MEMORIA

Programación Concurrente y Distribuida

Nombre Completo	Subgrupo	DNI
Juan Pedreño García	2.4	
Patricia Cuenca Guardiola	2.4	

Índice

Ejercicio 1: Introducción a la programación concurrente

Recursos no compartibles

En este ejercicio, la pantalla sería un recurso no compartible ya que está dentro de una sección crítica protegida porque dos hilos no pueden escribir a la vez en pantalla.

Condiciones de sincronización

Para garantizar el acceso en exclusión mutua a la pantalla se utiliza un lock, el cual bloquea mientras un proceso generador está imprimiendo su resultado por pantalla y desbloquea cuando este finaliza.

Pseudocódigo

```
process type hilo_multiplicador(lock: mutex)
var
  A, resultado: matriz[3][3]
  m: Matriz
  iter: integer
begin
  for iter := 1 to 10 do
    m := generar_matriz()
    A := m
    resultado := multiplicar_matriz(A)
    lock.lock()
    println("Iteración ", iter, " - Hilo Multiplicador (A × A):")
    imprimir("Matriz A:", A)
    imprimir("Resultado A × A:", resultado)
    lock.unlock()
  end for
end
```

```
process type hilo_sumador(lock: mutex)
var
  A, resultado: matriz[3][3]
  m: Matriz
  iter: integer
begin
  for iter := 1 to 10 do
    m := generar_matriz()
    A := m
    resultado := sumar_matriz(A)
    lock.lock()
    println("Iteración ", iter, " - Hilo Sumador (A + A):")
    imprimir("Matriz A:", A)
    imprimir("Resultado A + A:", resultado)
    lock.unlock()
  end for
end
procedure main()
var
  lock: mutex
begin
  lock := new mutex
    hilo_multiplicador(lock)
    hilo_sumador(lock)
end
process type matriz():
```

```
procedure generar_matriz()
var
  i, j: integer
  m: matriz[3][3]
begin
  for i := 0 to 2 do
    for j := 0 to 2 do
      m[i][j] := randomInt(0, 9)
  return m
end
procedure sumar_matriz(A: matriz[3][3])
var
  i, j: integer
  resultado: matriz[3][3]
begin
  for i := 0 to 2 do
    for j := 0 to 2 do
      resultado[i][j] := A[i][j] + A[i][j]
  return resultado
end
function multiplicar_matriz(A: matriz[3][3])
var
  i, j, k: integer
  resultado: matriz[3][3]
begin
  for i := 0 to 2 do
```

```
    for j := 0 to 2 do
      resultado[i][j] := 0
      for k := 0 to 2 do
        resultado[i][j] := resultado[i][j] + A[i][k] * A[k][j]
  return resultado
end
procedure imprimir(titulo: string, A: matriz[3][3])
var
  i, j: integer
begin
  println(titulo)
  for i := 0 to 2 do
    for j := 0 to 2 do
      print(A[i][j], " ")
    println("")
  println("")
end
```

Cuestiones planteadas

a) Si no utilizamos ningún mecanismo de sincronización, las líneas de un hilo se podrían intercalar con las del otro, sin orden. Aunque los hilos tengan su propia matriz y cálculo, ambos comparten el flujo de salida, que no está sincronizado por defecto.

Código

```
package ejercicio1;

import java.util.Random;

/**

* Clase para generar una matriz cuadrada de tamaño fijo y realizar operaciones con
ella.
```

```
*/
class Matriz {
      public static final int TAMANO_MATRIZ = 3; //Tamaño matriz
      public static final int NUM_RANDOM = 10;
  private int[][] matriz = new int[TAMANO_MATRIZ][TAMANO_MATRIZ]; //Matriz principal
  private Random random = new Random();    //Generar valores aleatorios
   /**
   * Rellena la matriz con valores aleatorios entre 0 y 9.
   */
  public void generar() {
      for (int i = 0; i < TAMANO_MATRIZ; i++)
          for (int j = 0; j < TAMANO_MATRIZ; j++)
              matriz[i][j] = random.nextInt(NUM_RANDOM); // números entre 0 y 9
  }
   /**
   * Devuelve la matriz generada.
   *
   * @return Matriz generada.
   */
  public int[][] getMatriz() {
      return matriz;
  }
   /**
   * Suma una matriz consigo misma.
   *
   * @param A Matriz a sumar.
   * @return Resultado de la suma A + A.
   */
  public static int[][] sumar(int[][] A) {
```

```
    int[][] resultado = new int[TAMANO_MATRIZ][TAMANO_MATRIZ];
    for (int i = 0; i < TAMANO_MATRIZ; i++)
        for (int j = 0; j < TAMANO_MATRIZ; j++)
            resultado[i][j] = A[i][j] + A[i][j];
    return resultado;
}
/**
 * Multiplica una matriz consigo misma (A × A).
 *
 * @param A Matriz a multiplicar.
 * @return Resultado de la multiplicación A × A.
 */
public static int[][] multiplicar(int[][] A) {
    int[][] resultado = new int[TAMANO_MATRIZ][TAMANO_MATRIZ];
    for (int i = 0; i < TAMANO_MATRIZ; i++)
        for (int j = 0; j < TAMANO_MATRIZ; j++)
            for (int k = 0; k < TAMANO_MATRIZ; k++)
                resultado[i][j] += A[i][k] * A[k][j];
    return resultado;
}
/**
 * Imprime una matriz con título.
 *
 * @param titulo Título a mostrar antes de la matriz.
 * @param A Matriz a imprimir.
 */
public static void imprimir(String titulo, int[][] A) {
    System.out.println(titulo);
    for (int[] fila : A) {
```

```
            for (int valor : fila) {
                System.out.printf("%4d", valor);
            }
            System.out.println();
        }
        System.out.println();
    }
}
package ejercicio1;
import java.util.concurrent.locks.ReentrantLock;
/**
* Hilo que genera una matriz aleatoria y la multiplica consigo misma en varias iteraciones.
* Sincroniza la impresión de resultados usando un ReentrantLock.
*/
class HiloMultiplicador extends Thread {
        public static final int NUM_ITERACIONES = 10;     //número de repeticiones
    private final ReentrantLock lock;  //Lock para sincronizar la salida por pantalla
    /**
     * Constructor.
     *
      * @param lock Objeto ReentrantLock compartido para sincronizar la salida por pantalla.
     */
    public HiloMultiplicador(ReentrantLock lock) {
        this.lock = lock;
    }
    @Override
    public void run() {
        for (int iter = 0; iter < NUM_ITERACIONES ; iter++) { //10 iteraciones
```

```
            //Generamos la matriz
      Matriz m = new Matriz();
            m.generar();
            int[][] A = m.getMatriz(); //Obtenemos la matriz generada
            int[][] resultado = Matriz.multiplicar(A);   //Multiplicar matriz por sí
misma
            lock.lock();   //Bloquear acceso a pantalla
            try {
                  System.out.println("Iteración " + (iter + 1) + " - Hilo Multiplicador
(A × A):");
                Matriz.imprimir("Matriz A:", A);
                Matriz.imprimir("Resultado A × A:", resultado);
            } finally {
                lock.unlock();   //Liberar bloqueo
            }
      }
  }
}
package ejercicio1;
import java.util.concurrent.locks.ReentrantLock;
/**
* Hilo que genera una matriz aleatoria y la suma consigo misma en varias
iteraciones.
* Utiliza un ReentrantLock para imprimir sin interferencia de otros hilos.
*/
class HiloSumador extends Thread {
  private final ReentrantLock lock;        //Sincronizar acceso a pantalla
      public static final int NUM_ITERACIONES = 10;
      /**
    * Constructor.
    *
```

```
    * @param lock ReentrantLock compartido para sincronización de pantalla.
    */
  public HiloSumador(ReentrantLock lock) {
      this.lock = lock;
  }
  @Override
  public void run() {
      for (int iter = 0; iter < NUM_ITERACIONES; iter++) {
      //Generamos matriz
          Matriz m = new Matriz();
          m.generar();
          int[][] A = m.getMatriz(); //Obtenemos la matriz generada
          int[][] resultado = Matriz.sumar(A);
          lock.lock();   //Bloquear acceso a pantalla
          try {
              System.out.println("Iteración "+(iter + 1) +"-Hilo Sumador (A+A):");
              Matriz.imprimir("Matriz A:", A);
              Matriz.imprimir("Resultado A + A:", resultado);
          } finally {
              lock.unlock();   //Liberar bloqueo
          }
      }
  }
}
package ejercicio1;
import java.util.concurrent.locks.ReentrantLock;
/**
* Clase principal que ejecuta dos hilos en paralelo para realizar operaciones con
matrices:
* uno para suma y otro para multiplicación.
```

```
*
* Usa concurrencia segura mediante ReentrantLock para evitar solapamiento en la consola.
*/
public class Ejercicio1 {
       /**
    * Punto de entrada del programa. Inicia los hilos y espera su finalización.
    *
    * @param args Argumentos de línea de comandos (no utilizados).
    */
  public static void main(String[] args) {
      //Lock para la sincronización entre hilos
      ReentrantLock lock = new ReentrantLock();
      //Creación de hilos, uno para sumar y otro para multiplicar
      Thread hilo1 = new HiloMultiplicador(lock);
      Thread hilo2 = new HiloSumador(lock);

      // Iniciar los hilos
      hilo1.start();
      hilo2.start();
      }
}
```

Ejercicio 2: Semáforos

Recursos no compartibles

En este ejercicio, los recursos no compartibles serían cada uno de los paneles debido a que estos solo pueden ser usados por un hilo a la vez.

Condiciones de sincronización

Cada hilo tiene que esperar que uno de los paneles esté disponible para poder escribir en él.

Pseudocódigo

```
//Variables globales
const NUM_PANELES = 3
const NUM_HILOS = 10
const NUM_ITERACIONES = 3
const NUM_FILAS = 10

Semaphore S[1..NUM_PANELES]    //Un semaforo por panel
Panel paneles[1..NUM_PANELES]

// Inicialización
para i := 1 hasta NUM_PANELES hacer
   initial(S[i], 1)
   paneles[i] := nuevo Panel("panel"+i, (i-1)*500, 0)
fin para
//Programa principal
HiloSuma[1..NUM_HILOS] hilos
begin
      para i := 1 hasta NUM_HILOS hacer
         hilos(i) := HiloSuma(i)
         start(hilos(i))                    //Inicializamos los hilos
      fin para
end
```

```
//Proceso HiloSuma
proceso HiloSuma(i: entero)
begin
   repetir NUM_ITERACIONES veces
      matriz A := generar()              //Hacemos las operaciones
      matriz B := generar()
      matriz C := sumaMatrices(A, B)

      mostrado := falso
      mientras no mostrado hacer
         para j := 1 hasta NUM_PANELES hacer
            si S[j].valor != 0 entonces //Cuando haya un panel libre, lo usamos para sacar el resultado
               wait(S[j]);
               paneles[j].escribir("Hilo: " + i)
               paneles[j].escribir("Matriz A:")
               para k := 1 hasta NUM_FILAS hacer
                  paneles[j].escribir(linea(A, k))
               fin para
               paneles[j].escribir("Matriz B:")
               para k := 1 hasta NUM_FILAS hacer
                  paneles[j].escribir(linea(B, k))
               fin para
               paneles[j].escribir("Matriz C (A+B):")
               para k := 1 hasta NUM_FILAS hacer
                  paneles[j].escribir(linea(C, k))
```

```
                fin para
                mostrado := verdadero
                signal(S[j])
                salir del para
            fin si
        fin para
    fin mientras
  fin repetir
fin
//Funcion para sumar 2 matrices diferentes
función sumaMatrices(A, B)
  para i := 1 hasta NUM_FILAS hacer
    para j := 1 hasta NUM_FILAS hacer
      C[i][j] := A[i][j] + B[i][j]       //Sumamos las matrices
    fin para
  fin para
  devolver C
fin
```

Cuestiones planteadas

a) Este problema recuerda al problema de los filósofos, por el número limitado de recursos (los 3 paneles) y por la necesidad de exclusión mutua (sólo un hilo puede acceder a un panel a la vez).

b) Si solo hay un panel, el acceso es secuencial, esto crea una cola de espera y si hay muchos hilos puede que la espera sea muy larga.

   En el caso de que hubiese tres paneles, hasta 3 hilos podrían usar simultáneamente un panel cada uno. Esto reduce el tiempo de espera y mejora la eficiencia y el paralelismo.

c) Para resolver este problema hemos usado tres semáforos binarios cada uno indicando si se puede o no se puede usar determinado panel.

Código

```
package ejercicio2;

import java.util.concurrent.Semaphore;

/**

* Clase principal que inicia la ejecución del programa.

* Se crean 3 paneles y semáforos, y se inician 10 hilos que operan sobre ellos.

*/

public class Ejercicio2 {

   /** Número de paneles utilizados en la aplicación. */

   public static int NUM_PANELES = 3;

   /** Número total de hilos que se ejecutarán. */

   public static int NUM_HILOS = 10;

   /**

    * Método principal que configura y lanza los hilos con sus respectivos paneles y
semáforos.

    *

       * @param args Argumentos de línea de comandos (no utilizados en esta
aplicación).

    */

   public static void main(String[] args) {


       // Inicialización de Paneles

       Panel pan1 = new Panel("panel1", 0, 0);

       Panel pan2 = new Panel("panel2", 500, 0);

       Panel pan3 = new Panel("panel3", 1000, 0);

       Panel[] paneles = new Panel[NUM_PANELES];

       paneles[0] = pan1;

       paneles[1] = pan2;
```

```
        paneles[2] = pan3;

        // Inicialización de Semáforos (uno por panel)
        Semaphore s1 = new Semaphore(1);
        Semaphore s2 = new Semaphore(1);
        Semaphore s3 = new Semaphore(1);
        Semaphore[] semaforos = new Semaphore[NUM_PANELES];
        semaforos[0] = s1;
        semaforos[1] = s2;
        semaforos[2] = s3;

        // Inicialización y ejecución de hilos
        HiloSuma[] hilos = new HiloSuma[NUM_HILOS];

        for (int i = 0; i < NUM_HILOS; i++) {
            hilos[i] = new HiloSuma(i, semaforos, paneles);
            hilos[i].start();
        }
    }
}
package ejercicio2;
import java.util.concurrent.Semaphore;
/**
* Clase que representa un hilo que realiza la suma de dos matrices y muestra el resultado
* en uno de los paneles disponibles, utilizando semáforos para sincronizar el acceso.
*/
public class HiloSuma extends Thread {
```

```
   /** Número de iteraciones que realiza cada hilo. */
   public static final int NUM_ITERACIONES = 3;
   /** Número de filas que contienen las matrices. */
   public static final int NUM_FILAS = 10;


     /** Arreglo de semáforos, uno por cada panel, para controlar el acceso concurrente. */
   private Semaphore[] semaforos;
   /** Arreglo de paneles donde se mostrarán los resultados. */
   private Panel[] paneles;
   /** Identificador del hilo. */
   private final int id;
   /**
    * Constructor de la clase HiloSuma.
    *
    * @param i Identificador del hilo.
    * @param s Arreglo de semáforos correspondientes a los paneles.
    * @param p Arreglo de paneles disponibles para mostrar resultados.
    */
   public HiloSuma(int i, Semaphore[] s, Panel[] p) {
       this.semaforos = s;
       this.paneles = p;
       this.id = i;
   }
   /**
     * Método que ejecuta el hilo. Cada hilo realiza varias iteraciones en las que genera dos matrices,
    * las suma, y muestra el resultado en uno de los paneles disponibles, asegurando exclusión mutua
    * mediante semáforos.
    */
```

```
    @Override
    public void run() {
        for (int iter = 0; iter < NUM_ITERACIONES; iter++) {
            // Generamos las matrices A y B con valores aleatorios
            Matriz2 a = new Matriz2();
            Matriz2 b = new Matriz2();
            a.generar();
            b.generar();
            int[][] A = a.getMatriz();
            int[][] B = b.getMatriz();
            // Realizamos la suma de matrices A y B, almacenando el resultado en C
            Matriz2 resultado = new Matriz2(Matriz2.sumar(A, B));
            boolean mostrado = false;
            // Intentamos mostrar el resultado en uno de los paneles disponibles
            while (!mostrado) {
                for (int i = 0; i < paneles.length; i++) {
                      if (semaforos[i].tryAcquire()) { // Intenta adquirir el semáforo
sin bloquear
                        try {
                              // Escribimos en el panel la información generada por el
hilo
                            paneles[i].escribir_mensaje("Hilo: " + id);
                            paneles[i].escribir_mensaje("Matriz A:");
                            for (int j = 0; j < NUM_FILAS; j++) {
                                paneles[i].escribir_mensaje(a.getLinea(j));
                            }
                            paneles[i].escribir_mensaje("Matriz B:");
                            for (int j = 0; j < NUM_FILAS; j++) {
                                paneles[i].escribir_mensaje(b.getLinea(j));
                            }
```

```
                            paneles[i].escribir_mensaje("Matriz C (A+B):");
                            for (int j = 0; j < NUM_FILAS; j++) {
                                paneles[i].escribir_mensaje(resultado.getLinea(j));
                            }
                            mostrado = true;
                            break;
                        } finally {
                            // Liberamos el semáforo tras usar el panel
                            semaforos[i].release();
                        }
                    }
                }
            }
        }
    }
}
package ejercicio2;
import java.util.Random;
/**
* Clase que representa una matriz cuadrada de enteros y proporciona métodos para
* generar valores aleatorios, sumar matrices, imprimirlas y obtener líneas como texto.
*/
class Matriz2 {
    /** Tamaño de la matriz (número de filas y columnas). */
    public static final int TAMANO_MATRIZ = 10;
    /** Límite para los valores aleatorios generados. */
    public static final int NUM_RANDOM = 10;
    /** Matriz de enteros representada como arreglo bidimensional. */
    private int[][] matriz = new int[TAMANO_MATRIZ][TAMANO_MATRIZ];
```

```
/** Semilla de números aleatorios. */
private Random random = new Random();
/**
 * Constructor por defecto. Inicializa la matriz sin valor.
 */
public Matriz2() {

}
/**
 * Constructor que recibe una matriz preexistente para asignarla internamente.
 *
 * @param a Matriz de enteros que se utilizará como base.
 */
public Matriz2(int[][] a) {
    this.matriz = a;
}
/**
 * Genera números aleatorios para llenar la matriz.
 */
public void generar() {
    for (int i = 0; i < TAMANO_MATRIZ; i++) {
        for (int j = 0; j < TAMANO_MATRIZ; j++) {
            matriz[i][j] = random.nextInt(NUM_RANDOM);
        }
    }
}
/**
 * Retorna la matriz actual.
 *
```

```
     * @return Matriz de enteros.
     */
    public int[][] getMatriz() {
        return matriz;
    }
    /**
     * Realiza la suma de dos matrices A y B.
     *
     * @param A Primera matriz de enteros.
     * @param B Segunda matriz de enteros.
     * @return Nueva matriz con la suma de A y B.
     */
    public static int[][] sumar(int[][] A, int[][] B) {
        int[][] resultado = new int[TAMANO_MATRIZ][TAMANO_MATRIZ];
        for (int i = 0; i < TAMANO_MATRIZ; i++) {
            for (int j = 0; j < TAMANO_MATRIZ; j++) {
                resultado[i][j] = A[i][j] + B[i][j];
            }
        }
        return resultado;
    }
    /**
      * Devuelve una línea de la matriz como una cadena de texto con los valores
separados por espacios.
     *
     * @param l Índice de la fila deseada (0 ≤ l < TAMANO_MATRIZ).
     * @return Cadena de texto con los valores de la fila separados por espacios.
     */
    public String getLinea(int l) {
        StringBuffer str = new StringBuffer();
```

```
        for (int i = 0; i < TAMANO_MATRIZ; i++) {
            str.append(matriz[l][i] + " ");
        }
        return str.toString();
    }
}
```

Ejercicio 3: Monitores

Recursos no compartibles

En este caso, los recursos no compartibles son la pantalla, ya que solo un hilo cliente puede imprimir a la vez. Así como el monitor, que es un objeto para pedir acceso a las máquinas. Por otro lado, está el mutexPantalla, un objeto para pedir acceso a la pantalla. Por último, el lavadero, objeto que controla el acceso a la zona de lavado.

Condiciones de sincronización

Las condiciones de sincronización serían las máquinas, ya que un cliente no puede usar ninguna hasta que haya alguna disponible y los lavaderos, debido a que los clientes también tienen que esperar a que quede alguno libre.

Pseudocódigo

//Variables globales

const NUM_CLIENTES = 50

const NUM_LAVADEROS = 4

const NUM_MAQUINAS = 3

Lavadero lavadero //Gestiona los lavaderos

Monitor monitor //Gestiona las maquinas

MutexPantalla mutex //Gestiona el acceso a la pantalla

//Inicialización

HiloCliente[0..NUM_CLIENTES] clientes

```
begin
   para i := 0 hasta NUM_CLIENTES hacer
      clientes(i) := HiloCliente(i)
      start(clientes(i))  //Inicializamos los hilos y los iniciamos
   fin para
end
//Proceso HiloCliente
proceso HiloCliente(i: entero)
int tiempoServicio := random(500, 1000)
int tiempoLavado := random(1000, 2000)
begin
        int maquina = monitor.pedirMaquina() //Solicitamos una maquina
        esperar(tiempoServicio)
        mutex.pedirAcceso()             //Pedimos acceso a pantalla
        maquina := maquina + 1          //Imprimimos la info de la maquina
        print("Cliente " + i + " ha solicitado su servicio en la máquina: " + maquina)
        print("Tiempo en solicitar el servicio: " + tiempoServicio)
        idLavadero := lavadero.seleccionarCola()     //Miramos en que lavadero nos ponemos
        print("El coche será lavado en el lavadero nº: " + idLavadero)        //Info del lavadero
        print("Tiempo en lavar el coche: " + tiempoLavado)
        print("Tiempo de espera en la ")
        para j := 0 hasta 3 hacer         //Info de las colas
                print("lavadero" + j + " = " + tiempoCola[j])
        fin para
        print("\n")
        mutex.liberarAcceso()           //Liberamos la pantalla
```

```
        lavadero.añadirTiempoLavado(tiempoLavado)
        esperar(tiempoLavado)
        lavadero.liberarLavadero(idLavadero)
end
//Clase Lavadero
l1 := lock
c1 := condition          //Una condición por cada lavadero
c2 := condition
c3 := condition
c4 := condition
colas := int[NUM_LAVADEROS]
libre := boolean[NUM_LAVADEROS]
timpoColas := Mapa<entero, entero>
arrayColas := {c1, c2, c3, c4}
funcion colaMasVacia()
        l1.lock()
        masPequeña := INFINITO
        idMasPequeña := 0
        para entero idCola in tiempoColas.llaces()
                actual := tempoColas.get(idCola) || 0          //Buscamos la cola con menos tiempo de
                si actual < masPequeña entonces                //espera y la devolvemos
                        idMasPequeña := idCola
                        masPequeña := actual
                fin si
        fin para
        l1.unlock()
```

```
        devuelve idMasPequeña


función seleccionarCola()
        l1.lock()
        id :=  -1
        para i in 0..NUM_COLAS
                si(libre[i]) entonces id := i fin si        //Si hay una libre seleccionamos esa
        fin para
        si id == -1 entonces id = colaMasVacia()      //Si no hay ninguna libre la que tenga menos
        l1.unlock()                                   //tiempo de espera
        devuelve id
procedimiento añadirTiempoLavado(entero: id, tiempoLavado)
        l1.lock()
        si NOT libre[id] entonces
                colas[id] += 1                  //Añadimos la cola
                actual = tiempoColas.get(id) || 0
                actual += tiempoLavado        //Sumamos el tiempo de lavado
                tiempoColas.put(id, actual)
        fin si
        mientras NOT libre[id]
                esperar(arrayColas[id])
        fin mientras
        libre[id] := false
        actual := tiempoColas.get(id) || 0
        si actual != 0 entonces
                actual -= tiempoLavado        //Cuando acaba el lavado lo restamos
```

```
        l1.unlock()
procedimiento liberarLavadero(entero: id)
        l1.lock()
        libre[id] := true
        signalAll(arrayColas[id])
        l1.unlock()
//Clase Monitor
ocupadas := entero
libre := boolean[NUM_MAQUINAS]          //Indica que colas estan libres
l := lock
infolibre := condition
funcion pedirMaquina
        l.lock()
        mientras(ocupadas == 3)             //Si estan todas ocupadas esperamos
                wait(nolibre)
        fin mientras
        id := 0
        mientras NOT libre[id]              //Si no miramos cual es la que esta libre
                id += 1
        fin mientras
        libre[id] = false                   //La ocupamos
        ocupadas += 1
        l.unlock()
        devuelve id
procedimiento liberarMaquina(entero: id)
        l.lock()
```

```
        libre[id] = true          //Liberamos la cola
        ocupadas -= 1
        signalAll(nolibre)
        l.unlock()
//Clase Mutex Pantalla
ocupada := false
l1 := lock
c := condition
procedimiento pedirAcceso()
        l1.lock()
        mientras ocupada       //Si esta ocupada esperamos
                wait(c)
        fin mientras
        ocupada = true         //Cuano lo deje de estar la ocupamos
        l1.unlock()
procedimiento liberarAcceso()
        l1.lock()
        ocupada := false       //La desocupamos
        signal(c)
        l1.unlock()
```

Cuestiones planteadas

a) La acción de lavar un coche es concurrente, ya que puede haber 4 hilos cliente lavando su coche uno en cada lavadero al mismo tiempo.

b) Es un monitor implementado con ReentrantLock y Condition. Ponemos una condición por cada una de las 4 colas. En cada procedimiento, el lock impide que otros hilos accedan a la sección crítica. Además, con las 4 condiciones, en el momento en el que alguien termina de usar un lavadero se avisa a todos los otros hilos que están en esa cola para que vean si les toca.

c) Se ha utilizado notifyAll/signalAll, ya que no hay problema en avisar a todos los clientes, así, cuando una máquina o lavadero queda libre, todos los hiloCliente que estén esperando despiertan para usar el recurso. No se podría usar en caso de que solo hubiera que avisar a ciertos clientes de que una máquina/lavadero está libre.

d) Mediante un monitor específico para la pantalla llamado MutexPantalla, usando un ReentrantLock y una Condition. Haciendo que no más de un hilo pueda usar la consola a la vez.

Código

```
package ejercicio3;

/**

* Clase principal que simula la llegada de múltiples clientes a un lavadero de coches.

*/

public class Ejercicio3 {

  /** Número total de clientes que serán creados y ejecutados como hilos. */

  public static final int NUM_CLIENTES = 50;

  /**

   * Método principal que inicializa los monitores y lanza los hilos clientes.

   *

     * @param args Argumentos de la línea de comandos (no utilizados en esta aplicación).

   */

  public static void main(String[] args) {

      // Inicialización de monitores

      Monitor monitorMaquinas = new Monitor();

      MutexPantalla monitorPantalla = new MutexPantalla();

      Lavadero monitorLavadero = new Lavadero();
```

```
      // Creación y ejecución de hilos clientes
      HiloCliente[] clientes = new HiloCliente[NUM_CLIENTES];
      for (int i = 0; i < NUM_CLIENTES; i++) {
                  clientes[i] = new HiloCliente(i, monitorMaquinas, monitorPantalla,
monitorLavadero);
          clientes[i].start();
      }
  }
}
package ejercicio3;
import java.util.Random;
/**
* Clase que representa un cliente que solicita un servicio de máquina y lavado en un
lavadero.
* Cada cliente se ejecuta como un hilo independiente.
*/
public class HiloCliente extends Thread {
  /** Tiempo que el cliente usará la máquina de autoservicio (en milisegundos). */
  private int tiempoServicio;
  /** Tiempo que el coche del cliente estará en el lavadero (en milisegundos). */
  private int tiempoLavado;
  /** Identificador único del cliente. */
  private int id;
  /** Monitor compartido que gestiona las máquinas disponibles. */
  private Monitor monitor;
  /** Monitor para solicitar escribir en pantalla. */
  private MutexPantalla mutex;
  /** Lavadero compartido donde los coches serán lavados en colas independientes. */
  private Lavadero lavadero;
  /** Número total de colas del lavadero. */
```

```
  private final static int NUMERO_COLAS = 4;

  /**
      * Constructor que inicializa los datos del cliente y sus tiempos de
espera/lavado.
   *
   * @param id Identificador del cliente.
   * @param monitor Referencia al monitor que controla las máquinas.
   * @param mutex Referencia al monitor de acceso a la pantalla.
   * @param lav Lavadero donde se procesarán los lavados.
   */
  public HiloCliente(int id, Monitor monitor, MutexPantalla mutex, Lavadero lav) {
      this.id = id;
      this.monitor = monitor;
      this.tiempoLavado = tiempoLavado();
      this.tiempoServicio = tiempoMaquina();
      this.mutex = mutex;
      this.lavadero = lav;
  }
  /**
   * Genera un tiempo aleatorio para el lavado del coche.
   *
   * @return Tiempo de lavado en milisegundos (entre 1000 y 2000).
   */
  public int tiempoLavado() {
      int min = 1000;
      int max = 2000;
      Random random = new Random();
      return random.nextInt(max - min + 1) + min;
  }
  /**
```

```
    * Genera un tiempo aleatorio para el uso de la máquina de autoservicio.
    *
    * @return Tiempo de servicio en milisegundos (entre 500 y 1000).
    */
   public int tiempoMaquina() {
       int min = 500;
       int max = 1000;
       Random random = new Random();
       return random.nextInt(max - min + 1) + min;
   }
   /**
    * Método que define el comportamiento del hilo cliente:
    * pide una máquina, simula el uso, selecciona una cola de lavado, espera su turno
    * y finalmente libera el lavadero.
    */
   @Override
   public void run() {
       try {
           // Solicitar una máquina del monitor
           int maquina = this.monitor.pedirMaquina();
           // Simular uso de la máquina
           Thread.sleep(this.tiempoServicio);
           // Liberar la máquina
           this.monitor.liberarMaquina(maquina);
           // Mostrar información en pantalla (protegido por mutex)
           mutex.pedirAcceso();
              System.out.println("Cliente " + id + " ha solicitado su servicio en la
máquina: " + (maquina + 1));
                      System.out.println("Tiempo en solicitar el servicio: " +
this.tiempoServicio);
```

```
            // Seleccionar el lavadero más conveniente
            int idLavadero = lavadero.seleccionarCola();
                  System.out.println("El coche será lavado en el lavadero n°: " +
idLavadero);
            System.out.println("Tiempo en lavar el coche: " + this.tiempoLavado);
            mutex.liberarAcceso();
            // Mostrar tiempos de espera en todas las colas
            mutex.pedirAcceso();
            System.out.print("Tiempo de espera en la ");
            for (int i = 0; i < NUMERO_COLAS; i++) {
                  System.out.print("lavadero" + i + " = " + lavadero.tiempoCola(i) + "
");
            }
            System.out.println("\n");
            mutex.liberarAcceso();
            // Añadir tiempo de lavado a la cola seleccionada
            lavadero.anadirTiempoLavadero(idLavadero, this.tiempoLavado);
            // Simular el tiempo de lavado
            Thread.sleep(tiempoLavado);
            // Liberar el lavadero
            lavadero.liberarLavadero(idLavadero);
      } catch (InterruptedException e) {
            e.printStackTrace();
      }
  }
}
package ejercicio3;
import java.util.ArrayList;
import java.util.Collections;
import java.util.TreeMap;
```

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

import java.util.concurrent.locks.ReentrantLock;

/**

* Clase que simula un sistema de lavado de coches con 4 lavaderos.

* Cada lavadero tiene una cola controlada mediante condiciones y un control de tiempos acumulados.

* Se usa programación concurrente con ReentrantLock y Condition.

*/

public class Lavadero {

  /** Candado para proteger el acceso concurrente a los lavaderos. */

  private ReentrantLock l1 = new ReentrantLock();

  /** Condiciones asociadas a cada cola del lavadero. */

  private Condition cola1 = l1.newCondition();

  private Condition cola2 = l1.newCondition();

  private Condition cola3 = l1.newCondition();

  private Condition cola4 = l1.newCondition();

  /** Arreglo que almacena el número de clientes esperando en cada cola. */

  private int[] colas;

  /** Estado de disponibilidad de cada lavadero. */

  private boolean[] libre;

  /** Lista que contiene las condiciones para cada cola. */

  private ArrayList<Condition> arrayColas;

  /**

   * Mapa que almacena el tiempo total de espera acumulado por cola.

   * La clave es el ID del lavadero y el valor es el tiempo.

   */

  private TreeMap<Integer, Integer> tiempoColas;

  /** Número total de lavaderos disponibles. */

  private final static int NUM_LAVADEROS = 4;

  /**
```

```
 * Devuelve el tiempo de espera acumulado en la cola de un lavadero dado.
 *
 * @param id Identificador del lavadero.
 * @return Tiempo total de espera acumulado en milisegundos.
 */
public int tiempoCola(int id) {
    return this.tiempoColas.getOrDefault(id, 0);
}
/**
 * Constructor de la clase Lavadero.
 * Inicializa las colas, condiciones, estados de disponibilidad y tiempos.
 */
public Lavadero() {
    colas = new int[NUM_LAVADEROS];
    arrayColas = new ArrayList<>();
    Collections.addAll(arrayColas, cola1, cola2, cola3, cola4);
    // Inicializar colas y lavaderos como libres
    for (int i = 0; i < NUM_LAVADEROS; i++) {
        colas[i] = 0;
    }
    libre = new boolean[NUM_LAVADEROS];
    for (int i = 0; i < NUM_LAVADEROS; i++) {
        libre[i] = true;
    }
    tiempoColas = new TreeMap<>();
}
/**
 * Determina cuál es la cola con menor tiempo de espera acumulado.
 *
```

```
     * @return El índice de la cola más vacía.
     */
    private int colaConMenosTiempo() {
        l1.lock();
        int masPequeña = Integer.MAX_VALUE;
        int idColaMasPequeña = 0;
        for (int idcola : this.tiempoColas.keySet()) {
            int actual = tiempoColas.getOrDefault(idcola, 0);
            if (actual < masPequeña) {
                idColaMasPequeña = idcola;
                masPequeña = actual;
            }
        }
        l1.unlock();
        return idColaMasPequeña;
    }
    /**
     * Selecciona el lavadero más adecuado para el siguiente cliente.
     * Si hay algún lavadero libre, se elige ese.
     * En caso contrario, se elige la cola con menor tiempo de espera.
     *
     * @return El ID del lavadero seleccionado.
     */
    public int seleccionarCola() {
        l1.lock();
        int id = -1;
        for (int i = 0; i < this.libre.length; i++) {
            if (libre[i]) {
                id = i; // se mete en el último libre encontrado
```

```
            }
        }
        if (id == -1) {
            id = colaConMenosTiempo();
        }
        l1.unlock();
        return id;
    }
    /**
     * Añade el tiempo de lavado a la cola del lavadero y espera su turno si no está libre.
     *
     * @param id ID del lavadero.
     * @param tiempoLavado Tiempo de lavado en milisegundos.
     */
    public void anadirTiempoLavadero(int id, int tiempoLavado) {
        l1.lock();
        // Si el lavadero está ocupado, se suma el tiempo a la cola
        if (!libre[id]) {
            colas[id]++;
            int actual = this.tiempoColas.getOrDefault(id, 0);
            actual += tiempoLavado;
            this.tiempoColas.put(id, actual);
        }
        // Esperar a que el lavadero esté libre
        while (!libre[id]) {
            try {
                arrayColas.get(id).await();
            } catch (InterruptedException e) {
                e.printStackTrace();
```

```
            }
        }
        // Una vez liberado, continuar
        if (colas[id] != 0) {
            colas[id]--; // se retira de la cola
        }
        libre[id] = false;
        // Actualizar tiempo restante de espera
        int actual = this.tiempoColas.getOrDefault(id, 0);
        if (actual != 0) {
            actual -= tiempoLavado;
        }
        this.tiempoColas.put(id, actual);
        l1.unlock();
    }
    /**
     * Libera un lavadero y notifica a los hilos que están esperando en su cola.
     *
     * @param id ID del lavadero que se libera.
     */
    public void liberarLavadero(int id) {
        l1.lock();
        libre[id] = true;
        arrayColas.get(id).signalAll(); // notificar a todos los clientes en espera
        l1.unlock();
    }
}
package ejercicio3;
import java.util.concurrent.locks.*;
```

```
/**
* Monitor que gestiona el acceso concurrente a un conjunto de máquinas de servicio.
* Solo hay 3 máquinas, y los hilos deben esperar si todas están ocupadas.
* Utiliza ReentrantLock y Condition.
*/
public class Monitor {
  /** Número de máquinas actualmente ocupadas. */
  private int ocupadas;
  /** Arreglo que indica si cada máquina está libre (true) u ocupada (false). */
  private boolean[] libre;
  /** Lock para proteger el acceso concurrente al monitor. */
  private ReentrantLock l = new ReentrantLock();
  /** Condición utilizada para esperar si no hay máquinas libres. */
  private Condition nolibre = l.newCondition();
  /**
   * Constructor del monitor. Inicializa las 3 máquinas como libres.
   */
  public Monitor() {
      libre = new boolean[3];
      libre[0] = true;
      libre[1] = true;
      libre[2] = true;
      ocupadas = 0;
  }
  /**
   * Método que permite a un hilo solicitar una máquina.
   * Si todas están ocupadas, el hilo espera hasta que una esté disponible.
   *
   * @return El ID de la máquina asignada (0, 1 o 2).
```

```
     * @throws InterruptedException si el hilo es interrumpido mientras espera.
     */
    public int pedirMaquina() throws InterruptedException {
        l.lock();
        try {
            // Esperar si todas las máquinas están ocupadas
            while (ocupadas == 3) {
                nolibre.await();
            }
            // Buscar una máquina libre
            int id = 0;
            while (!libre[id]) {
                id++;
            }
            libre[id] = false; // Marcar como ocupada
            ocupadas++;
            return id;
        } finally {
            l.unlock();
        }
    }
    /**
     * Libera la máquina con el ID especificado y notifica a los hilos en espera.
     *
     * @param id ID de la máquina a liberar (0, 1 o 2).
     */
    public void liberarMaquina(int id) {
        l.lock();
        try {
```

```
            libre[id] = true;
            ocupadas--;
            nolibre.signalAll(); // Notificar a los hilos en espera
        } finally {
            l.unlock();
        }
    }
}
package ejercicio3;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;
/**
* MutexPantalla es un monitor que implementa exclusión mutua
* para el acceso a la pantalla con ReentrantLock y Condition.
*/
public class MutexPantalla {
    /** Indica si el recurso está ocupado por algún hilo. */
    private boolean ocupada;
    /** Lock utilizado para garantizar la exclusión mutua. */
    private ReentrantLock l1 = new ReentrantLock();
    /** Condición asociada al lock para esperar cuando el recurso está ocupado. */
    private Condition c = l1.newCondition();
    /**
     * Inicializa el estado del recurso como libre.
     */
    public MutexPantalla() {
        ocupada = false;
    }
    /**
```

```
 * Solicita acceso exclusivo al recurso.
 * Si ya está siendo utilizado por otro hilo, espera hasta que esté libre.
 */
public void pedirAcceso() {
    l1.lock();
    try {
        while (ocupada) {
            try {
                c.await();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        ocupada = true;
    } finally {
        l1.unlock();
    }
}
/**
 * Libera el acceso exclusivo al recurso y notifica a los hilos en espera.
 */
public void liberarAcceso() {
    l1.lock();
    try {
        ocupada = false;
        c.signal();  // Despierta a un hilo que esté esperando acceso
    } finally {
        l1.unlock();
    }
```

```
    }
}
```

Ejercicio 4: Paso de mensajes

Recursos no compartibles

En este problema, los recursos no compartibles son las colas para pagar y la pantalla por la que se muestran los mensajes. Las colas son de uso exclusivo para cada Persona y la pantalla es de uso exclusivo para el Controlador.

Condiciones de sincronización

Las condiciones de sincronización son el acceso a las colas ya que las personas compiten para acceder a ellas. Además el controlador gestiona el acceso a las colas, asegurándose de que cada persona sea asignada a una caja libre.

Por otro lado, después de que una persona completa el pago, la cola debe ser liberada para que otra persona pueda usarla. El controlador libera las cajas una vez que una persona ha completado su pago.

Asimismo, el controlador imprime información en la pantalla. La sincronización asegura que solo un hilo a la vez pueda imprimir en la pantalla para evitar problemas de concurrencia.

Pseudocódigo

```
process type Persona(id: integer;
                     enviar, respuesta, colaR, colaL, liberar, imprimir: mailbox of string);
const
    PAGOS = 5;
    NUM_RANDOM = 10;
var
    tiempoCompra, tiempoPago: integer;
    asignada: string;
    info: string;
begin
```

```
for i := 1 to PAGOS do
begin
   // 1. Simula tiempo de compra
   tiempoCompra := random(NUM_RANDOM) + 1;
   wait(tiempoCompra * 100);
   // 2. Solicita al controlador
   send(enviar, toString(id));
   // Recibe tiempoPago y cola asignada (R o L)
   receive(respuesta, texto);
   dividir texto en partes por ","
   tiempoPago := toInteger(partes[0]);
   asignada := partes[1];
   //  Entra en la cola correspondiente
   if asignada = "R" then
      send(colaR, toString(id))
   else
      send(colaL, toString(id));
  end;
   //  Espera confirmación para pagar
   receive(respuesta, _);
   // 3. Simula el pago
   wait(tiempoPago * 100);
   // 4. Libera la cola usada
   send(liberar, asignada);
   // 5. Envía mensaje para impresión
   info := "Persona " + toString(id) +
```

```
            " usó cola " + asignada +
            " durante " + toString(tiempoPago * 100);
        send(imprimir, info);
    end;
end;
process type Controlador(enviar, colaR, colaL, liberar, imprimir: mailbox of string;
                         buzonesPersonas: array[1..N] of mailbox of string);
var
    colaRLibre, colaLLibre: boolean := true;
    activo: boolean := true;
    id, respuesta, mensaje, cola: string;
    tiempoPago: integer;
    selector: selector;
begin
    agregar al selector: enviar, colaR, colaL, liberar, imprimir
    while activo do
    begin
        setGuard(colaR, colaRLibre);
        setGuard(colaL, colaLLibre);
       select
            // 1. Solicitud de cola
            receive(enviar, id);
            tiempoPago := random(10) + 1;
            if tiempoPago >= 5 then
                respuesta := toString(tiempoPago) + ",R"
            else
```

```
            respuesta := toString(tiempoPago) + ",L";
        send(buzonesPersonas[toInteger(id)], respuesta);
    // 2. Entrada a cola R
   or
        receive(colaR, id);
        colaRLibre := false;
        send(buzonesPersonas[toInteger(id)], "ok");
    // 3. Entrada a cola L
   or
        receive(colaL, id);
        colaLLibre := false;
        send(buzonesPersonas[toInteger(id)], "ok");
    // 4. Liberación de cola
  or
        receive(liberar, cola);
        if cola = "R" then
            colaRLibre := true
        else
            colaLLibre := true;
    // 5. Imprimir o terminar
or
        receive(imprimir, mensaje);
        if mensaje = "TERMINAR" then
            activo := false
        else
            println(mensaje);
```

```
        end select;
    end;
end;
program ejercicio4;
const
    NUM_PERSONAS = 50;
var
    i: integer;
    enviar, colaR, colaL, liberar, imprimir: mailbox of string;
    buzonesPersonas: array[1..NUM_PERSONAS] of mailbox of string;
    personas: array[1..NUM_PERSONAS] of process;
begin
    crear buzones: enviar, colaR, colaL, liberar, imprimir;
    for i := 1 to NUM_PERSONAS do
        crear buzonesPersonas[i];
    start proceso Controlador(enviar, colaR, colaL, liberar, imprimir, buzonesPersonas);
    for i := 1 to NUM_PERSONAS do
        start proceso Persona(i, enviar, buzonesPersonas[i], colaR, colaL, liberar, imprimir);
end;
```

Cuestiones planteadas

a) Sí, ya que aunque cada cola solo puede ser utilizada por un único proceso a la vez, es posible que la cola rápida esté siendo utilizada por un hilo y la cola lenta por otro. El controlador se encarga de llevar un registro del estado de ocupación de las colas. Si una cola está ocupada, cuando un proceso Persona solicita acceso a ella, deberá esperar hasta que el proceso anterior haya liberado la cola correspondiente. De este modo, se asegura que únicamente un proceso persona utilice la cola en un momento dado, permitiendo así la concurrencia en el uso de ambas colas.

b) La exclusión mutua en el acceso a la pantalla se ha resuelto mediante el uso de un buzón de impresión. Cuando un proceso Persona desea imprimir en pantalla, envía un mensaje al buzón con el contenido a mostrar. El controlador recibe ese mensaje y se encarga de realizar la impresión. Esto garantiza que el acceso a la pantalla se haga de forma exclusiva, evitando que se produzcan solapamientos en la salida.

Código

```
package ejercicio4;

import messagepassing.*;

/**

* Clase que representa a una persona que realiza pagos en colas (R o L).

* Implementa Runnable para que cada persona sea un hilo concurrente.

*/

public class Persona implements Runnable {

   public static final int NUM_RANDOM = 10; // Máximo tiempo aleatorio

   public static final int PAGOS = 5;       // Cantidad de compras/pagos

   private int id; // Identificador único de la persona

   private int tiempoCompra;   //Tiempo que tarda en seleccionar las entradas

   private int tiempoPago;           //Tiempo que tara en pagar

   private MailBox buzonEnvio; //Buzon para enviar id al controlador


   // Buzones de comunicación entre hilos

   private MailBox buzonRespuesta;   //Buzón para recibir respuesta

   private MailBox colaR;            //Buzón para enviar id a la cola rápida

   private MailBox colaL;            //Buzón para enviar id a la cola lenta

   private MailBox liberar;    // Buzón para liberar la cola

   private MailBox imprimir;   //Buzón para imprimir

   private String asignadas;  //Cola asignada a cada persona, R o L

   /**

    * Constructor de Persona
```

```
     * @param id ID de la persona
     * @param buzonEnvio buzón para enviar solicitudes al controlador
     * @param buzonRespuesta buzón para recibir la respuesta del controlador
     * @param colaR buzón para cola R
     * @param colaL buzón para cola L
     * @param liberar buzón para liberar la cola
     * @param buzonImprimir buzón para enviar mensajes a imprimir
     */
    public Persona(int id, MailBox buzonEnvio, MailBox buzonRespuesta, MailBox colaR,
MailBox colaL,
                    MailBox liberar, MailBox buzonImprimir) {
        this.id = id;
        this.buzonEnvio = buzonEnvio;
        this.buzonRespuesta = buzonRespuesta;
        this.colaR = colaR;
        this.colaL = colaL;
        this.liberar = liberar;
        this.imprimir = buzonImprimir;
    }
    /**
     * Método que define el comportamiento concurrente de la persona:
     * 1. Simula compras.
     * 2. Solicita cola al controlador.
     * 3. Espera confirmación y paga.
     * 4. Libera la cola y notifica al sistema.
     */
    public void run() {
        for (int i = 0; i < PAGOS; i++) {
            //1.- Simula el tiempo de compra
            tiempoCompra = (int) (Math.random() * NUM_RANDOM) + 1;
```

```
try {
    Thread.sleep(tiempoCompra * 100);
} catch (InterruptedException e) {
    e.printStackTrace();
}
buzonEnvio.send(id); // 2.- Envía solicitud al controlador

//Recibe el tiempo de pago y la cola asignada
String[] partes = buzonRespuesta.receive().toString().split(",");
tiempoPago = Integer.parseInt(partes[0]);
asignadas = partes[1];
// Se pone en la cola asignada
if (asignadas.equals("R"))
    colaR.send(id);
else
    colaL.send(id);
// Espera confirmación de la cola
buzonRespuesta.receive();
//3.- Simula el pago
try {
    Thread.sleep(tiempoPago * 100);
} catch (InterruptedException e) {
    e.printStackTrace();
}
//4.- Libera la cola usada
liberar.send(asignadas);
//5.- Envía mensaje para impresión
Object info = "-------------------------\n" +
              "Persona " + id + " ha usado la cola " + asignadas +
```

```
                        "\nTiempo de pago = " + tiempoPago * 100 +
                        "\nThread.sleep(" + tiempoPago*100 + ")" +
                        "\nPersona " + id + " liberando la cola " + asignadas +
                        "\n-------------------------\n";
            imprimir.send(info);
        }
    }
}
package ejercicio4;
import messagepassing.*;
/**
* Clase que representa al controlador central del sistema.
* Se encarga de asignar colas a las personas, controlar el acceso concurrente y gestionar la liberación.
*/
public class Controlador extends Thread {
      private MailBox enviar, colaR, colaL, liberar, pantalla;
           private boolean colaRLibre = true, colaLLibre = true; //Estados de las colas
          private Selector s;   //Para manejar múltiples buzones
          private volatile boolean activo = true; //Controlar el flujo de ejecución del hilo
          private MailBox[] buzonesPersonas; // Buzones individuales por persona
          /**
            * Constructor del controlador.
            *
            * @param enviar Buzón de solicitudes de personas
            * @param colaR Cola derecha
            * @param colaL Cola izquierda
            * @param liberar Buzón de liberación de cola
            * @param pantalla Buzón para imprimir mensajes
```

```
     * @param buzonesPersonas Arreglo de buzones para responder a cada persona
     */
    public Controlador(MailBox enviar, MailBox colaR, MailBox colaL, MailBox liberar, MailBox pantalla, MailBox[] buzonesPersonas) {
        this.s = new Selector();
        this.enviar = enviar;
        this.liberar = liberar;
        this.colaR = colaR;
        this.colaL = colaL;
        this.pantalla = pantalla;
        this.buzonesPersonas = buzonesPersonas;
        //Añadir buzones al selector
        s.addSelectable(enviar, false);
        s.addSelectable(colaR, false);
        s.addSelectable(colaL, false);
        s.addSelectable(liberar, false);
        s.addSelectable(pantalla, false);
    }
    /**
     * Permite detener el bucle principal del hilo controlador.
     */
    public void terminar() {
        activo = false;
    }
    /**
     * Bucle principal del controlador. Atiende mensajes según disponibilidad de colas y realiza asignaciones.
     * Utiliza selector para evitar bloqueo activo.
     */
    public void run() {
```

```
while (activo) {
     //Actualiza las condiciones
    colaR.setGuardValue(colaRLibre);
    colaL.setGuardValue(colaLLibre);
    switch (s.selectOrBlock()) {
        case 1: // solicitud cola
            Object id = enviar.receive();
            String cajaAsignada;
            int tiempoPago = (int) (Math.random() * 10) + 1;
            cajaAsignada = (tiempoPago >= 5) ? "R" : "L";
            String respuesta = tiempoPago + "," + cajaAsignada;
            buzonesPersonas[(int) id].send(respuesta);
            break;
        case 2: // Entrada a cola R
            id = colaR.receive();
            colaRLibre = false;
            buzonesPersonas[(int) id].send("ok");
            break;
        case 3: // Entrada a cola L
            id = colaL.receive();
            colaLLibre = false;
            buzonesPersonas[(int) id].send("ok");
            break;
        case 4: // liberar
            Object cola = liberar.receive();
            if (cola.equals("R")) colaRLibre = true;
            else colaLLibre = true;
            break;
        case 5: // imprimir
```

```
                        Object mensaje = pantalla.receive();
                        if(mensaje.equals("TERMINAR")) {
                              terminar();
                        }else {
                              System.out.println(mensaje);
                        }
                        break;
                  }
               }
          }
}
package ejercicio4;
import messagepassing.*;
/**
* Clase principal que inicia la simulación de personas comprando y pagando en colas.
* Gestiona hilos y controla la concurrencia mediante el uso de buzones.
*/
public class Ejercicio4 {
       public static final int NUM_PERSONAS = 50; //Número personas en la simulación
           public static Thread[] Personas = new Thread[NUM_PERSONAS]; //Arreglo de
hilos
          /**
           * Método principal que lanza la ejecución del programa.
              * Crea los buzones, inicia al controlador y lanza todos los hilos
Persona.
           *
           * @param args No se utilizan argumentos de entrada.
           */
          public static void main(String[] args) {
              //Buzones para la comunicación
```

```
                MailBox enviar = new MailBox();

                MailBox colaR = new MailBox();

                MailBox colaL = new MailBox();

                MailBox liberar = new MailBox();

                MailBox pantalla = new MailBox();

                MailBox[] buzonesPersonas = new MailBox[NUM_PERSONAS];

                //Se crea el controlador

                     Controlador controlador = new Controlador(enviar, colaR, colaL,
liberar, pantalla, buzonesPersonas);

                controlador.start(); //Lo lanzamos

                //Creamos los clientes

                for (int i = 0; i < NUM_PERSONAS; i++) {

                    buzonesPersonas[i] = new MailBox();

                                    Personas[i] = new Thread(new Persona(i, enviar,
buzonesPersonas[i], colaR, colaL, liberar, pantalla));

                }

                //Inicializamos los hilos

                for (Thread persona : Personas) {

                    persona.start();

                }

           }

}
```