {
        return esperando;
    }

    /**
     * Obtener el array de transeuntes pasando
     *
     * @return
     */
    public int[] getPasando() {
        return pasando;
    }

    /**
     * Código que se ejecuta al lanzar el proceso, su función es cambiar el turno.
     */
    @Override
    public void run() {
        // Se espar un tiempo inicial para
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (int i = 0; i < NUM_TURNOS; i++) {
            try {
                this.mutex.acquire(); // Se acquiere la exclusión mutua para editar la variable compartida
turno
```

```
            // Se pasa al siguiente turno
            this.turno = (this.turno + 1) % 3;
            System.out.println("~Cambio de Turno nº " + (i + 1) + " turno actual " + this.turno);

            this.mutex.release(); // Libera exclusión mutua
            Thread.sleep(5000); // Espera a que termine el turno
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    // Terminamos  el programa
    try {
        this.mutex.acquire();
        this.turno = -1; // Se cambia el turno a -1 para indicar el fin
        System.out.println("~Cambio de Turno a Fin " + this.turno);
        for (int i = 0; i < 3; i++) {
            if (semaforosTurnos[i].hasQueuedThreads()) {
                System.out.println("/ Liberando Parados " + i);
                // Se liberan todos los procesos esperando. Estos comprueban si se ha terminado el
programa antes
                // de realizar cualqueir acción. Así que no debe haber ningún problema.
                semaforosTurnos[i].release(semaforosTurnos[i].getQueueLength());
            }
        }
        this.mutex.release();
    } catch (Exception ex){
        ex.printStackTrace();
    }
  }
}
```

COCHE

```
package ejercicio2;

/**
 * Clase que representa al proceso Coche que para por el cruce
 */
public class Coche extends Thread {
   private static final int DIRECCIONES = 2;
   private static final int TIEMPO_ESPERA = 7000;
   private static final int TIEMPO_CRUCE = 500;
   public final static int MAX_PASAR_COCHES = 4;

   private SemaforoCarretera s;
   private int calleInicial;
   private int id;

   /**
```

```java
    * Contructor del proceso
    *
    * @param semaforo     referencia al semaforo que cambiar turno y contiene las variables compartidas
    * @param calleInicial la dirección que debe cruzar en un principio
    * @param id           el id del proceso @class Coche
    */
   public Coche(SemaforoCarretera semaforo, int calleInicial, int id) {
      super();
      this.s = semaforo;
      this.calleInicial = calleInicial;
      this.id = id;
   }

   /**
    * Método ejecutado por el proceso al ser lanzado
    */
   @Override
   public void run() {
      /* NOTE: por motivos de tiempo, se me olvido que esto no estaba comentado, no se ha podido
comentar
      correctamente. Por favor, lea la @class Peaton que está comentada y es muy similar
       */
      int turnoPasar = this.calleInicial;

      try {
         s.getMutex().acquire();
      } catch (InterruptedException e) {
         // TODO Auto-generated catch block
         e.printStackTrace();
      }
      System.out.println("Comienzo ejecución coche:" + id);
      while (s.getTurno() != -1) {
         if (s.getTurno() != turnoPasar || s.getPasando()[turnoPasar] >= MAX_PASAR_COCHES
               || s.getPasando()[turnoAnterior(turnoPasar)] != 0) {
            s.getEsperando()[turnoPasar]++;
            s.getMutex().release();
            try {
               s.getSemaforosTurnos()[turnoPasar].acquire();
            } catch (InterruptedException e) {
               // TODO Auto-generated catch block
               e.printStackTrace();
            }

            s.getEsperando()[turnoPasar]--;
         }

         if (s.getTurno() == -1) {
            System.out.println("=Saliendo coche " + id);
            break;
```

```
        }

        s.getPasando()[turnoPasar]++;

        System.out.println(
                "*Pasando Coche " + id + " en dirección " + turnoPasar + " total " + s.getPasando()
[turnoPasar]);
        if (s.getEsperando()[turnoPasar] != 0 && s.getPasando()[turnoPasar] < MAX_PASAR_COCHES) {
            System.out.println("#Coche " + id + " hace un realease." + s.getPasando()[turnoPasar]);
            s.getSemaforosTurnos()[turnoPasar].release();
        } else {
            System.out.println("%Coche " + id + " no hace nada." + s.getPasando()[turnoPasar]);
            s.getMutex().release();
        }
        // Pasar
        try {
            Thread.sleep(TIEMPO_CRUCE);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        // Fin Pasar
        try {
            s.getMutex().acquire();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("-Ha Pasado Coche " + id + " en dirección " + turnoPasar);
        s.getPasando()[turnoPasar]--;
        if (s.getTurno() == turnoPasar && s.getEsperando()[turnoPasar] != 0) {
            System.out.println("##Coche " + id + " libera tras pasar . " + s.getPasando()[turnoPasar] + " "
                    + s.getSemaforosTurnos()[turnoPasar].getQueueLength());
            s.getSemaforosTurnos()[turnoPasar].release();
        } else if (s.getPasando()[turnoPasar] == 0 && turnoPasar != s.getTurno()
                && s.getEsperando()[turnoPasar + 1] > 0) {
            System.out.println("$Coche " + id + " libera siguiente Turno. " + s.getPasando()[turnoPasar]);
            s.getSemaforosTurnos()[turnoPasar + 1].release();
        } else {
            System.out.println("%Coche " + id + " no hace nada");
            s.getMutex().release();
        }

        try {
            Thread.sleep(TIEMPO_ESPERA);
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
```

```
            // Cambiar de dirección
            turnoPasar = (turnoPasar + 1) % DIRECCIONES;
            try {
               s.getMutex().acquire();
            } catch (InterruptedException e) {
               // TODO Auto-generated catch block
               e.printStackTrace();
            }
//                              System.out.println("Coche"+id +" vuelve a intentar cruzar en dirección
"+turnoPasar);
         }
         System.out.println("FIN coche " + id + " " + s.getMutex().hasQueuedThreads());
         s.getMutex().release();
      }

      /**
       * Método auxiliar que calcula en turno anterior al actual
       * @param turno turno actual
       * @return turno anterior
       */
      private int turnoAnterior(int turno) {
         int anterior = (turno - 1) % 3;
         return (anterior == -1) ? 2 : anterior;
      }

}
```

PEATÓN

```
package ejercicio2;

/**
 * Clase que representa a un peatón intentado pasar por el paso de cebra.
 */
public class Peaton extends Thread {
   public final static int TURNO_PEATONES = 2, ANTERIOR_TURNO = 1;
   private static final long TIEMPO_CRUCE = 3000;
   private static final long TIEMPO_ESPERA = 8000;
   private static final int MAX_PASAR_PEATONES = 10;
   private SemaforoCarretera s;
   private int id;

   /**
    * Contructor de la clase peaton
    *
    * @param s  la clase que contiene las vairbles compartidas
    * @param id el id del proceso
    */
   public Peaton(SemaforoCarretera s, int id) {
```

```
        super();
        this.s = s;
        this.id = id;
    }

    /**
     * El método principal donde se ejecuta el código del peatón
     */
    @Override
    public void run() {

        try {
            s.getMutex().acquire();
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        System.out.println("Comienzo ejecución peaton: " + id);
        while (s.getTurno() != -1) {
                                /* Un peaton no puede pasar directamente si se cumple una de las siguiente
condiciones:
                                        1. No es su turno
                                        2. Ya están pasando el número máximo de Peatones
                                        3. Hay coches del turno anterior pasando
                                 */
            if (s.getTurno() != TURNO_PEATONES || s.getPasando()[TURNO_PEATONES] ==
MAX_PASAR_PEATONES
                    || s.getPasando()[ANTERIOR_TURNO] != 0) {
                // En caso de no pasar, se añade a esperando
                s.getEsperando()[TURNO_PEATONES]++;
                s.getMutex().release(); // Se libera el mutex
                try {
                    s.getSemaforosTurnos()[TURNO_PEATONES].acquire(); // Aquí se espera
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                // Una vez liberado se reduce el número de peatones esperando
                s.getEsperando()[TURNO_PEATONES]--;
            }
            // Si se ha terminado la ejecución se puede salir del bucle
            if (s.getTurno() == -1) {
                System.out.println("=Saliendo peaton " + id);
                break;
            }
            // Como está pasando se aumenta el núemor de peatones pasando
            s.getPasando()[TURNO_PEATONES]++;

            System.out.println("*Pasando peaton:" + id);
```

```
        // Si hay más peatones esperando y no se ha llegado todavía al máximo, se liberan más
peatones, despertar
        // encadenado
        if (s.getEsperando()[TURNO_PEATONES] != 0 && s.getPasando()[TURNO_PEATONES] <
MAX_PASAR_PEATONES) {
            System.out.println("#Peaton " + id + " hace un release." + s.getPasando()[TURNO_PEATONES]);
            // Se cede la exclusión mutua al siguiente proceso, por lo que no se tiene que liberar y no se
ha tenido
            // que pedir.
            s.getSemaforosTurnos()[TURNO_PEATONES].release();
        } else {
            // Sino se puede despertar ningún otro proceso se libera el mutex.
            System.out.println("%Peaton " + id + " no hace nada." + s.getPasando()[TURNO_PEATONES]);
            s.getMutex().release();
        }
        // Pasar
        try {
            Thread.sleep(TIEMPO_CRUCE);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // Fin Pasar
        try {
            // Obtenemos otra vez la exclusión mutua para poder realizar las comprobaciones y actualizar
el
            // número de transeuntes pasando.
            s.getMutex().acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("-Ha pasado peaton: " + id);
        // Se reduce el número de transeuntes pasando
        s.getPasando()[TURNO_PEATONES]--;
        // Si seguimos en el mismo turno, se puede liberar a un @class Peaton esperando
        if (s.getTurno() == TURNO_PEATONES) {
            System.out.println("## Peaton " + id + " despierta siguiente");
            // Cedemos la exclusión mutua por lo que no tenemo que liberar el mutex
            s.getSemaforosTurnos()[TURNO_PEATONES].release();
            // Si no quedan peatones pasando y ya ha terminado nuestro turno despertamos un proceso
del siguiente turno
        } else if (s.getPasando()[TURNO_PEATONES] == 0 && TURNO_PEATONES != s.getTurno()) {
            // Esta acción solo debe realizarse por el último peatón pasando
            System.out.println("$Peaton " + id + " libera siguiente Turno. " + s.getPasando()
[TURNO_PEATONES]);
            s.getSemaforosTurnos()[0].release(); // Se cede la exclusión mutua al proceso despertado
        } else {
            // Como no hemos podido ceder la exclusión mutua, la liberamos
            System.out.println("%Peaton " + id + " no hace nada");
            s.getMutex().release();
```

```
        }
        // Se espera un tiempo antes de volver a pasar
        try {
            Thread.sleep(TIEMPO_ESPERA);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // Se necesita la exclusión mutua para poder realizar las comprobaciones del principio del bucle
        try {
            s.getMutex().acquire();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Peaton " + id + " vuelve a intentar cruzar.");
    }
    System.out.println("FIN peaton " + id);
    s.getMutex().release(); // Liberamos la exclusión mutua para no provocar un deadlock.
  }

}
```

MAIN

```
package ejercicio2;


/**
 * Clase donde de encuentra el main del ejercicio
 */
public class Ejercicio2 {

        /**
         * El problema
         * @param args no se utiliza
         */
        public static void main(String[] args) {
                SemaforoCarretera semaforo = new SemaforoCarretera(); // Semáforo que coordina
                Coche[] procesosCoches = new Coche[100]; // Los proceso que simulan coches
                Peaton[] procesosPeatones = new Peaton[50]; // Los procesos que simular peatones

                System.out.println("Contruyendo procesos");
                // Los primeros cincuenta coches empiezan en la dirección Norte-Sur (0)
                for (int i = 0; i < 50; i++)
                        procesosCoches[i] = new Coche(semaforo, 0, i);

                // Los restantes empiezan en la dirección Este-Oeste (1)
                for (int i = 50; i < 100; i++)
                        procesosCoches[i] = new Coche(semaforo, 1, i);
```

```
                // Definición de los peatones que pasan por el paso de cebra
                for (int i = 0; i < 50; i++)
                        procesosPeatones[i] = new Peaton(semaforo, i);

                System.out.println("Comenzando Ejecución");

                // Lanzamos procesos
                semaforo.start();
                for (int i = 0; i < procesosCoches.length; i++)
                        procesosCoches[i].start();

                for (int i = 0; i < procesosPeatones.length; i++)
                        procesosPeatones[i].start();

                // Esperamos que terminen
                try {
                        semaforo.join();
                        for (int i = 0; i < procesosCoches.length; i++)
                                procesosCoches[i].join();

                        for (int i = 0; i < procesosPeatones.length; i++)
                                procesosPeatones[i].join();
                } catch (Exception e) {
                        e.printStackTrace();
                }
                System.out.println("\n\nFin de la ejecución");
        }
}
```

2.3 CUESTIONES

2.3.1 CUESTIÓN A

¿Qué acciones pueden realizar simultáneamente los hilos?

Un proceso de transeúnte (de tipo coche o peatón) se puede dividir en cuatro partes según su comportamiento: antes de pasar, pasando, después de pasar y esperando siguiente paso. La primera y la tercera parte consisten de una serie de comprobaciones de la variables compartidas que requieren tener la exclusión mutua (salvo en la parte de la primera donde se espera en *semaforosTurnos[turnoPasar]*) y se separan de la anterior y siguiente parte por la petición y liberación del mutex, respectivamente. Las otras dos consisten en su mayoría de un comando *sleep* que simula el transeúnte pasando por el cruce o haciendo cualquier otra cosa en la parte de esperando siguiente paso (en esta parte también se cambia la dirección en el caso de los coches). Es por esto que estas dos

partes se pueden ejecutar simultáneamente entre sí o consigo mismas, en tantos proceso como se pueda.

De la misma manera, casi todo el código del proceso *semaforoCarretera* requiere tener la exclusión mutua, siendo las únicas partes que no lo hacen y que por lo tanto pueden ejecutarse al mismo tiempo que el resto de procesos las de *sleep* y actualización del contador del bucle *for*.

2.3.2 CUESTIÓN B

Explica el papel de los semáforos que has usado para resolver el problema.

El primer semáforo es el mutex, es un semáforo binario que se encarga de asegurar la exclusión mutua de las variables compartidas.

Los otros tres son semáforos binarios que se encuentran en un array (*semaforosTurno*) y sirven para la sincronización de los procesos, cada semáforo representa una dirección o el semáforo que se encontraría en esa dirección en la vida real (de ahí que sean binario, rojo igual a cero y verde igual a 1). Como el cruce solo puede cruzado en una dirección a la vez, solo puede haber un semáforo “activo” a la vez, es decir, un semáforo que se puede poner a 1 a la vez. El comportamiento es parecido al paso de un testigo, cuando el turno cambia, el semáforo le pasa el testigo al siguiente del array. Para asegurarnos que de esto ocurra, cada semáforo despertará al siguiente cuando termine su turno y el último despertará al primero, sino se provocaría un deadlock. Así se puede hacer que los turnos ocurran de manera secuencial: primero norte-sur, a continuación este-oeste, finalmente peatones y vuelta a empezar (aunque cualquier combinación de direcciones sería correcta siempre que ocurran siempre igual, esta a sido la que se simula en esta resolución del problema).

2.3.3 CUESTIÓN C

¿Puede haber varios vehículos cruzando de Norte a Sur y de Este a Oeste simultáneamente? Justifica tu respuesta.

No, debido al comportamiento de los semáforos utilizados para controlar la sincronización de procesos, solo se puede cruzar una dirección a la vez. Lo que significa que solo puede haber proceso cruzando de norte a sur o de este a oeste, pero no a la vez.

3 EJERCICIO 3

3.1 RECURSOS NO COMPARTIBLES Y SINCRONIZACIÓN

En este problema podemos identifica que se utilizan una serie de máquinas y mesas. Como estas tienen que ser utilizadas por varios procesos clientes, podemos asumir que estos se tratan de recursos no compartidos. Ambos se gestionarán dentro de un monitor.

Para las máquinas, tenemos un subproblema similar al del problema de las impresoras visto en teoría. Con tres máquinas distintas y una cola común. En el monitor tendremos una cola, una array de booleanos cada uno simbolizando una máquina y un contador con el número de máquinas libres. Cuando no halla ninguna máquina libre, el proceso se meterá a la cola, sino, se le asignará la primera máquina libre. Al liberar, el proceso pasará la máquina libre, esta se marcará como tal en el monitor y se despertará al siguiente proceso, antes de terminar se deberá devolver las colas de las mesas al proceso cliente, para que este pueda seleccionar una e imprimirlas por pantalla.

Para las mesas, como cada mesa tiene su propia cola, estas funcionan de manera similar a un monitor para la exclusión mutua. Las dos diferencias con un monitor para la exclusión mutua serían: que el id de la mesa a la que se quiere acceder debería ser especificado por al invocar los procesos de petición y liberación y que las variables de tal deberían estar en un array con el que se accedería con el id especificado; y que se tiene que llevar la cuenta de las cantidades de clientes esperando en cada cola para que los clientes que todavía no han elegido mesa puedan coger la cola con menor tiempo de espera como se especifica en el problema. Por lo demás, tendrían la misma estructura que un monitor de exclusión mutua visto en teoría.

La única condición de sincronización que se puede ver es que los procesos clientes deben primero acceder a una máquina y después a una mesa. Sin embargo, esto se determinará en el código del propio cliente y no se gestiona en el monitor.

Finalmente, decir que como el problema requiere que se imprima por pantalla, esta también será un recurso no compartido. Para asegurar la exclusión mutua se utilizará un monitor separado igual al monitor *semáforo_binario* visto en la teoría.

En resumen, los recursos no compartidos son la máquinas y las mesas que se gestionarán con un monitor, donde las máquinas tendrán una cola compartida para todas y las mesas una cola por mesa. En otro monitor, se asegurará la exclusión mutua de la pantalla.

3.2 CÓDIGO

3.2.1 PSEUDOCÓDIGO

```
Monitor banco:
export irAMáquina, liberarMáquina, esperaMesa, liberarMesa;
var
      //// Para las máquinas
      colaMaquina: Condition; // Cola común para todas las máquina
      maquinasLibres: array[0..2] de boolean; // Máquinas, si están a true están
libres
      maquinas: int; // Número de máquinas disponibles
      //// Para las mesas
      colasMesas: array[0..3] de Condition; // Colas individuales de las mesas
      // El número de personas esperando en la cola de mesa, si llega a cero es
porque la mesa está libre
      esperandoMesa: array[0..3] de int;
      mesasLibres: array[0..3] de boolean; // Indican que mesas están libres y
cuales no, empiezan todas a true
begin

      // El cliente intenta coger una máquina. Si no hay libre, espera.
      procedure irAMáquina():
      var máquina: int;
      begin
            // El cliente comprueba si hay alguien en la cola
            // o todas las máquina están ocupadas. En caso de estarlo,
            // se espera
            if maquinas == 0 then
                  delay(colaMáquina);
            id = 0;
            while not(maquinasLibres[id]) do
                  i += 1;
            maquinasLibres[id] = false;
            maquinas --;

            return id; // Devolvemos la máquina libre al cliente.
      end;


      // El cliente libera una máquina cuando termina de usarala.
      procedure liberarMáquina(int máquinaUsada):
      begin
            máquinasLibres[máquinaUsada] = true; // Libera la máquina
            maquinas++;
            resume(colaMáquina); // Se reanuda la cola de las máquina

            return copy(esperandoEnMáquina); // Le pasamos el estado actual de
las mesas para que no
      end;
```

```
	// Se espera en la cola de la mesa
	procedure esperarMesa(int cola)
	begin
		// Como en Java se puede liberar la cola en orden con el atributo
fairness de ReentrantLock
		// no necesitamos asegurarlo de ninguna otra manera.
		if not mesasLibres[cola] then
		begin
			esperandoMesa[cola]++;
			delay(colasMesas[cola]);
			// El "uso de la Mesa" (el sleep) se realiza fuera para evitar
bloquear el monitor
		end else
			mesasLibres[cola] = false;
		end;
	end;

	// Se despierta al siguiente proceso de la colas de mesa
	procedure liberarMesa(int cola):
	begin
		if esperandoMesa[cola] > 0 then
		begin
			esperandoMesa[cola]--;
			resume(colasMesas[cola]);
		end else
			mesasLibres[cola] = true;
		end;
	end;
end;

// Se implementa un semáforo binario estándar para el uso de la pantalla
Monitor Pantalla;
export esperaPantalla, liberaPantalla;
var
	colaPantalla: Condition;
	pantallaLibre: boolean;
begin
	procedure esperaPantalla()
	begin
		if pantallaLibre
		begin
			delay(colaPantalla);
		end else
		begin
			pantallaLibre=false;
		end;
	end;

	procedure liberarPantalla()
	begin
		if not empty(colaPantalla)
```

```
            begin
                  resume(colaPantalla);
            end else
                  pantallaLibre = true;
            end;
      end;
end;

process type Cliente(banco: Monitor Banco, pantall: Monitor Pantalla):
begin
      var x, y: int;
      var máquinaAsignada : int;
      var colaMesa : int;
      var personasEnColaMesas: array[0..3] de int;
end
begin
      // Generar Números Aleatorios X e Y
      x = rand();
      y = rand();

      // Utiliza la máquina
      máquinaAsignada = banco.irAMáquina();
      sleep(x); // El tiempo que está utilizando Máquina
      // Liberamos las máquina
      personasEnColaMesas = banco.liberarMáquina(máquinaUtilizada); // Este
proceso también devuelve los tiempos de espera

      colaMesa = menorColaDeMesa(personasEnColaMesas); // Selecciona la cola con
menos personas

      // Imprime el mensaje
      pantalla.esperaPantalla(); // Espera que le den la exclusión mutua
      writeln(formarMensaje(x,y,máquinaAsignada,personasColaMesa,colaMesa)); //
Escribe
      pantalla.liberarPantalla(); // Libera la pantalla

      // Usar la mesa
      banco.esperarEnCola(colaMesa); // Espera la exclusión mutua
      sleep(y); // Simula ser atendido en la mesa
      banco.liberarCola(colaMesa); // Liberar la mesa
end;

// Función auxiliar que devuelve el indice de la cola de mesas con menor número
de personas esperando
int function menorColaDeMesa(array de int:colasMesas):
var
      indiceDeMenor:int ; // El indice de la cola con menos personas esperando
begin
      // Se supone que la menor mesa es la primera y se va comprobando con las
demás
      indiceDeMenor = 0
      for i:= 1..colasMesas.length - 1 do:
```

```
      begin
            // Se comprueba si la cola de indice i es menor que la actual
            if esperandoMesa[indiceDeMenor] > esperandoMesa[i]
            begin
                  // De ser menor i se convierte en el nuevo menor
                  indiceDeMenor = i;
            end;
      end;

      return indiceDeMenor; // Se devulve el indice de la cola que menos
personas tiene
end;

main()
begin
var
      banco: Monitor Banco;
      pantall: Monitor Pantalla;
      clientes: array[0..49] de Cliente;
end
begin
      // Inicializar variables
      banco = new Monitor Banco();
      pantalla = new Monitor Pantalla();

      for i := 0 .. 49 do
      begin
            cliente[i] = new Cliente(banco, pantalla);
      end;

      // Lanzar procesos
      cobegin
            banco;
            pantalla;
            for i := 0 .. 49 do
            begin
                  cliente[i];
            end;
      coend;

end;
```

3.2.2 CÓDIGO JAVA

Nota: la asignación de las colas de las mesas se ha realizado en el monitor Banco en vez de en el Cliente como se comenta en el enunciado, porque debido a la implementación de Java, los clientes tendían a acumularse en una cola hasta que se actualizaba. Esto ocurria con cinco o seis procesos a la vez, quedando la cola de la primera mesa llena y el resto vacias.

BANCO

```java
package ejercicio3;

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

/**
 * Clase principal del ejercicio 3. Se encarga de coordinar el uso de las máquinas y las mesas como recursos no
 * compartibles
 */
public class Banco extends Thread {
    /**
     * Constante que contiene la cantidad de Máquina presentes
     */
    private final static int N_MAQUINAS = 3;
    /**
     * Constante que contiene la cantidad de mesas existentes
     */
    private final static int N_MESAS = 4;
    /**
     * Cerrojo que aseguara la ejecución de los procesos en exclusión mutua
     */
    ReentrantLock l;
    ///// Para las máquinas
    /**
     * Cola de las máquinas
     */
    Condition colaMaquina;
    /**
     * Array de booleanos, cuando un elemento se encuentra a true quiere indicar que la máquina del id igual al indice
     * se encuentra libre.
     */
    boolean[] maquinasLibres;
    /**
     * Entero que indica el número de máquinas libres en el momento
     */
    int maquinas;
    // Para las mesas
    /**
     * Colas de las mesas, cada mesa tiene su propia cola
     */
    final Condition[] colasMesas;
    /**
     * Número de personas esperando en la cola de mismo índice
     */
    int[] esperandoMesa;
```

```java
/**
 * Indica si la mesa relacionada con la cola del mismo índice se encuentra libre
 */
boolean[] mesasLibres;

/**
 * Constructor del Monitor Banco
 */
public Banco() {
    this.l = new ReentrantLock(true);
    this.colaMaquina = l.newCondition();
    this.maquinasLibres = new boolean[N_MAQUINAS];
    this.maquinas = N_MAQUINAS;
    for (int i = 0; i < N_MAQUINAS; i++)
        maquinasLibres[i] = true;

    this.colasMesas = new Condition[N_MESAS];
    this.esperandoMesa = new int[N_MESAS];
    this.mesasLibres = new boolean[N_MESAS];

    for (int i = 0; i < N_MESAS; i++) {
        this.colasMesas[i] = l.newCondition();
        this.esperandoMesa[i] = 0;
        this.mesasLibres[i] = true;
    }
}

/**
 * Método para solicitar la exclusión mutua de una máquina
 *
 * @return el id de la máquina a utilizar
 */
public int irAMaquina() {
    int id = 0;
    l.lock();
    try {
        while (maquinas == 0)
            colaMaquina.await();

        while (!maquinasLibres[id])
            id++;
        maquinasLibres[id] = false;
        maquinas--;
    } catch (Exception ex) {
        ex.printStackTrace();
    } finally {
        l.unlock();
    }
    return id;
```

```
    }

    /**
     * Método para liberar la exclusión mutua de la máquina utilizada
     *
     * @param id el id de la máquina que ha sido utilizada
     */
    public void liberarMaquina(int id) {
        int[] copiaEsperandoMesa;
        l.lock();
        try {
            maquinasLibres[id] = true;
            maquinas++;
            colaMaquina.signal();
        } catch (Exception exception) {
            exception.printStackTrace();
        } finally {
            l.unlock();
        }
    }

    /**
     * Método para pedir cola de mesa, también "reserva" sitio en la cola. Se realiza en banco en vez de en cliente,
     * porque si se hace en cliente se tienden a apilar todos en una sola. Sin embargo, al ejecutarse en Banco, como
     * tiene acceso directo al array de seguiemiento de tiempos de colas, se ejecuta correctamente.
     *
     * @return un objeto que contiene la cola asignada y el estado de los tiempos en el momento de asignarla.
     */
    public Mensaje solicitudAdelantadaMesa() {
        l.lock();
        int mejorCola;
        int[] copiaColas;
        try {
            mejorCola = menorColaDeMesa(esperandoMesa);
            copiaColas = esperandoMesa.clone();
            esperandoMesa[mejorCola]++;
        } finally {
            l.unlock();
        }
        return new Mensaje(copiaColas, mejorCola);
    }

    /**
     * Método para solicitar la exclusión mutua de una mesa en concreto
     *
     * @param id el id de la mesa que se quiere utilizar
```

```
     */
    public void esperarMesa(int id) {
        l.lock();
        try {
            if (!mesasLibres[id]) {
                colasMesas[id].await();
            } else {
                esperandoMesa[id]--;
                mesasLibres[id] = false;
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            l.unlock();
        }
    }

    /**
     * Método para liberar la exclusión mutua de una mesa
     *
     * @param id el id de la mesa de la que se quiere liberar
     */
    public void liberarMesa(int id) {
        l.lock();
        try {
            if (esperandoMesa[id] > 0) {
                esperandoMesa[id]--;
                colasMesas[id].signal();
            } else {
                mesasLibres[id] = true;
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            l.unlock();
        }
    }


    /**
     * Función auxiliar para calcular cuál es la cola con el menor número de personas.
     *
     * @param personasEnColaMesas Array de colas, los valores son el número de personas esperando
     * @return El indice de la cola con menor número de personas esperando
     */
    private int menorColaDeMesa(int[] personasEnColaMesas) {
        // El indice de la cola con menor personas esperando
        int indiceMenor = 0; // Suponemos que el primero es el menor y comprobamos a partir de ahí
        for (int i = 1; i < personasEnColaMesas.length; i++)
```

```
            // Se comprubea si la mesa de indice i es menor que la actual menor
            if (personasEnColaMesas[indiceMenor] > personasEnColaMesas[i])
                // En caso de que lo sea, esta se convierte en la nueva menor
                indiceMenor = i;

        // Devolvemos cual es el menor
        return indiceMenor;
    }
}
```

CLIENTE

```
package ejercicio3;

/**
 *  Clase cliente que simula un cliente en un banco para el ejercicio 3
 */
public class Cliente extends Thread {
    /**
     * Id del proceso
     */
    private int id;
    /**
     * Tiempo de espera en máquina
     */
    private int x;
    /**
     * Tiempo de espera en Mesa
     */
    private int y;
    /**
     * Máquina que se le ha asignado
     */
    private int maquinaAsignada;
    /**
     * Id de la mesa utilizada
     */
    private int colaMesa;
    /**
     * Copia del estado de las colas a la hora de elegirla mejor
     */
    private int[] personasEnColaMesas;
    /**
     * Puntero al monitor Banco que controla la
     */
    private Banco banco;
    /**
     * Puntero al monitor Pantalla que controla la exclusión mutua de la pantalla
```

```
 */
private Pantalla pantalla;


/** Constructor de la clase Clinete, requiere del id y los punteros a los monitores usado
 */
public Cliente(int id, Banco banco, Pantalla pantalla) {
    this.id = id;
    this.banco = banco;
    this.pantalla = pantalla;
    this.x = (int) (Math.random() * 10000) + 1;
    this.y = (int) (Math.random() * 20000) + 1;
}

/**
 * Método que se ejecuta en concurrencia
 */
@Override
public void run() {
    super.run();

    // Utilizar la máquina
    maquinaAsignada = banco.irAMaquina(); // Pedir una máquina
    try {
        // Simular uso
        Thread.sleep(x);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
    // Liberar máquina y obtener tiempos colas
    banco.liberarMaquina(maquinaAsignada);

    // Obtener la cola con menor tiempo de espera
    // colaMesa = menorColaDeMesa(personasEnColaMesas);
    Mensaje msj = banco.solicitudAdelantadaMesa();
    personasEnColaMesas = msj.getColas();
    colaMesa = msj.getColaAsignada();

    pantalla.esperarPantalla();
    System.out.println("------------------------------------------------------------" +
            "\nCliente " + id + " ha solicitado su servicio en la máquina: " + maquinaAsignada +
            "\nTiempo en solicitar el servicio: " + x +
            "\nSerá atendido en la mesa: " + colaMesa +
            "\nTiempo en la mesa = " + y +
            "\nTiempo de espera en la mesa1 = " + personasEnColaMesas[0] + ", mesa2 = " +
personasEnColaMesas[1] +
            ", mesa3 = " + personasEnColaMesas[2] + ", mesa4 = " + personasEnColaMesas[3] +
            "\n------------------------------------------------------------");
    pantalla.liberarPantalla();
```

```
            banco.esperarMesa(colaMesa);
            try {
                // Simular uso
                Thread.sleep(y);
            } catch (InterruptedException e) {
                throw new RuntimeException(e);
            }
            banco.liberarMesa(colaMesa);
        }

    }
```

PANTALLA

```
package ejercicio3;

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

/**
 * Clase monitor para controlar la exclusión mutua de la pantalla
 */
public class Pantalla extends Thread {
    /**
     * Cerrojo del monitor
     */
    private final ReentrantLock l;
    /**
     * Cola asociada a la pantalla
     */
    private final Condition pantalla;
    /**
     * Variable que indica si la pantalla se encuentra libre o no
     */
    private boolean pantallaLibre;
    /**
     * Indica el número de procesos esperando la exclusión mutua. Se utiliza por la falta de la función empty, se podría
     * utilizar l.hasQueuedThreads(), pero no es correcto.
     */
    private int esperando;

    /**
     * Cosntructor de la clase
     */
    public Pantalla() {
```

```
        this.l = new ReentrantLock(true);
        this.pantalla = l.newCondition();
        this.pantallaLibre = true;
        this.esperando = 0;
    }

    /**
     * Pedir la exclusión mutua del recurso no compartible Pantalla
     */
    public void esperarPantalla() {
        l.lock();
        try {
            if (!pantallaLibre) {
                esperando++;
                pantalla.await();
            } else {
                pantallaLibre = false;
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            l.unlock();
        }
    }

    /**
     * Liberar la exclusión mutua del recurso no compartible Pantalla
     */
    public void liberarPantalla() {
        l.lock();
        try {

            if (esperando > 0) {
                esperando--;
                pantalla.signal();
            } else {
                pantallaLibre = true;
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        } finally {
            l.unlock();
        }
    }
}
```

MENSAJE

```java
package ejercicio3;

/**
 * Clase que se utiliza en vez de un struct, para pasar dos parámetros por un return. Únicamente para pasar info.
 */
public class Mensaje {
    /**
     *  Copia del estado de las colas en el momento de asignar la mejor
     */
    private final int[] colas;
    /**
     * Cola asignada
     */
    private final int colaAsignada;


    /**
     * Contructor
     * @param colas las colas
     * @param colaAsignada la cola asignada
     */
    public Mensaje(int[] colas, int colaAsignada) {
        this.colas = colas.clone();
        this.colaAsignada = colaAsignada;
    }

    public int[] getColas() {
        return colas;
    }

    public int getColaAsignada() {
        return colaAsignada;
    }
}
```

MAIN

```java
package ejercicio3;


/**
 * Clase donde se contiene el método main del problema
 */
public class Ejercicio3 {
    /**
```

```
     * El problema
     * @param args mete lo que quieras que no voy a coger nada
     */
    public static void main(String[] args) {
        Banco banco = new Banco();
        Pantalla pantalla = new Pantalla();
        Cliente[] clientes = new Cliente[50];

        for (int i = 0; i < clientes.length; i++)
            clientes[i] = new Cliente(i, banco, pantalla);

        System.out.println("Inicio de la  ejecución\n\n");
        banco.start();
        pantalla.start();

        for (Cliente cliente : clientes) cliente.start();

        try {
            for (Cliente cliente : clientes) cliente.join();
        } catch (Exception ex) {
            ex.printStackTrace();
        }

        System.out.println("\n\nFin de la ejecución");

    }
}
```

3.3 CUESTIONES

3.3.1 CUESTIÓN A

Indica si la acción de ser atendido en las mesas es concurrente o en exclusión mutua justificando la respuesta.

La acción de ser atendido en una mesa se ejecuta en exclusión mutua con todos los clientes que quieren acceder a la misma mesa pero de manera concurrente con las otras mesas. La acción en si misma es la sección crítica, siendo la anterior, *esperarCola*, el protocolo de entrad y la posterior, *liberarCola*, el protocolo de salida. Sin embargo, como cada mesa es independiente la una de la otra, se ejecutan concurrentemente entre sí.

3.3.2 CUESTIÓN B

¿Qué tipo de monitor Java has usado?. Justifica la respuesta.

Se ha utilizado un monitor con `ReentrantLock` y `Condition` porque se necesitan cinco colas distintas dentro de un solo monitor, una para las máquinas y cuatro para las mesas (una por mesa), cosa que no posible utilizando `synchronized`.

3.3.3 Cuestión C

En el monitor diseñado, ¿has usado notify/signal o notifyAll/signalAll? Justifica la respuesta.

Se ha utilizado solo `signal` y no `signalAll` porque el monitor se utiliza para proteger la exclusión mutua de una serie de recursos no compartibles que se liberan y ocupan de uno en uno. Cuando se libera, por ejemplo una máquina, solo es necesario que el siguiente cliente en la cola se notifique para que ocupe la que se acaba de liberar, todos los procesos ocupan un solo recurso no compartible y todos liberan un solo recurso no compartido, por lo que, al liberarlo, no es necesario despertar mas que el siguiente que quiera ocuparlo.

3.3.4 Cuestión D

¿Cómo se ha resuelto la exclusión mutua de la pantalla en este problema?

Se ha creado un monitor aparte de exclusión mutua. El permiso es dado en el propio monitor, así como la gestión de las colas. Mientras que la impresión se realiza en el propio cliente.

4 Ejercicio 4

4.1 Recursos No Compartibles y Secciones Críticas

Los recursos no compartibles en este algoritmo son las cajas A y B y las pantalla. Los dos primeros lo hacen con buzones centralizados en el controlador, mientras que la pantalla se hace a través de paso de testigo.

En las cajas tenemos primero una simulación de una cola inicial de solicitud de caja donde se le asigna un tiempo de espera y una caja según este tiempo de espera al proceso Persona. En el pseudocódigo esto se consigue teniendo un buzón centralizado de solicitudes donde las Personas envían sus buzones donde esperan respuesta, de esta manera el controlador puede ir asignando y enviando los resultados a los buzones respuesta recibidos a través del buzón de solicitudes centralizado. De la misma manera se utilizan los buzones de solicitud y liberación de caja. En el de solicitud de caja que solo se lee si la variable booleana de caja dice que esta se encuentra libre, se le da el estado ocupado a la caja, dándole así la exclusión mutua al proceso Persona que envió el buzón de respuesta. Por el contrario, en el buzón liberar solo se lee cuando la caja

correspondiente se encuentra ocupada y solo le puede enviar mensajes el proceso Persona que actualmente tiene la exclusión mutua.

Cuando los procesos Persona tienen la exclusión mutua pueden ejecutar la sección critica que se encuentra entre las llamadas de abrir caja y liberar caja que garantizan la exclusión mutua. Como se trata de una simulación solo se duerme el proceso en estas.

El recurso no compartible pantalla asegura la exclusión mutua mediante paso de testigos. El proceso main envía el primer mensaje. Desde entonces, los procesos Persona solo tiene que ejecutar la sección critica entre las llamadas de recibir mensaje y enviar otro para pasar el testigo al siguiente proceso.

4.2 CÓDIGO

4.2.1 PSEUDOCÓDIGO

```
process Controlador(procesosRestantes: int, /** todos los buzones de recibir **/
)
var
      /****** Buzones de recibir ******/
      solicitudCaja: mailbox of mailbox of int;
      // Recibe las solicitudes de los procesos persona que
      // quieren ser asignados a una caja.

      buzonTerminar: mailbox of mailbox of string;
      // Recibe mensajes de los procesos que terminan. Va asociado a la variable
      // procesosRestantes, cada vez que recibe uno, esta se decrementa. Cuando
      // Llegue a cero el proceso controlador debe terminar.

      // Los siguientes buzones son para el control de la exclusión mutua
      // Sin embargo la exclusión mutua de la pantall se controla con un buzón
      // de testigo descentralizado.

      abrirCajaA: mailbox of mailbox of string;
      // Recibe peticiones de procesos persona que quieren obtener la
      // exclusión mútua de la caja A

      liberarCajaA: mailbox of mailbox of string;
      // Recibe mensajes del proceso que tiene la exclusión mútua de la
      // caja A para liberarla.

      abrirCajaB: mailbox of mailbox of string;
      // Recibe peticiones de procesos persona que quieren obtener la
      // exclusión mútua de la caja B.

      liberarCajaB: mailbox of mailbox of string;
      // Recibe mensajes del proceso que tiene la exclusión mútua de la
      // caja B para liberarla.
```

```
      /****** Entero ******/
      procesosRestantes: int;
      // Lleva la cuenta de los procesos que quedan ejecutando.
      // Cuando llega a cero se termina el proceso Controlador.

      /****** Booleanos ******/
      cajaALibre: bool;
      // Variable para el seguimiento de la exclusión mútua de la Caja A.

      cajaBLibre: bool;
      // Variable para el seguimiento de la exclusión mútua de la Caja B.

      escrituraLibre: bool;
      // Variable para el seguimiento de la exclusión mútua de la pantalla.
begin

      cajaALibre = true;
      cajaBLibre = true;
      escrituraLibre = true;

      repeat
            Select
                        receive(solicitudCaja, buzonRespuesta);
                        // Genera un número entre 1 y 10 que representa el
tiempo (en Segundos) a estar en caja
                        tiempoEspera = generarTiempoEspera();
                        // Si el tiempo es mayor o igual al mínimo requerido
para la caja A
                        // se le asigna esta de lo contrario será la B.
                        if (tiempoEspera >= TIEMPO_MIN_CAJA_A)
                              // Se envía una respuesta que contiene el tiempo
de espera y la caja asignada
                              send(buzonRespuesta, crearMensaje(tiempoEspera,
cajaA));
                        else
                              send(buzonRespuesta, crearMensaje(tiempoEspera,
cajaB));
                  or
                  when(cajaALibre) =>
                        receive(abrirCajaA, buzonRespuesta);
                        cajaALibre = false;
                        send(buzonRespuesta, "ok");
                  or
                  when(!cajaALibre) =>
                        receive(liberarCajaA, buzonRespuesta);
                        cajaALibre = true;
                        // No es necesario notificar la liberación
                  or
                  when(cajaBLibre) =>
                        receive(abrirCajaB, buzonRespuesta);
                        cajaBLibre = false;
                        send(buzonRespuesta, "ok");
```

```
                  or
                  when(!cajaBLibre) =>
                        receive(liberarCajaB, buzonRespuesta);
                        cajaBLibre = true;
                        // No es necesario notificar la liberación
                  or
                        receive(buzonTerminar, msj);
                        procesosRestantes--;
            end;
      until procesosRestantes == 0 ;
end;

process persona (id, /*Buzones del controlador*/, /*Buzones de la persona*/)
var
      /****** Buzones del controlador ******/
      solicitudCaja: mailbox of mailbox of int;
      abrirCajaA: mailbox of mailbox of string;
      liberarCajaA: mailbox of mailbox of string;
      abrirCajaB: mailbox of mailbox of string;
      liberarCajaB: mailbox of mailbox of string;
      buzonTerminar: mailbox of mailbox of string;

      /****** Buzones de la persona ******/
      respuestaSolicitudCaja: mailbox of string;
      respuestaAbrirCajaA: mailbox of string;
      respuestaAbrirCajaB: mailbox of string;
      respuestaSolicitudEscritura: mailbox of string;
      respuestaLiberaEscritura: mailbox of string;

      /****** Buzón control del a exclusión mutua pantalla con testigo ******/
      mutexEscritura: mailbox of mailbox of string;
      // Este buzón sirve para el control de la exclusión mutua mediante paso de
testigo.

      id: int;
      // El id del proceso

      caja: string; // También puede ser un enum
      // Guardamos la caja utilizada para imprimir en el mensaje más tarde

      tiempoPago: int;
      // El tiempo que pasa la persona en caja.

begin
      for i := 0..N_REPETICIONES do
      begin
            ///// 1 Realiza la compra
            sleep(rand());

            ///// 2 Solicita una Caja
            send(solicitudCaja, respuestaSolicitudCaja);
            receive(respuestaSolicitudCaja, msj);
```

```
            // Como solicitud caja tiene que devolver dos argumentos y el buzon
solo puede enviar uno,
            // Se ha elegido codificarlos en un mensaje. Se podrían poner juntos
en un struct, pero eso
            // sería más dificil de traducir en Java.
            tiempoPago, caja = decodificarMensaje(msj);

            ///// 3 y 4 realiza el pago en una caja y las libera
            // Solicitas exclusión mutua de la caja correspondiente
            if (caja = "A")
            begin
                  send(abrirCajaA, respuestaAbrirCajaA);
                  receive(respuestaAbrirCajaA, msj);
                  sleep(tiempoPago); // Simular realizar pago en caja
                  send(liberarCajaA, respuestaLiberarCajaA);
            end else
            begin
                  send(abrirCajaB, respuestaAbrirCajaB);
                  receive(respuestaAbrirCajaB, msj);
                  sleep(tiempoPago); // Simular realizar pago en caja
                  send(liberarCajaB, respuestaLiberarCajaB);
            end;

            ///// 5 Imprime en pantalla información

            receive(mutexEscritura, testigo); // Solicita la exlucisón mutua

            // Imprime por pantalla la información del pago
            // informaciónDelPago forma el mensaje según enunciado
            print(informaciónDelPago(id, caja, tiempoPago);

            send(mutexEscritura, testigo); // Liberar exclusión mutua
      end;

      send(buzonTerminar, "ok");
end;

main()
var
      /****** Buzones del controlador ******/
      solicitudCaja: mailbox of mailbox of int;
      abrirCajaA: mailbox of mailbox of string;
      liberarCajaA: mailbox of mailbox of string;
      abrirCajaB: mailbox of mailbox of string;
      liberarCajaB: mailbox of mailbox of string;
      buzonTerminar: mailbox of mailbox of string;

      /****** Buzones de la persona ******/
      respuestaSolicitudCaja: array[0..29] of mailbox of string;
      respuestaAbrirCajaA: array[0..29] of mailbox of string;
      respuestaAbrirCajaB: array[0..29] of mailbox of string;
```

```
      /****** Buzón control del a exclusión mutua pantalla con testigo ******/
      mutexEscritura: mailbox of mailbox of string;

      controlador: process Controlador;
      clientes: array[0.. N_CLIENTES-1] of process Persona;
begin
      /*Realiza la inicialización de los buzones aquí*/
      // Se obvia porque es muy simple

      writeln("Definiendo procesos");

      controlador = Controlador(solicitudCaja,
                        abrirCajaA,
                        liberarCajaA,
                        abrirCajaB,
                        liberarCajaB,
                        buzonTerminar,
                        N_CLIENTES);

      for i:= 0..N_CLIENTES-1 do
      begin
      clientes[i] = Cliente(i,
                  solicitudCaja,
                  abrirCajaA,
                  liberarCajaA,
                  abrirCajaB,
                  liberarCajaB,
                  solicitudEscritura,
                  liberaEscritura,
                  buzonTerminar,
                  respuestaSolicitudCaja[i],
                  respuestaAbrirCajaA[i],
                  respuestaAbrirCajaB[i],
                  mutexEscritura);
      end;

      writeln("Comenzando ejecución");

      // Se envia el primer testigo desde el main.
      send(mutexEscritura, testigo);

      // Lanzamos los procesos
      cobegin
            controlador;
            for c in clientes do
                  c;
            end;
      coend;

      writeln("Ejecución terminada");
end;
```

4.2.2 CÓDIGO JAVA

CONTROLADOR

```
package ejercicio4;

import messagepassing.MailBox;
import messagepassing.Selector;

/**
 * Clase principal del problema. Asigna personas a caja y controla la exclusión mutua de estas.
 */
public class Controlador extends Thread {
    /**
     * El tiempo máximo que puede estar una @class Persona en una caja
     */
    private static final int MAX_TIEMPO = 10;
    /**
     * El timepo mínimo que puede estar una @class Persona en una Caja
     */
    private static final int MIN_TIEMPO = 1;
    /**
     * El tiempo mínimo que se debe estimar que una persona estará par que se le asigne la caja A
     */
    private static final int MIN_TIEMPO_A = 5;
    /**
     * String de respuesta cuando esta da igual
     */
    private static final String OK = "ok";
    /**
     * El separador para el mensaje que contiene dos datos
     */
    public static final String SEPARADOR = ",";
    /**
     * String que indica la caja A
     */
    public static final String CAJA_A = "A";
    /**
     * String que indica la caja B
     */
    public static final String CAJA_B = "B";
    /****** Buzones de recibir ******/
    /**
     * Recibe las solicitudes de los procesos persona que quieren ser asignados a una caja.
     */
    private final MailBox solicitudCaja;
    /**
     * Recibe mensajes de los procesos que terminan. Va asociado a la variable procesosRestantes, cada vez
que recibe
```

```
   * uno, esta se decrementa. Cuando llegue a cero el proceso controlador debe terminar.
   */
  private final MailBox buzonTerminar;

  // Los siguientes buzones son para el control de la exclusión mutua
  // Sin embargo la exclusión mutua de la pantalla se controla con un buzón
  // de testigo descentralizado.

  /**
   * Recibe peticiones de procesos persona que quieren obtener la exclusión mútua de la caja A.
   */
  private final MailBox abrirCajaA;

  /**
   * Recibe mensajes del proceso que tiene la exclusión mútua de la caja A para liberarla.
   */
  private final MailBox liberarCajaA;

  /**
   * Recibe peticiones de procesos persona que quieren obtener la exclusión mútua de la caja B.
   */
  private final MailBox abrirCajaB;

  /**
   * Recibe mensajes del proceso que tiene la exclusión mútua de la caja B para liberarla.
   */
  private final MailBox liberarCajaB;

  /****** Buzones de respuetas******/
  // Como en Java la clase MailBox que representa un buzón no se puede pasar como parámetro por
MailBox, se debe tener
  // los buzones respuesta ya presentes en la clase. De esta manera, simplemente se puede pasar el id
del proceso y
  // responder al buzón con la posición igual a este id, que será el de dicho proceso.

  /**
   * Para responder a las peticiones de SolicitudCaja.
   */
  private final MailBox[] respuestaSolicitudCaja;

  /**
   * Para responder a las peticiones de AbrirCajaA.
   */
  private final MailBox[] respuestaAbrirCajaA;

  /**
   * Para responder a las peticiones de AbrirCajaB.
   */
  private final MailBox[] respuestaAbrirCajaB;
```

```
  /****** Selector******/
  /**
   * Objeto necesario para implementar un selector que nos permita escuchar las peticioens de varios buzones de
   * recibir al mismo tiempo.
   */
  private final Selector selector;

  /****** Entero ******/
  /**
   * Lleva la cuenta de los procesos que quedan ejecutando. Cuando llega a cero se termina el proceso
   * @class ejercicio4.Controlador .
   */
  private int procesosRestantes;

  /****** Booleanos ******/
  /**
   * Variable para el seguimiento de la exclusión mútua de la Caja A.
   */
  private boolean cajaALibre;

  /**
   * Variable para el seguimiento de la exclusión mútua de la Caja B.
   */
  private boolean cajaBLibre;

  /**
   * Contructor del proceso Controlador
   * @param solicitudCaja buzón en el que recibe las solicitudes de caja
   * @param buzonTerminar buzón en el que se informa que un proceso ha terminado
   * @param abrirCajaA buzón en el que se solicita la exclusión muta de la Caja A
   * @param liberarCajaA buzón en el que se libera la exclusión mutua de la Caja A
   * @param abrirCajaB buzón en el que se solicita la exblusión mutua de la Caja B
   * @param liberarCajaB buzón en el que se libera la exclusión mutua de la Caja B
   * @param respuestaSolicitudCaja array de buzones de respuesta para cuando se solicita una caja
   * @param respuestaAbrirCajaA array de buzones de respuesta para cuando se pide la exclusión mutua de la Caja A
   * @param respuestaAbrirCajaB array de buzones de respuesta para cuando se pide la exclusión mutua de la Caja B
   * @param procesosRestantes número de procesos con los que se empieza la ejecución
   */
  public Controlador(MailBox solicitudCaja, MailBox buzonTerminar, MailBox abrirCajaA, MailBox liberarCajaA,
                     MailBox abrirCajaB, MailBox liberarCajaB, MailBox[] respuestaSolicitudCaja,
                     MailBox[] respuestaAbrirCajaA, MailBox[] respuestaAbrirCajaB, int procesosRestantes) {
    this.solicitudCaja = solicitudCaja;
    this.buzonTerminar = buzonTerminar;
    this.abrirCajaA = abrirCajaA;
```

```
        this.liberarCajaA = liberarCajaA;
        this.abrirCajaB = abrirCajaB;
        this.liberarCajaB = liberarCajaB;
        this.respuestaSolicitudCaja = respuestaSolicitudCaja;
        this.respuestaAbrirCajaA = respuestaAbrirCajaA;
        this.respuestaAbrirCajaB = respuestaAbrirCajaB;
        this.procesosRestantes = procesosRestantes;
        // Las cajas compartidas están libres
        this.cajaALibre = true;
        this.cajaBLibre = true;
        // Definir el selector
        this.selector = new Selector();
        this.selector.addSelectable(this.solicitudCaja, false);
        this.selector.addSelectable(this.abrirCajaA, false);
        this.selector.addSelectable(this.liberarCajaA, false);
        this.selector.addSelectable(this.abrirCajaB, false);
        this.selector.addSelectable(this.liberarCajaB, false);
        this.selector.addSelectable(this.buzonTerminar, false);
    }

    /**
     * Método que se ejecuta cuando le lanza el proceso.
     */
    @Override
    public void run() {
        // Definimos las variables locales a utilizar en este metodo
        Integer id; // El id del proceso que se está comunicando con el controlador
        int tiempoEspera; // El tiempo de espera estimado para la solicitud de caja
        String caja; // Variable auxiliar para contener la caja asignada a la solicitud de caja

        do {
            // Denifinimos las guardas del selector, se han puesto en orden del según su aparición en el
switch
            solicitudCaja.setGuardValue(true);              // 1
            abrirCajaA.setGuardValue(cajaALibre);           // 2
            liberarCajaA.setGuardValue(!cajaALibre);        // 3
            abrirCajaB.setGuardValue(cajaBLibre);           // 4
            liberarCajaB.setGuardValue(!cajaBLibre);        // 5
            buzonTerminar.setGuardValue(true);              // 6

            switch (selector.selectOrBlock()) {
                case 1:
                    id = (Integer) solicitudCaja.receive(); // Se obtiene el id del proceso persona según el
mensaje
                    tiempoEspera = generarTiempoEspera(); // Se estima el tiempo que va a tardar

                    // Se le asigna una caja según el tiempo de pago
                    if (tiempoEspera >= MIN_TIEMPO_A) caja = CAJA_A;
                    else caja = CAJA_B;
```

```
                // Se le responde al proceso que solicitó con los datos estimados
                respuestaSolicitudCaja[id].send(crearMensaje(tiempoEspera, caja));
                break;
            case 2:
                id = (Integer) abrirCajaA.receive(); // Se obtiene el id del proceso persona según el mensaje

                // Se le asigna el valor false a la cajaA para que ningún otro proceso pueda pedirla
                cajaALibre = false;

                // Se le responde al proceso que preguntó para que sepa que la operación ha terminado
                respuestaAbrirCajaA[id].send(OK);
                break;
            case 3:
                liberarCajaA.receive(); // Se obtiene el id del proceso persona según el mensaje

                // Se le asigna el valor true a la cajaA para liberla
                cajaALibre = true;

                break;
            case 4:
                abrirCajaB.receive(); // Se obtiene el id del proceso persona según el mensaje

                // Se le asigna el valor false a la cajaB para que ningún otro proceso pueda pedirla
                cajaBLibre = false;

                break;
            case 5:
                liberarCajaB.receive();// Se obtiene el id del proceso persona según el mensaje

                // Se le asigna el valor true a la cajaA para liberla
                cajaBLibre = true;

                break;
            case 6:
                buzonTerminar.receive(); // Se recibe un mensaje cuando un proceso persona termina
                procesosRestantes--; // Se actualiza el número de procesos que siguen activos
        }
        // Se termina el proceso controlador cuando el número de procesos cliente sea cero.
    } while (procesosRestantes != 0);
}

/**
 * Codifica los argumentos en un mensaje de String para poder pasar varios valores por MailBox sin tener que crear
 * un objeto serializable. Este mensaje debe ser "decodificado" por el proceso que lo recibe.
 *
 * @param tiempoEspera tiempo en segundos que el proceso debe espera en caja
 * @param caja         caja asignada
```

```
     * @return un string que contiene el tiempo y la caja asignada segados por un separador
     */
    private String crearMensaje(Integer tiempoEspera, String caja) {
        return tiempoEspera.toString() + SEPARADOR + caja;
    }

    /**
     * Genera un número aleatorio entre MAX_TIEMPO y MIN_TIEMPO que representa el tiempo en segundos
que un proceso
     * tardará en pasar por caja
     *
     * @return un número entero aleatorio entre MAX_TIMEPO y MIN_TIEMPO
     */
    public Integer generarTiempoEspera() {
        return (int) (Math.random() * (MAX_TIEMPO - MIN_TIEMPO + 1)) + MIN_TIEMPO;
    }
}
```

PERSONA

```
package ejercicio4;

import messagepassing.MailBox;

/**
 * Clase que simula una persona en un supermercado
 */
public class Persona extends Thread {
    /**
     * Número de veces que se tiene que repetir el proceso
     */
    private static final int N_REPETICIONES = 5;
    /**
     * Número de milisegundos en un segundo
     */
    public static final int MILIS_A_SEGUNDOS = 1000;
    /**
     * Respuesta a un mensaje cuando no importa el contenido
     */
    public static final String OK = "ok";
    /****** Buzones de enviar ******/
    /**
     * Recibe las solicitudes de los procesos persona que quieren ser asignados a una caja.
     */
    private final MailBox solicitudCaja;
    /**
     * Recibe mensajes de los procesos que terminan. Va asociado a la variable procesosRestantes, cada vez
que recibe
     * uno, esta se decrementa. Cuando Llegue a cero el proceso controlador debe terminar.
```

```
     */
    private final MailBox buzonTerminar;

    // Los siguientes buzones son para el control de la exclusión mutua
    // Sin embargo la exclusión mutua de la pantall se controla con un buzón
    // de testigo descentralizado.
    /**
     * Recibe peiciones de procesos persona que quieren obtener la exclusión mútua de la caja A.
     */
    private final MailBox abrirCajaA;

    /**
     * Recibe mensajes del proceso que tiene la exclusión mútua de la caja A para liberarla.
     */
    private final MailBox liberarCajaA;

    /**
     * Recibe peticiones de procesos persona que quieren obtener la exclusión mútua de la caja B.
     */
    private final MailBox abrirCajaB;

    /**
     * Recibe mensajes del proceso que tiene la exclusión mútua de la caja B para liberarla.
     */
    private final MailBox liberarCajaB;

    /****** Buzones de recibir******/
    // Como en Java la clase MailBox que representa un buzón no se puede pasar como parámetro por
MailBox, se debe tener
    // los buzones respuesta ya presentes en la clase. De esta manera, simplemente se puede pasar el id
del proceso y
    // responder al buzón con la posición igual a este id, que será el de dicho proceso.

    /**
     * Para responder a las peticiones de SolicitudCaja.
     */
    private final MailBox respuestaSolicitudCaja;

    /**
     * Para responder a las peticiones de AbrirCajaA.
     */
    private final MailBox respuestaAbrirCajaA;

    /**
     * Para responder a las peticiones de AbrirCajaB.
     */
    private final MailBox respuestaAbrirCajaB;

    /****** Buzón control del a exclusión mutua pantalla con testigo ******/
```

```java
    /**
     * Este buzón sirve para el control de la exclusión mutua mediante paso de testigo.
     */
    private final MailBox mutexEscritura;

    /**
     *El id del proceso
     */
    private final Integer id;


    /**
     * Guardamos la caja utilizada para imprimir en el mensaje más tarde
     */
    private String caja; // También puede ser un enum

    /**
     * El tiempo que pasa la persona en caja.
     */
    private Integer tiempoPago;

    /**
     * Contructor de la clase
     * @param id el id del proceso
     * @param solicitudCaja buzón en el que realizar la solicitud de la caja
     * @param buzonTerminar buzón en el que notificar el final de la ejecución
     * @param abrirCajaA buzón en el que solicitar la exclusión mutua de la caja A
     * @param liberarCajaA buzón en el que notificar la liberación de la exblusión mutua de la caja A
     * @param abrirCajaB buzón en el que solicitar la exclusión mutua de la caja B
     * @param liberarCajaB buzón en el que notificar la liberación de la exblusión mutua de la caja B
     * @param respuestaSolicitudCaja buzón en que recibir la respuesta del controlador a solitudCaja
     * @param respuestaAbrirCajaA buzón en que recibir la respuesta del controlador a abrirCajaA
     * @param respuestaAbrirCajaB buzón en que recibir la respuesta del controlador a abrirCajaB
     * @param mutexEscritura buzón de paso de testigo para el control de la exclusión mutua de la escritura
en pantalla
     */
    public Persona(Integer id, MailBox solicitudCaja, MailBox buzonTerminar, MailBox abrirCajaA,
MailBox liberarCajaA, MailBox abrirCajaB, MailBox liberarCajaB, MailBox respuestaSolicitudCaja,
MailBox respuestaAbrirCajaA,  MailBox respuestaAbrirCajaB,  MailBox mutexEscritura) {
        this.id = id;
        this.solicitudCaja = solicitudCaja;
        this.buzonTerminar = buzonTerminar;
        this.abrirCajaA = abrirCajaA;
        this.liberarCajaA = liberarCajaA;
        this.abrirCajaB = abrirCajaB;
        this.liberarCajaB = liberarCajaB;
        this.respuestaSolicitudCaja = respuestaSolicitudCaja;
        this.respuestaAbrirCajaA = respuestaAbrirCajaA;
        this.respuestaAbrirCajaB = respuestaAbrirCajaB;
```

```
        this.mutexEscritura = mutexEscritura;
    }

    /**
     * Método auxiliar que genera un tiempo que la persona va a utilizar para pasearse por la tienda
     * @return el tiempo a pasearse por la tienda
     */
    private Integer tiempoPaseoPorLaTienda() {
        return ((int) (Math.random() * 10)+1) * MILIS_A_SEGUNDOS;
    }

    @Override
    public void run() {
        for (int i = 0; i < N_REPETICIONES; i++) {

            ///// 1 Realiza la compra
            try {
                // Este tiempo de espera simula a la persona paseando por la tienda o algo
                Thread.sleep(tiempoPaseoPorLaTienda());
            } catch (Exception ignored) {}

            ///// 2 Solicita una Caja
            solicitudCaja.send(id); // Se hace una solicitud de caja
            // Se espera la respuesta que es un mensaje que contiene el tiempo de espera en caja y la caja asignada
            String respuestaSolicitud = respuestaSolicitudCaja.receive().toString();

            // Este Método separa el mensaje escrito en String e inserta los valores en las variables tiempoPago y caja
            decodificarMensaje(respuestaSolicitud);

            ///// 3 y 4 realiza el pago en una caja y las libera
            // Solicitas exclusión mutua de la caja correspondiente
            if (caja.equals(Controlador.CAJA_A)) {
                abrirCajaA.send(id); // Solicita exclusión mutua
                respuestaAbrirCajaA.receive(); // Se espera la respuesta
                try {
                    Thread.sleep(tiempoPago * MILIS_A_SEGUNDOS); // Se espera el tiempo de pago
                } catch (Exception ignored) {
                }

                liberarCajaA.send(id); // Se envia petición de liberaciópn de la exclusión mutua
            } else {
                abrirCajaB.send(id); // Solicita exclusión mutua
                respuestaAbrirCajaB.receive(); // Se espera la respuesta
                try {
                    Thread.sleep(tiempoPago * MILIS_A_SEGUNDOS); // Se espera el tiempo de pago
                } catch (Exception ignored) {
                }
```

```
            liberarCajaB.send(id); // Se envia petición de liberaciópn de la exclusión mutua
        }

        ///// 5 Imprime en pantalla información
        mutexEscritura.receive(); // Solicita la exlucisón mutua

        // Imprime por pantalla la información del pago
        // informaciónDelPago forma el mensaje según enunciado
        System.out.println(informacionDelPago());

        mutexEscritura.send(OK); // Solicita la exlucisón mutua
      }
      buzonTerminar.send(OK); // Se informa al controlador cuando se termina de el proceso
   }

   /**
    * Método auxiliar para formar el mensaje a mostrar por pantalla
    * @return el mensaje bien formateado con la información requerida
    */
   private String informacionDelPago() {
      return "\nejercicio4.Persona " + id + " ha usado la caja " + caja + "\nTiempo de pago = " +
tiempoPago
            + "\n\tThread.sleep(" + tiempoPago + ")\n" + "ejercicio4.Persona " + id + " liberando la caja " +
caja;
   }

   /**
    * Método auxiliar que decodifica el mensaje devuelto por respuestaSolicitudCaja
    * @param respuestaSolicitud el mensaje devuelto por la respuesta de solicitudCaja
    */
   private void decodificarMensaje(String respuestaSolicitud) {
      String[] elementos = respuestaSolicitud.split(Controlador.SEPARADOR); // Se separa los elementos
por el separador
      // Se castea el primero a integer, este representa el tiempo de pago
      this.tiempoPago = Integer.valueOf(elementos[0]);
      this.caja = elementos[1]; // Este valor representa la caja a la que se le ha asignado el proceso
   }
}
```

Main

```
package  ejercicio4;

import messagepassing.MailBox;

/**
 * Clase donde se ejecuta el main
```

```
 */
public class Ejercicio4 {
   /**
    * Número de procesos cliente al principio de la ejecución del programa.
    */
   private static final int N_CLIENTES = 30;

   /**
    * El problema
    *
    * @param args no se usa
    */
   public static void main(String[] args) {
      /* Buzones de escritura para ejercicio4.Controlador */
      MailBox solicitudCaja = new MailBox();
      // Recibe las solicitudes de los procesos persona que
      // quieren ser asignados a una caja.

      MailBox buzonTerminar = new MailBox();
      // Recibe mensajes de los procesos que terminan. Va asociado a la variable
      // procesosRestantes, cada vez que recibe uno, esta se decrementa. Cuando
      // Llegue a cero el proceso controlador debe terminar.

      // Los siguientes buzones son para el control de la exclusión mutua
      // Sin embargo la exclusión mutua de la pantall se controla con un buzón
      // de testigo descentralizado.

      MailBox abrirCajaA = new MailBox();
      // Recibe peticiones de procesos persona que quieren obtener la
      // exclusión mútua de la caja A

      MailBox liberarCajaA = new MailBox();
      // Recibe mensajes del proceso que tiene la exclusión mútua de la
      // caja A para liberarla.

      MailBox abrirCajaB = new MailBox();
      // Recibe peticiones de procesos persona que quieren obtener la
      // exclusión mútua de la caja B.

      MailBox liberarCajaB = new MailBox();
      // Recibe mensajes del proceso que tiene la exclusión mútua de la
      // caja B para liberarla.

      /* Buzones de escritura para ejercicio4.Persona */
      // Como en Java la clase MailBox que representa un buzón no se puede pasar como parámetro
por MailBox, se debe tener
      // los buzones respuesta ya presentes en la clase. De esta manera, simplemente se puede pasar el
id del proceso y
      // responder al buzón con la posición igual a este id, que será el de dicho proceso.
```

```
    MailBox[] respuestaSolicitudCaja = new MailBox[N_CLIENTES];
    // Para responder a las peticiones de SolicitudCaja

    MailBox[] respuestaAbrirCajaA = new MailBox[N_CLIENTES];
    // Para responder a las peticiones de AbrirCajaA

    MailBox[] respuestaLiberarCajaA = new MailBox[N_CLIENTES];
    // Para responder a las peticiones de LiberarCajaA;

    MailBox[] respuestaAbrirCajaB = new MailBox[N_CLIENTES];
    // Para responder a las peticiones de AbrirCajaB;

    MailBox[] respuestaLiberarCajaB = new MailBox[N_CLIENTES];
    // Para responder a las peticiones de LiberarCajaB;

    MailBox mutexEscritura = new MailBox();

    /* Clases de procesos */
    Controlador controlador;
    Persona[] clientes;

    System.out.println("Definiendo clases.");

    for (int i = 0; i < N_CLIENTES; i++) {
      respuestaSolicitudCaja[i] = new MailBox();
      respuestaAbrirCajaA[i] = new MailBox();
      respuestaLiberarCajaA[i] = new MailBox();
      respuestaAbrirCajaB[i] = new MailBox();
      respuestaLiberarCajaB[i] = new MailBox();
    }

    controlador = new Controlador(solicitudCaja, buzonTerminar, abrirCajaA, liberarCajaA, abrirCajaB,
liberarCajaB,
        respuestaSolicitudCaja, respuestaAbrirCajaA, respuestaLiberarCajaA, N_CLIENTES);

    clientes = new Persona[N_CLIENTES];
    for (int i = 0; i < N_CLIENTES; i++)
      clientes[i] = new Persona(i, solicitudCaja, buzonTerminar, abrirCajaA, liberarCajaA, abrirCajaB,
          liberarCajaB, respuestaSolicitudCaja[i], respuestaAbrirCajaA[i], respuestaAbrirCajaB[i],
          mutexEscritura);

    System.out.println("Comezando Ejecución");
    // Se envia el primer testigo desde el main.
    mutexEscritura.send("ok");
    // Lanzamos los procesos
    controlador.start();
    for (Persona p : clientes)
      p.start();
```

```
        try {
            controlador.join();
            for (Persona p : clientes)
                p.join();
        } catch (Exception ex) {
            ex.printStackTrace();
        }

        System.out.println("Ejecución terminada");
    }
}
```

4.3 CUESTIONES

4.3.1 CUESTIÓN A

¿Se pueden usar simultáneamente las dos cajas? Justifica la respuesta.

Sí, se pueden utilizar ambas cajas simultáneamente, porque el buzón que recibe las peticiones de la exclusión mutua, así como las variables que se utilizan para seguirla, son diferentes.

Por lo que si existieran dos procesos, cada uno asignado a una caja y asumiendo que ambas estuvieran libres, podrían ejecutarse simultáneamente. Si bien, en caso de que ambos procesos hicieran las petición al mismo tiempo, se tendrían que otorgar la exclusión mutua de cada caja de forma secuencial y el uso de estas de manera simultanea.

4.3.2 CUESTIÓN B

¿Cómo has resuelto la exclusión mutua de la pantalla?

Se ha utilizado un buzón en los procesos clientes que aseguran la exclusión mutua por medio de paso de testigo. Solo se pasa un testigo en el método main del programa, por lo que solo un proceso puede tenerlo y cuando lo tienen pueden acceder a la sección critica, asegurando la exclusión mutua. Como no se trata de una caja, este no es necesario incluirlo en el controlador, aunque se podría incluir de la misma manera que las cajas.