GÁLVEZ DÍAZ, JOAQUÍN (PORTAVOZ)
URBELZ ALONSO-CORTÉS, JORGE

PROGRAMACIÓN CONCURRENTE Y DISTRIBUIDA
GRUPO 3.1 y 3.3
CURSO 2023/24 – MAYO
SERGIO LÓPEZ BERNAL

BOLETÍN EJERCICIOS

Tabla de contenido

Ejercicio 1

Recursos no compartibles

La pantalla, ya que en un momento concreto solo un proceso consumidor podrá imprimir su id junto al resultado obtenido. Este acceso exclusivo es gestionado por un ReentrantLock.

Condiciones de sincronización

Los procesos consumidores no podrán leer el arrayValoresCompartido hasta que el proceso generador haya finalizado, ya que este es el encargado de introducir los valores aleatorios en el array.

El proceso sumador no podrá leer el arrayResultadosCompartido hasta que los procesos consumidores hayan finalizado, ya que cada uno de ellos se encarga de introducir el valor calculado en el array.

Pseudocódigo

```
***HILO GENERADOR***
/**
 * En esta clase se define la funcionalidad del proceso generador, el cuál introduce un
 * número y una operación (codificada con un entero, suma = 1, resta = 2 y multiplicación
 * = 3) alternativamente en un array de enteros de 110 posiciones,
 * "arrayValoresCompartido".
 */

arrayValoresCompartido[]: array de enteros privado volatil;

/**
 * El constructor recibe un array de enteros que será rellenado por el método "run" de la
 * clase.
 * @param arrayValoresCompartido     array de enteros.
 */

Constructor HiloGenerador(arrayValoresCompartido[]: array de enteros)
begin
	arrayValoresCompartido <- arrayValoresCompartido;
end

Procedimiento run()
begin
	longitudSubArray: entero;
	r: random;
	valor: entero;

	longitudSubArray <- Ejercicio1.N_VALORES / Ejercicio1.N_CONSUMIDORES;
	r <- Creacion Random;

	for i in 0..Ejercicio1.N_VALORES hacer
		valor <- i % longitudSubArray;

		si (valor % 2 = 0) entonces
			arrayValoresCompartido[i] <- r.Entero(100) + 1;
		sino
			arrayValoresCompartido[i] <- r.Entero(3) + 1;
		fin_si
	fin_for
end



***HILO CONSUMIDOR***
/**
 * En esta clase se define la funcionalidad de los procesos consumidores.
```

```
 * Cada uno de ellos, lee 11 posiciones consecutivas del array de enteros
 * "arrayValoresCompartido", calcula el resultado de las operaciones, lo almacena en otro
 * array de enteros llamado "arrayResultadosCompartido" e imprime por pantalla su id junto
 * al resultado obtenido.
*/

id: entero privado;
arrayValoresCompartido[]: array de enteros privado volatil;
arrayResultadosCompartido[]: array de enteros privado volatil;

/**
 * El constructor recibe el identificador del proceso y 2 arrays de enteros,
 * uno para leer las operaciones y otro para almacenar el resultado calculado.
 *
 * @param id   entero que representa el identificador del proceso.
 * @param arrayValoresCompartido     array de enteros.
 * @param arrayResultadosCompartido          array de enteros.
*/

Constructor HiloConsumidor(id: entero, arrayValoresCompartido[]: array de enteros,
                           arrayResultadosCompartido[]: array de enteros)
begin
       id <- id;
       arrayValoresCompartido <- arrayValoresCompartido;
       arrayResultadosCompartido <- arrayResultadosCompartido;
end

Procedimiento run()
begin
       longitudSubArray, inicioSubArray, finalSubArray, resultado, operador, operando:
entero;

       longitudSubArray <- Ejercicio1.N_VALORES / Ejercicio1.N_CONSUMIDORES;
       inicioSubArray <- id * longitudSubArray;
       finalSubArray <- inicioSubArray + Ejercicio1.N_CONSUMIDORES;
       resultado <- arrayValoresCompartido[inicioSubArray];

       for i <- inicioSubArray + 1; i < finalSubArray; i <- i + 2;
              operador <- arrayValoresCompartido[i];
              operando <- arrayValoresCompartido[i + 1];

              switch(operador)
                     caso 1:
                            resultado <- resultado + operando;
                            romper;
                     caso 2:
                            resultado <- resultado - operando;
                            romper;
                     caso 3:
                            resultado <- resultado * operando;
                            romper;
              fin_switch
       fin_for

       Ejercicio1.l.bloquear();
       intentar
              Imprimir("Hilo" + id + " : " + resultado + "\n");
       fin_intentar

       finalmente
              Ejercicio1.l.desbloquear();
       fin_finalmente

       arrayResultadosCompartido[id] <- resultado;

end
```

```
***HILO SUMADOR***
/**
 * En esta clase se define la funcionalidad del proceso sumador, el cuál lee el array de
 * enteros llamado "arrayResultadosCompartido" para sumar todos sus valores e imprimir
 * por pantalla el resultado.
 */

arrayResultadosCompartido[]: array de enteros privado volatil;

/**
 * El constructor recibe un array de enteros para sumar sus valores en el método "run".
 *
 * @param arrayResultadosCompartido         array de enteros.
 */

Constructor HiloSumador(arrayResultadosCompartido[]: array de enteros)
begin
        arrayResultadosCompartido <- arrayResultadosCompartido;
end

Procedimiento run()
begin
        total: entero;
        total <- 0;

        for valor:entero in arrayResultadosCompartido[] hacer
                total <- total + valor;
        fin_for

        Imprimir("Total: " + total);
end



***EJERCICIO1***
/**
 * La clase principal del programa. En ella se declaran todas las variables necesarias
 * para la correcta ejecución del programa.
*/

constante N_VALORES <- 110;
constante N_CONSUMIDORES <- 10;

l: Cerrojo público estático;

l <- Creación Cerrojo;

/**
 * Método principal del programa donde se realiza la ejecución de los distintos procesos
 * que lo componen. El cuál sigue el siguiente orden: Primero se ejecuta el proceso
 * generador, tras su finalización se ejecutan los procesos consumidores en concurrencia
 * y por último, se ejecuta el proceso sumador.
 *
 * @param args
 */

Procedimiento main()
begin
        arrayValoresCompartido[]: array de enteros;
        arrayResultadosCompartido[]: array de enteros;
        consumidores[]: array de HiloConsumidor;
        tConsumidores[]: array de Thread;

        g: HiloGenerador;
        s: HiloSumador;
```

```
        tg: Thread;
        ts: Thread;

        arrayValoresCompartido[] <- Creación entero[N_VALORES];
        arrayResultadosCompartido[] <- Creación entero[N_CONSUMIDORES];
        consumidores[] <- Creación HiloConsumidor[N_CONSUMIDORES];
        tConsumidores[] <- Creación Thread[N_CONSUMIDORES];

        g <- Creacion HiloGenerador(arrayValoresCompartido[]);

        for i in 0..N_CONSUMIDORES hacer
                consumidores[i] <- Creacion HiloConsumidor(i, arrayValoresCompartido[],
                                                           arrayResultadosCompartido[]);
        fin_for

        s <- Creacion HiloSumador(arrayResultadosCompartido[]);

        tg <- Creacion Thread(g);

        for i in 0..N_CONSUMIDORES hacer
                tConsumidores[i] <- Creacion Thread(consumidores[i]);
        fin_for

        ts <- Creacion Thread(s);

        tg.iniciar();

        intentar
                tg.unir();

        fin_intentar

        atrapar(e: ExcepcionInterrumpida)
                e.imprimirTrazaPila();

        fin_atrapar

        for i in 0..N_CONSUMIDORES hacer
                tConsumidores[i].iniciar();
        fin_for

        intentar
                for i in 0..N_CONSUMIDORES hacer
                        tConsumidores[i].unir();
                fin_for

        fin_intentar

        atrapar(e: ExcepcionInterrumpida)
                e.imprimirTrazaPila();

        fin_atrapar

        ts.iniciar();
end
```

Cuestiones planteadas en los ejercicios

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

Los hilos que pueden trabajar concurrentemente son los del tipo HiloConsumidor, ya que cada uno tiene unas posiciones de memoria asignadas y esto evita que haya conflictos entre ellos. Por lo tanto, pueden leer el arrayValoresCompartido, realizar el cálculo, imprimir por pantalla el resultado obtenido y escribir en el arrayResultadosCompartido simultaneamente.

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.

¿Cuál de las opciones consideras que es la correcta? Justifica la respuesta.

Entre hilos del tipo HiloConsumidor las impresiones están desordenadas, ya que esto no afecta al resultado final del problema.

Entre los hilos del tipo HiloConsumidor y el hilo del tipo HiloSumador, las impresiones están ordenadas, ya que es condición de sincronización del problema que hasta que los procesos consumidores no hayan finalizado el proceso generador no puede ejecutarse.

Por lo tanto, consideramos que esta es la opción correcta por lo anteriormente dicho.

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

La impresión del total podría aparecer antes o entre la impresión de los resultados obtenidos por cada HiloConsumidor.

Código Java

```
package ejercicio1;

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

/**
 * En esta clase se define la funcionalidad del proceso generador, el cuál introduce un
 * número y una operación (codificada con un entero, suma = 1, resta = 2 y
 * multiplicación = 3) alternativamente * en un array de enteros de 110 posiciones,
 * "arrayValoresCompartido".
 *
 * @author galve
 *
 */
public class HiloGenerador implements Runnable {
	private volatile int arrayValoresCompartido[];

	/**
	 * El constructor recibe un array de enteros que será rellenado por el método
	 * "run" de la clase.
	 * @param arrayValoresCompartido	array de enteros.
	 */
	public HiloGenerador(int arrayValoresCompartido[]) {
		this.arrayValoresCompartido = arrayValoresCompartido;
	}

	public void run() {
		int longitudSubArray = Ejercicio1.N_VALORES / Ejercicio1.N_CONSUMIDORES;
		Random r = new Random();
		int valor;

		for (int i = 0; i < Ejercicio1.N_VALORES; i++) {
			valor = i % longitudSubArray;

			if (valor % 2 == 0)
				arrayValoresCompartido[i] = r.nextInt(100) + 1;

			else
				arrayValoresCompartido[i] = r.nextInt(3) + 1;
		}
	}
}


/**
 * En esta clase se define la funcionalidad de los procesos consumidores.
 * Cada uno de ellos, lee 11 posiciones consecutivas del array de enteros
 * "arrayValoresCompartido", calcula el resultado de las operaciones, lo almacena en
 * otro array de enteros llamado "arrayResultadosCompartido" e imprime por pantalla su
```

```
 * id junto al resultado obtenido.
 *
 * @author galve
 *
 */
public class HiloConsumidor implements Runnable{
	private int id;
	private volatile int arrayValoresCompartido[];
	private volatile int arrayResultadosCompartido[];

	/**
	 * El constructor recibe el identificador del proceso y 2 arrays de enteros,
	 * uno para leer las operaciones y otro para almacenar el resultado calculado.
	 *
	 * @param id   entero que representa el identificador del proceso.
	 * @param arrayValoresCompartido     array de enteros.
	 * @param arrayResultadosCompartido          array de enteros.
	 */
	public HiloConsumidor(int id, int arrayValoresCompartido[], int
                                                        arrayResultadosCompartido[]){
		this.id = id;
		this.arrayValoresCompartido = arrayValoresCompartido;
		this.arrayResultadosCompartido = arrayResultadosCompartido;
	}

	public void run() {
		int longitudSubArray = Ejercicio1.N_VALORES / Ejercicio1.N_CONSUMIDORES;
		int inicioSubArray = id * longitudSubArray;
		int finalSubArray = inicioSubArray + Ejercicio1.N_CONSUMIDORES;
		int resultado = arrayValoresCompartido[inicioSubArray];
		int operador, operando;

		for (int i = inicioSubArray + 1; i < finalSubArray; i += 2) {
			operador = arrayValoresCompartido[i];
			operando = arrayValoresCompartido[i + 1];

			switch (operador) {
				case 1:
					resultado += operando;
					break;
				case 2:
					resultado -= operando;
					break;
				case 3:
					resultado *= operando;
					break;
			}
		}

		Ejercicio1.l.lock();
		try {
			System.out.println("Hilo " + id + " : " + resultado);
		} finally {
			Ejercicio1.l.unlock();
		}

		arrayResultadosCompartido[id] = resultado;
	}
}

/**
 * En esta clase se define la funcionalidad del proceso sumador, el cuál lee el array de
 * enteros llamado "arrayResultadosCompartido" para sumar todos sus valores e imprimir
 * por pantalla el resultado.
 */
public class HiloSumador implements Runnable {
	private volatile int arrayResultadosCompartido[];

	/**
	 * El constructor recibe un array de enteros para sumar sus valores en el método
	 * "run".
	 *
	 * @param arrayResultadosCompartido   array de enteros.
	 */
	public HiloSumador(int arrayResultadosCompartido[]){
		this.arrayResultadosCompartido = arrayResultadosCompartido;
	}
```

```
	public void run() {
		int total = 0;

		for (int valor : arrayResultadosCompartido)
			total += valor;

		System.out.println("Total: " + total);
	}
}

/**
 * La clase principal del programa. En ella se declaran todas las variables necesarias
 * para la correcta ejecución del programa.
 * @author galve
 *
 */
public class Ejercicio1 {
	public static final int N_VALORES = 110;
	public static final int N_CONSUMIDORES = 10;

	public static ReentrantLock l = new ReentrantLock();

	/**
	 * Método principal del programa donde se realiza la ejecución de los distintos
	 * procesos que lo componen. El cuál sigue el siguiente orden: Primero se ejecuta
	 * el proceso generador, tras su finalización se ejecutan los proceso
	 * consumidores en concurrencia y por último, se ejecuta el proceso sumador.
	 *
	 * @param args
	 */
	public static void main(String[] args) {
		int arrayValoresCompartido[] = new int[N_VALORES];
		int arrayResultadosCompartido[] = new int[N_CONSUMIDORES];
		HiloConsumidor consumidores[] = new HiloConsumidor[N_CONSUMIDORES];
		Thread tConsumidores[] = new Thread[N_CONSUMIDORES];

		HiloGenerador g = new HiloGenerador(arrayValoresCompartido);

		for (int i = 0; i < N_CONSUMIDORES; i++)
			consumidores[i] = new HiloConsumidor(i, arrayValoresCompartido,
                                                 arrayResultadosCompartido);

		HiloSumador s = new HiloSumador(arrayResultadosCompartido);

		Thread tg = new Thread(g);

		for (int i = 0; i < N_CONSUMIDORES; i++)
			tConsumidores[i] = new Thread(consumidores[i]);

		Thread ts = new Thread(s);

		tg.start();

		try {
			tg.join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}

		for (int i = 0; i < N_CONSUMIDORES; i++)
			tConsumidores[i].start();

		try {
			for (int i = 0; i < N_CONSUMIDORES; i++)
				tConsumidores[i].join();
		} catch (InterruptedException e) {
			e.printStackTrace();
		}

		ts.start();
		}
}
```

Ejercicio 2

Recursos no compartibles

En el ejercicio de los semáforos, “vehiculosCruzandoNS”, “vehiculosCruzandoEO”, “peatonCruzando”, “vehiculosEsperandoNS”, “vehiculosEsperandoEO”, “peatonesEsperando” son recursos no compartibles, ya que al ser variables enteras que cambian de valor, debemos asegurarnos de que nadie más usa la variable mientras un hilo lo esté haciendo. Además, la pantalla es otro recurso no compartible, no puede haber más de un proceso usándola a la vez., por lo que la protegeremos con un semáforo.

Condiciones de sincronización

En las condiciones de sincronización, debemos asegurarnos de que los vehículos y los peatones antes de cruzar tengan su semáforo en verde (cambiando el valor de la variable turno desde el hilo cruce), si van a cruzar no haya: en el caso de peatones, vehículos cruzando; y en el caso de los vehículos, peatones cruzando ni vehículos cruzando en la otra dirección. Para acabar, el número de entidades cruzando no debe superar: 10 en el caso de peatones y 4 en el caso de vehículos.

Pseudocódigo

```
***HILO CRUCE***
/**La clase HiloCruce se encargará de del cruce del semáforo. Cada
 * cinco segundos irá alterando el valor de la variable turno. Cuando
 * turno = 1, es el momento de cruce de los vehículos provenientes del
 * Norte. Cuando turno = 2, le toca a los que vengan del Este. Por
 * último, los peatones cruzarán cuando turno = 3.
 */
Procedimiento run()
begin
	for i in 0..5 hacer
		intentar
			Main.turno <- 1;
			wait(Main.pantalla);
			Imprimir("Semáforo verde para vehículos Norte-Sur\n");
			signal(Main.pantalla);
			sleep(5000);

			Main.turno <- 2;
			wait(Main.pantalla);
			Imprimir("Semáforo verde para vehículos Este-Oeste\n");
			signal(Main.pantalla);
			sleep(5000);

			Main.turno <- 3;
			wait(Main.pantalla);
			Imprimir("Semáforo verde para peatones\n");
			signal(Main.pantalla);
			sleep(5000);
		fin_intentar
		atrapar(e: ExcepcionInterrumpida)
			e.imprimirTrazaPila();
		fin_atrapar
	fin_for
end


***HILO CRUCE_PEATON***
/**CruceP se encarga de toda la parafernalia de los peatones. Cada uno
 * de los hilos de los peatones recorrerá este código. Al principio se
 * encuentra la condición de entrada. Si el peatón no la cumple, es que
 * es su turno y puede empezar a cruzar. En cambio, si la cumple, debe
 * aguardar a que sea su turno.
```

```
 */
Procedimiento run()
begin
	mientras true hacer
		intentar
			wait(Main.mutex);
			si Main.turno!=3 o Main.vehiculoCruzandoNS>0 o
								Main.vehiculoCruzandoEO>0 o
								Main.peatonCruzando>=10 hacer
				Main.peatonesEsperando++;

				si Main.turno=1 o Main.vehiculoCruzandoNS<4 o
								Main.vehiculosEsperandoNS>0 hacer
					signal(Main.SemaforoNS);
				sino si Main.turno=2 o Main.vehiculoCruzandoEO<4 o
								Main.vehiculosEsperandoEO>0 hacer
					signal(Main.SemaforoEO);
				fin_si

				signal(Main.mutex);
				wait(Main.SemaforoPeaton);
				Main.peatonesEsperando--;
			fin_si

			Main.peatonCruzando++;
			si Main.peatonCruzando<10 y Main.peatonesEsperando>0 hacer
				signal(Main.SemaforoPeaton);
			sino signal(Main.mutex);
			fin_si

			wait(Main.pantalla);
			Imprimir("Peatón cruzando\n");
			signal(Main.pantalla);
			sleep(3000);

			wait(Main.mutex);
			Main.peatonCruzando--;
			si Main.peatonCruzando=0 hacer
				si Main.turno=1 y Main.vehiculosEsperandoNS>0 hacer
					signal(SemaforoNS);
				sino si Main.turno=2 y Main.vehiculosEsperandoEO>0 hacer
					signal(SemaforoEO);
				fin_si
			sino si Main.turno=3 y Main.peatonesEsperando>0 hacer
				signal(Main.SemaforoPeaton);
			fin_si

			sino signal(Main.mutex);
			sleep(8000);
		fin_intentar
		atrapar(e: ExcepcionInterrumpida)
			e.imprimirTrazaPila();
		fin_atrapar
	fin_mientras
end


***HILO CRUCEVEHICULO***
/**CruceV se encarga de toda la parafernalia de los vehículos. Cada uno
 * de los hilos de los vehículos recorrerá este código. Al principio se
 * encuentra la condición de entrada. Si el vehículo no la cumple, es que
 * es su turno y puede empezar a cruzar. En cambio, si la cumple, debe
 * aguardar a que sea su turno. En este código tratamos la posibilidad,
 * tanto de que el vehículo esté cruzando en la dirección Norte-Sur, como
 * la Este-Oeste (la comenzará a recorrer una vez cruzado en la otra
 * dirección).
 */
Procedimiento run()
```

```
begin
	mientras true hacer
		intentar
			wait(Main.mutex);
			si Main.turno!=1 o Main.vehiculoCruzandoEO>0 o
								Main.peatonCruzando>0 o
								Main.vehiculoCruzandoNS>=4 hacer
				Main.vehiculosEsperandoNS++;

				si Main.turno=2 o Main.vehiculoCruzandoeo<4 o
								Main.vehiculosEsperandoEO>0 hacer
					signal(Main.SemaforoEO);
				sino si Main.turno=3 o Main.peatonCruzando<10 o
								Main.peatonesEsperando>0 hacer
					signal(Main.SemaforoPeaton);
				fin_si

				signal(Main.mutex);
				wait(Main.SemaforoNS);
				Main.vehiculosEsperandoNS--;
			fin_si

			Main.vehiculosCruzandoNS++;
			si Main.vehiculoCruzandoNS<4 y Main.vehiculosEsperandoNS>0 hacer
				signal(Main.SemaforoNS);
			sino signal(Main.mutex);
			fin_si

			wait(Main.pantalla);
			Imprimir("Vehículo cruzando direccíon Norte-Sur\n");
			signal(Main.pantalla);
			sleep(500);

			wait(Main.mutex);
			Main.vehiculosCruzandoNS--;
			si Main.vehiculosCruzandoNS=0 hacer
				si Main.turno=2 y Main.vehiculosEsperandoEO>0 hacer
					signal(SemaforoEO);
				sino si Main.turno=3 y Main.peatonesEsperando>0 hacer
					signal(SemaforoPeaton);
				fin_si
			sino si Main.turno=1 y Main.vehiculosEsperandoNS>0 hacer
				signal(Main.SemaforoNS);
			fin_si

			signal(Main.mutex);
			sleep(7000);


			wait(Main.mutex);
			si Main.turno!=2 o Main.vehiculoCruzandoNS>0 o
								Main.peatonCruzando>0 o
								Main.vehiculoCruzandoEO>=4 hacer
				Main.vehiculosEsperandoEO++;

				si Main.turno=1 o Main.vehiculoCruzandoNS<4 o
								Main.vehiculosEsperandoNS>0 hacer
					signal(Main.SemaforoNS);
				sino si Main.turno=3 o Main.peatonCruzando<10 o
								Main.peatonesEsperando>0 hacer
					signal(Main.SemaforoPeaton);
				fin_si

				signal(Main.mutex);
				wait(Main.SemaforoEO);
				Main.vehiculosEsperandoEO--;
			fin_si
```

```
                        Main.vehiculosCruzandoEO++;

                        si Main.vehiculoCruzandoEO<4 y Main.vehiculosEsperandoEO > 0 hacer
                                signal(Main.SemaforoEO);
                        sino signal(Main.mutex);
                        fin_si

                        wait(Main.pantalla);
                        Imprimir("Vehículo cruzando dirección Este-Oeste\n");
                        signal(Main.pantalla);
                        sleep(500);

                        wait(Main.mutex);
                        Main.vehiculosCruzandoEO--;
                        si Main.vehiculosCruzandoEO=0 hacer
                                si Main.turno=1 y Main.vehiculosEsperandoNS>0 hacer
                                        signal(SemaforoNS);
                                sino si Main.turno=3 y Main.peatonesEsperando>0 hacer
                                        signal(SemaforoPeaton);
                                fin_si
                        sino si Main.turno=2 y Main.vehiculosEsperandoEO>0 hacer
                                signal(Main.SemaforoEO);
                        fin_si

                        signal(Main.mutex);
                        sleep(7000);
                fin_intentar
                atrapar(e: ExcepcionInterrumpida)
                        e.imprimirTrazaPila();
                fin_atrapar
        fin_mientras
end


***Main***
/**La clase principal del programa. Se encargará de inicializar todas
 * las variables y semáforos.
 */
vehiculosCruzandoNS: entero;
vehiculosCruzandoEO: entero;
peatonCruzando: entero;
vehiculosEsperandoNS: entero;
vehiculosEsperandoEO: entero;
peatonesEsperando: entero;

vehiculosCruzandoNS <- 0;
vehiculosCruzandoEO <- 0;
peatonCruzando <- 0;
vehiculosEsperandoNS <- 0;
vehiculosEsperandoEO <- 0;
peatonesEsperando <- 0;

turno: entero;
turno <- 0;

SemaforoNS: Semáforo;
SemaforoEO: Semáforo;
SemaforoPeaton: Semáforo;
mutex: Semáforo;
pantalla: Semáforo;

initial(SemaforoNS, 0);
initial(SemaforoEO, 0);
initial(SemaforoPeaton, 0);
initial(mutex, 1);
initial(pantalla, 1);
```

```
Procedimiento main()
begin
	cruce: HiloCruce;
	cruceVehiculo: HiloCruceVehiculo;
	crucePeaton: HiloCrucePeaton;

	hilosVehiculos: Array de hilos[50];
	hilosPeatones: Array de hilos[100];

	cruce.iniciar();

	for i in 0..50 hacer
		hilosVehiculos[i] <- Creacion Hilo(cruceVehiculo);
		hilosVehiculos[i].iniciar();
	fin_for

	for i in 0..100 hacer
		hilosPeatones[i] <- Creacion Hilo(crucePeaton);
		hilosPeatones[i].iniciar();
	fin_for

	intentar
		cruce.unir();
		for valor:hilo in hilosVehiculos hacer
			hilo.unir();
		fin_for
		for valor:hilo in hilosPeatones hacer
			hilo.unir();
		fin_for

	fin_intentar
	atrapar(e: ExcepcionInterrumpida)
		e.imprimirTrazaPila();
	fin_atrapar

	Imprimir("Fin del hilo principal\n");
end
```

Cuestiones planteadas en los ejercicios

a) ¿Qué acciones pueden realizar simultáneamente los hilos?

En este ejercicio disponemos de un total de 151 hilos funcionando al unísono (sin contar el programa principal). Todos funcionan sin parar. El hilo del bucle se encarga de cambiar cada cinco segundo el turno de cruce. Empieza dando la prioridad a los vehículos provenientes del Norte, para seguidamente, dársela a los del Este y luego a peatones. Y así sucesivamente. Por otro lado, los hilos de peatones y vehículos tendrán una duración infinita (no nos interesan que paren de cruzar). Tendrán varias fases dentro de su bucle. Primero intentarán cruzar: si las condiciones son las ideales, cruzan; si no, se bloquean hasta que les liberen. Una vez se disponen a cruzar, tienen la opción de liberar a otros hilos hermanos o también pueden liberar el mutex. Tras esto, piden permiso para imprimir (y tras recibirlo, imprimen). Para acabar, deben cruzar la condición de salida que, dependiendo de las condiciones que se den, liberaran a un tipo de hilo o a otro. Finalmente, se duermen y comienzan desde el principio.

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

Vayamos por partes. El semáforo más fácil de explicar, por su motivo de existencia, es el de la pantalla. Cuando un hilo quiera imprimir por pantalla, simplemente deberá pedir acceso al semáforo de pantalla. Una vez se le conceda ese acceso, bloqueará al resto de hilos que intenten imprimir. Una vez, acabe, lo liberará.

Los tres semáforos utilizados para cada uno de los turnos (peatones, vehículos en dirección Norte y vehículos en dirección Este) funcionan exactamente igual. Una vez uno de los hilos intenta cruzar pero se le deniega tal, se bloquea en un wait() de su semáforo a la espera de recibir un signal(). Este signal() lo puede recibir de muchos otros hilos. Quizás se lo ofrezca otro hilo de su índole, una vez esté cruzando. O quizás, lo recibirá por parte de un hilo opuesto una vez que este termine o también se vaya a bloquear.

El último semáforo, el mutex. Basado en las diapositivas de teoría (lector/escritor, siendo el escritor el que lleva la prioridad), decidimos hacer uso de este semáforo. Se encargará de controlar el acceso a algunos recursos no compartibles, así como de la condición de entrada y salida de cada hilo.

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

Según se nos planteaba el ejercicio, el semáforo tiene tres turnos que deben ser alternados sucesivamente. El semáforo comienza dando el paso a los vehículos provenientes del Norte. Posteriormente (a los cinco segundos), será el turno de los vehículos que vengan desde el Este. Por último, a los cinco segundos será el momento de los peatones de cruzar. Este ciclo se repite una y otra vez.

En el caso de los vehículos, todos empezarán viniendo desde la dirección Norte, y una vez hayan cruzado, volverán a intentarlo desde el Este. Su secuencia sería la siguiente: N-E-N-E-N-E, y así, sucesivamente. Respondiendo a la pregunta: no, no es posible. En el turno de Norte a Sur solamente podrán pasar vehículos con esa dirección, en el caso contrario igual.

Código Java

```
package ejercicio2;

import java.util.concurrent.Semaphore;

/**La clase HiloCruce se encargará de del cruce del semáforo. Cada
 * cinco segundos irá alterando el valor de la variable turno. Cuando
 * turno = 1, es el momento de cruce de los vehículos provenientes del
 * Norte. Cuando turno = 2, le toca a los que vengan del Este. Por
 * último, los peatones cruzarán cuando turno = 3.
 */
public class HiloCruce extends Thread{
	public void run() {
		for(int i=0;i<5;i++){
			try {
				Main.turno=1;
				Main.pantalla.acquire();
				System.out.println("Semáforo verde para vehículos NS");
				Main.pantalla.release();
				Thread.sleep(5000);

				Main.turno=2;
				Main.pantalla.acquire();
				System.out.println("Semáforo verde para vehículos EO");
				Main.pantalla.release();
				Thread.sleep(5000);

				Main.turno=3;
				Main.pantalla.acquire();
				System.out.println("Semáforo verde para peatones");
				Main.pantalla.release();
				Thread.sleep(5000);

			} catch (InterruptedException e) {}
		}
	}
}

/**CruceP se encarga de toda la parafernalia de los peatones. Cada uno
```

```
 * de los hilos de los peatones recorrerá este código. Al principio se
 * encuentra la condición de entrada. Si el peatón no la cumple, es que
 * es su turno y puede empezar a cruzar. En cambio, si la cumple, debe
 * aguardar a que sea su turno.
 */
public class CruceP implements Runnable{
	public void run() {
		while(true) {
			try {
				Main.mutex.acquire();
				if (Main.turno!=3 || Main.vehiculosCruzandoNS>0 ||
				     Main.vehiculosCruzandoEO>0|| Main.peatonCruzando>=10){
				   Main.peatonesEsperando++;
				   if (Main.turno==1 && Main.vehiculosCruzandoNS<4 &&
				                Main.vehiculosEsperandoNS>0) {
				          Main.SemaforoNS.release();
				   }
				   else if (Main.turno==2 &&
				                Main.vehiculosCruzandoEO<4 &&
				                Main.vehiculosEsperandoEO>0) {
				          Main.SemaforoEO.release();
				   }
				   Main.mutex.release();
				   Main.SemaforoPeaton.acquire();
				   Main.peatonesEsperando--;
				}

				Main.peatonCruzando++;
				if (Main.peatonCruzando<10 && Main.peatonesEsperando>0) {
					Main.SemaforoPeaton.release();
				}else Main.mutex.release();

				Main.pantalla.acquire();
				System.out.println("Peatón cruzando");
				Main.pantalla.release();
				Thread.sleep(3000);

				Main.mutex.acquire();
				Main.peatonCruzando--;

				if (Main.peatonCruzando==0) {
					if (Main.turno==1 && Main.vehiculosEsperandoNS>0) {
						Main.SemaforoNS.release();
					}
					else if(Main.turno==2&&Main.vehiculosEsperandoEO>0){
						Main.SemaforoEO.release();
					}
				}else if (Main.turno==3 && Main.peatonesEsperando>0) {
					Main.SemaforoPeaton.release();
				}

				Main.mutex.release();
				Thread.sleep(8000);

			} catch (InterruptedException e) {}
		}
	}
}

/**CruceV se encarga de toda la parafernalia de los vehículos. Cada uno
 * de los hilos de los vehículos recorrerá este código. Al principio se
 * encuentra la condición de entrada. Si el vehículo no la cumple, es que
 * es su turno y puede empezar a cruzar. En cambio, si la cumple, debe
 * aguardar a que sea su turno. En este código tratamos la posibilidad,
 * tanto de que el vehículo esté cruzando en la dirección Norte-Sur, como
 * la Este-Oeste (la comenzará a recorrer una vez cruzado en la otra
 * dirección).
 */
public class CruceV implements Runnable{
	public void run() {
		while(true) {
			try {
				Main.mutex.acquire();
				if (Main.turno!=1 || Main.peatonCruzando>0 ||
					Main.vehiculosCruzandoEO>0 ||
					Main.vehiculosCruzandoNS>=4) {
```

```
                        Main.vehiculosEsperandoNS++;
                        if (Main.turno==2 && Main.vehiculosCruzandoEO<4 &&
                                        Main.vehiculosEsperandoEO>0) {
                                Main.SemaforoEO.release();
                        }
                        else if (Main.turno==3 && Main.peatonCruzando<10 &&
                                        Main.peatonesEsperando>0) {
                                Main.SemaforoPeaton.release();
                        }
                        Main.mutex.release();
                        Main.SemaforoNS.acquire();
                        Main.vehiculosEsperandoNS--;
                }

                Main.vehiculosCruzandoNS++;
                if (Main.vehiculosCruzandoNS<4 &&
                                Main.vehiculosEsperandoNS>0) {
                        Main.SemaforoNS.release();
                }else Main.mutex.release();

                Main.pantalla.acquire();
                System.out.println("Vehículo cruzando dirección Norte-
                                                                        Sur");
                Main.pantalla.release();
                Thread.sleep(500);

                Main.mutex.acquire();
                Main.vehiculosCruzandoNS--;

                if (Main.vehiculosCruzandoNS==0) {
                        if (Main.turno==2 && Main.vehiculosEsperandoEO>0) {
                                Main.SemaforoEO.release();
                        }
                        else if(Main.turno==3 && Main.peatonesEsperando>0){
                                Main.SemaforoPeaton.release();
                        }
                }else if (Main.turno ==1 && Main.vehiculosEsperandoNS>0) {
                        Main.SemaforoNS.release();
                }

                Main.mutex.release();
                Thread.sleep(7000);


//*****************************************************************
//*****************************************************************

                Main.mutex.acquire();
                if (Main.turno!=2 || Main.peatonCruzando>0 ||
                                Main.vehiculosCruzandoNS>0 ||
                                Main.vehiculosCruzandoEO>=4) {
                        Main.vehiculosEsperandoEO++;
                        if (Main.turno==1 && Main.vehiculosCruzandoNS<4 &&
                                        Main.vehiculosEsperandoNS>0) {
                                Main.SemaforoNS.release();
                        }
                        else if (Main.turno==3 && Main.peatonCruzando<10 &&
                                        Main.peatonesEsperando>0) {
                                Main.SemaforoPeaton.release();
                        }
                        Main.mutex.release();
                        Main.SemaforoEO.acquire();
                        Main.vehiculosEsperandoEO--;
                }


                Main.vehiculosCruzandoEO++;
                if (Main.vehiculosCruzandoEO<4 &&
                                Main.vehiculosEsperandoEO>0) {
                        Main.SemaforoEO.release();
                }else Main.mutex.release();

                Main.pantalla.acquire();
                System.out.println("Vehículo cruzando dirección Este-
                                                                Oeste");
                Main.pantalla.release();
                Thread.sleep(500);
```

```
                                Main.mutex.acquire();
                                Main.vehiculosCruzandoEO--;

                                if (Main.vehiculosCruzandoEO==0) {
                                        if (Main.turno==1 && Main.vehiculosEsperandoNS>0) {
                                                Main.SemaforoNS.release();
                                        }
                                        else if(Main.turno==3 && Main.peatonesEsperando>0){
                                                Main.SemaforoPeaton.release();
                                        }
                                }else if (Main.turno==2 && Main.vehiculosEsperandoEO>0) {
                                        Main.SemaforoEO.release();
                                }


                                Main.mutex.release();
                                Thread.sleep(7000);

                        } catch (InterruptedException e) {}
                }
        }
}

/**La clase principal del programa. Se encargará de inicializar todas
 * las variables y semáforos.
 */
public class Main {
        public static int vehiculosCruzandoNS = 0;
        public static int vehiculosCruzandoEO = 0;
        public static int peatonCruzando = 0;

        public static int vehiculosEsperandoNS = 0;
        public static int vehiculosEsperandoEO = 0;
        public static int peatonesEsperando = 0;

        public static int turno = 0;                // 1 = NS, 2 = EO, 3 = P

        public static Semaphore SemaforoNS = new Semaphore(0);
        public static Semaphore SemaforoEO = new Semaphore(0);
        public static Semaphore SemaforoPeaton = new Semaphore(0);
        public static Semaphore mutex = new Semaphore(1);
        public static Semaphore pantalla = new Semaphore(1);

        /**El método main genera todos los hilos que vamos a usar durante la
         * práctica. En total, generaremos un hilo para el cruce de semáforo,
         * 50 hilos para los vehículos y 100 para los peatones.
         * @param args
         * @throws InterruptedException
         */
        public static void main(String[] args) throws InterruptedException {
                CruceP cruceP = new CruceP();
                CruceV cruceV = new CruceV();

                Thread cruce = new HiloCruce();
                Thread[] hilosPeatones = new Thread[100];
                Thread[] hilosVehiculos = new Thread[50];

                cruce.start();

                for(int i=0; i < 50; i++) {
                        hilosVehiculos[i] = new Thread(cruceV);
                        hilosVehiculos[i].start();
                }

                for (int i = 0; i < 100; i++) {
                        hilosPeatones[i] = new Thread(cruceP);
                        hilosPeatones[i].start();
                }

                try {
                        cruce.join();
                        for (Thread hiloPeaton : hilosPeatones) { hiloPeaton.join(); }
                        for (Thread hiloVehiculo : hilosVehiculos){ hiloVehiculo.join(); }
                }catch (InterruptedException e) {
                        e.printStackTrace();
                }
```

```
                System.out.println("Fin del hilo principal");
        }
}
```

Ejercicio 3

Recursos no compartibles

Destacamos la pantalla y los arrays de enteros “maquinas[]”, “tiempoColaMesa[]” y “clienteColaMesa” declarados en MonitorMáquina y MonitorMesa.

Condiciones de sincronización

Un cliente no podrá adquirir una mesa sin antes haber adquirido y liberado una máquina.
Un cliente solo podrá adquirir una máquina si hay al menos una disponible.
Un cliente solo podrá adquirir su mesa asignada cuando el cliente que va por delante de él en la misma cola la haya liberado.

Pseudocódigo

```
***COMPONENTES DEL GRUPO***
Joaquín Gálvez Díaz (Portavoz)
Jorge Urbelz Alonso-Cortés


***MONITOR MAQUINA***
/**
 * Esta clase representa la funcionalidad que gestiona el acceso a las 3 máquinas
 * disponibles.
 * Cuenta con 2 métodos principales para solicitar y liberar una máquina.
 *
 */

constante privada entera N_MAQUINAS <- 3;

l: ReentrantLock privado
colaMaquina: Condition privado
maquinas[]: array de enteros privado

l <- ReentrantLock(true);
colaMaquina <- l.newCondition();
maquinas[] <- entero[N_MAQUINAS];

/**
 * El constructor inicializa a -1 el array que registra que clientes están ocupando las
máquinas.
 */

Constructor MonitorMaquina()
begin
        for i in 0 … N_MAQUINAS hacer
                maquinas[i] <- -1;
        fin_for
end

/**
 * Este método devuelve la primera máquina que está disponible (0, 1 o 2) o -1 si todas
 * las máquinas
 * están ocupadas.
 *
 * @param n           entero.
 * @return maquinaAsignada           entero.
 */

privado getMaquinaDisponible(n: entero): entero
begin
        maquinaAsignada: entero
        maquinaAsignada <- -1;
        for i in 0 … N_MAQUINAS hacer
```

```
                si maquinas[i] == n entonces
                        maquinaAsignada <- i;
                        devolver maquinaAsignada;
                fin_si
        fin_for
        devolver maquinaAsignada;
end

/**
 * Método para solicitar el acceso a una máquina.
 *
 * @param id   identificador del cliente.
 * @return maquinaAsignada   entero que representa la máquina que ha sido asginada al
 * cliente.
 * @throws InterruptedException .
 */

publico solicitarMaquina(id: entero): entero provoca InterruptedException
begin
        l.bloquear();
        maquinaAsignada: int

        intentar
                mientras(getMaquinaDisponible(-1) == -1) hacer
                                colaMaquina.await();
                        fin_mientras
                        maquinaAsignada <- getMaquinaDisponible(-1);
                        maquinas[maquinaAsignada] <- id;
        finalmente
                l.desbloquear();

        devolver maquinaAsignada;
end

/**
 * Método mediante el cuál un cliente abandona una máquina.
 * En caso de que haya algún cliente durmiendo porque esté a la espera de solicitar una
 * máquina se le despertará.
 *
 * @param maquinaAsignada     entero que representa la máquina que ha sido asginada al
cliente.
 */

publico liberarMaquina(maquinaAsignada: entero)
begin
        l.bloquear();
        intentar
                maquinas[maquinaAsignada] <- -1;
                colaMaquina.signal();
        finalmente
                l.desbloquear();
end


***MONITOR MESA***

/**
 * Esta clase representa la funcionalidad que gestiona el acceso a las 4 mesas
 * disponibles, cada una con una cola propia.
 * Cuenta con 2 métodos principales para solicitar y liberar una mesa.
 *
 */


constante privada N_MESAS <- 4;

l: ReentrantLock privado
```

```
colaMesa[]: array de Condition
tiempoColaMesa[], clienteColaMesa[]: array de enteros

l <- ReentrantLock(true);
colaMesa[] <- Condition[N_MESAS];
tiempoColaMesa[] <- entero[N_MESAS];
clienteColaMesa[] <- entero[N_MESAS];

/**
 * El constructor inicializa a 0 el array que registra el tiempo de cola de cada mesa y
 * a -1 el array que registra que clientes están ocupando las mesas.
 */
Constructor MonitorMesa()
begin
        for i in 0 … N_MESAS hacer
                tiempoColaMesa[i] <- 0;
                clienteColaMesa[i] <- -1;
                colaMesa[i] <- l.newCondition();
        fin_for
end

/**
 * Este método devuelve un entero que representa la mesa con menor tiempo de espera.
 *
 * @return mesaActual entero.
 */

privado mesaMenorTiempoEspera(): entero
begin
        tiempoActual, mesaActual: entero;
        tiempoActual <- tiempoColaMesa[0];
        mesaActual <- 0;
        for i in 1 … N_MESAS hacer
                        si tiempoColaMesa[i] < tiempoActual entonces
                                tiempoActual <- tiempoColaMesa[i];
                                mesaActual <- i;
                        fin_si

        devolver mesaActual;
end


/**
 * Método para imprimir por pantalla los tiempos de espera que le llevarán al cliente
 * a tomar la decisión de a qué mesa dirigirse y solicitar el acceso a ella.
 *
 * @param id   entero que representa el identificador del cliente.
 * @param maquinaAsignada     entero que representa la máquina que ha sido asginada al
cliente.
 * @param tiempoMaquina              entero que representa el tiempo en la máquina.
 * @param tiempoMesa          entero que representa el tiempo en la mesa.
 * @return mesaAsignada              entero que representa la mesa que ha sido asginada
al cliente.
 * @throws InterruptedException
 */

publico solicitarMesa(id, maquinaAsignada, tiempoMaquina, tiempoMesa: entero) provoca
InterruptedException
begin
        l.bloquear();
        mesaAsignada, numeroMaquina, numeroMesa: entero;
        intentar
                mesaAsignada <- mesaMenorTiempoEspera();
                numeroMaquina <- maquinaAsignada + 1;
                Imprimir("Cliente " + id + " ha solicitado su servicio en la máquina: " +
numeroMaquina + "\n");
                Imprimir("Tiempo en solicitar el servicio: " + tiempoMaquina + "\n");
```

```
                numeroMesa <- mesaAsignada + 1;
                Imprimir("Será atendido en la mesa: " + numeroMesa + "\n");
                Imprimir("Tiempo en la mesa = " + tiempoMesa + "\n");
                numeroMesa <- 1;
                Imprimir("Tiempo de espera en la mesa" + numeroMesa + "= " +
tiempoColaMesa[0]);
                numeroMesa <- numeroMesa + 1;
                for i in 1 … N_MESAS hacer
                        Imprimir(", mesa" + numeroMesa + "= " + tiempoColaMesa[i]);
                        numeroMesa <- numeroMesa + 1;
                fin_for
                Imprimir("\n\n");
                tiempoColaMesa[mesaAsignada] <- tiempoColaMesa[mesaAsignada] +
tiempoMesa;
                mientras clienteColaMesa[mesaAsignada] > -1 hacer
                        colaMesa[mesaAsignada].await();
                fin_mientras
                clienteColaMesa[mesaAsignada] <- id;
        finalmente
                l.desbloquear();
        devolver mesaAsignada;
end

/**
 * Método mediante el cuál un cliente abandona una mesa.
 * En caso de que haya otro cliente durmiendo en la cola de esta mesa porque esté a la
 * espera de solicitarla se le despertará.
 *
 * @param mesaAsignada   entero que representa la mesa que ha sido asginada al cliente.
 * @param tiempoMesa     entero que representa el tiempo en la mesa.
 */




Publico liberarMesa(mesaAsignada, tiempoMesa: entero)
begin
        l.bloquear();
        intentar
                clienteColaMesa[mesaAsignada] <- -1;
                tiempoColaMesa[mesaAsignada] <-tiempoColaMesa[mesaAsignada] - tiempoMesa;
                colaMesa[mesaAsignada].signal();
        finalmente
                l.desbloquear();
end

***HILO CLIENTE*** EXTIENDE THREAD

/**
 * En esta clase se define la funcionalidad de los hilos clientes, cada uno de ellos,
solicitará
 * la primera máquina que encuentre disponible para seleccionar un servicio.
 * Tras la selección, se dirigirá a la cola de la mesa con menor tiempo de espera para
realizar la gestión
 * que desea.
 *
 */

id, tiempoMaquina, tiempoMesa: entero;
monitorMaquina: MonitorMaquina;
monitorMesa: MonitorMesa;

/**
 * El constructor de la clase recibe el identificador del cliente,
 * el tiempo que tardará en seleccionar el servicio en la máquina,
 * el tiempo que tardará en realizar la gestión en la mesa
```

```
 * y los monitores que definen la funcionalidad que gestiona el acceso a las máquinas y
a las mesas.
 *
 * @param id   entero que representa el identificador del cliente.
 * @param tiempoMaquina             entero que representa el tiempo en la máquina.
 * @param tiempoMesa          entero que representea el tiempo en la mesa.
 * @param monitorMaquina      monitor de tipo máquina.
 * @param monitorMesa         monitor de tipo mesa.
 */

Constructor HiloCliente(id, tiempoMaquina, tiempoMesa: entero, monitorMaquina:
MonitorMaquina, monitorMesa: MonitorMesa)
begin
        id <- id;
        tiempoMaquina <- tiempoMaquina;
        tiempoMesa <- tiempoMesa;
        monitorMaquina <- monitorMaquina;
        monitorMesa <- monitorMesa;
end

publico run()
begin
        maquinaAsignada, mesaAsignada: entero;
        intentar
                maquinaAsignada <- monitorMaquina.solicitarMaquina(id);
                                        Thread.dormir(tiempoMaquina);
                monitorMaquina.liberarMaquina(maquinaAsignada);

                mesaAsignada <- monitorMesa.solicitarMesa(id, maquinaAsignada,
tiempoMaquina, tiempoMesa);
                Thread.dormir(tiempoMesa);
                monitorMesa.liberarMesa(mesaAsignada, tiempoMesa);


        capturar(e: InterruptedException)
                e.imprimirTrazaPila();
end


***MAIN***

/**
 * La clase principal del programa. En ella, se declaran e inicializan los monitores y
los hilos clientes.
 *
 */

constante privada N_CLIENTES <- 50;

/**
 * Método main donde se realiza la ejecución del programa principal.
 *
 * @param args .
 */

publico estático main(args[]: array de Strings)
begin
        r: Random;
        monitorMaquina: MonitorMaquina;
        monitorMesa: MonitorMesa;
        clientes[]: array de HiloCliente;

        r <- Random();
        monitorMaquina <- MonitorMaquina();
        monitorMesa <- MonitorMesa();
        clientes[] <- HiloCliente[N_CLIENTES];
```

```
        for i in 0 … N_CLIENTES hacer
                clientes[i] <- HiloCliente(i, r.nextInt(1000), r.nextInt(1000),
monitorMaquina, monitorMesa);
                clientes[i].start();
        fin_for

end
```

Cuestiones planteadas en los ejercicios

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

En exclusión mutua, lo garantiza la naturaleza de los monitores, ya que solo un hilo podrá estar ejecutando un método del monitor en cada momento.

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

Para el monitor de las maquinas he utilizado un monitor con una cola Condition y para el monitor de las mesas, un monitor con cuatro colas Condition, cada una de ellas representa una mesa.

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

En ambos monitores he utilizado signal para despertar al siguiente cliente en la cola y que sea lo más parecido a la vida real. A parte, en el caso del monitor de las mesas no tiene sentido despertar a todos los clientes si las mesas están ocupadas, ya que volverán a dormirse.

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

Se ha llevado a cabo en el método “solicitarMesa” del monitorMesa.
Se realiza antes de que el cliente se ponga a la cola de la mesa asignada.

Código Java

```
package ejercicio3;

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

/**
 * Esta clase representa la funcionalidad que gestiona el acceso a las 3 máquinas
 * disponibles.
 * Cuenta con 2 métodos principales para solicitar y liberar una máquina.
 *
 * @author Joaquín Gálvez Díaz - Effect3
 * @author Jorge Urbelz Alonso-Cortés - juacmola
 */
public class MonitorMaquina {
        private static final int N_MAQUINAS = 3;

        private ReentrantLock l = new ReentrantLock(true);
        private Condition colaMaquina = l.newCondition();
        private int maquinas[] = new int[N_MAQUINAS];

        /**
         * El constructor inicializa a -1 el array que registra que clientes están
         * ocupando las máquinas.
         */
        public MonitorMaquina() {
                for (int i = 0; i < N_MAQUINAS; i++) {
                        maquinas[i] = -1;
                }
        }

        /**
         * Este método devuelve la primera máquina que está disponible (0, 1 o 2) o -1 si
```

```
         * todas las máquinas están ocupadas.
         *
         * @param n    entero.
         * @return maquinaAsignada   entero.
         */
        private int getMaquinaDisponible(int n) {
                int maquinaAsignada = -1;

                for (int i = 0; i < N_MAQUINAS; i++) {
                        if (maquinas[i] == n) {
                                maquinaAsignada = i;
                                return maquinaAsignada;
                        }
                }

                return maquinaAsignada;
        }

        /**
         * Método para solicitar el acceso a una máquina.
         *
         * @param id   identificador del cliente.
         * @return maquinaAsignada   entero que representa la máquina que ha sido
         * asginada al cliente.
         * @throws InterruptedException .
         */
        public int solicitarMaquina(int id) throws InterruptedException {
                l.lock();
                int maquinaAsignada;

                try {
                        while(getMaquinaDisponible(-1) == -1) {
                                colaMaquina.await();
                        }

                        maquinaAsignada = getMaquinaDisponible(-1);
                        maquinas[maquinaAsignada] = id;

                } finally {
                        l.unlock();
                }

                return maquinaAsignada;
        }

        /**
         * Método mediante el cuál un cliente abandona una máquina.
         * En caso de que haya algún cliente durmiendo porque esté a la espera de
         * solicitar una máquina se le despertará.
         *
         * @param maquinaAsignada   entero que representa la máquina que ha sido asginada
         * al cliente.
         */
        public void liberarMaquina(int maquinaAsignada) {
                l.lock();
                try {
                        maquinas[maquinaAsignada] = -1;
                        colaMaquina.signal();

                } finally {
                        l.unlock();
                }
        }
}

/**
 * Esta clase representa la funcionalidad que gestiona el acceso a las 4 mesas
 * disponibles,
 * cada una con una cola propia.
 * Cuenta con 2 métodos principales para solicitar y liberar una mesa.
 *
 * @author Joaquín Gálvez Díaz - Effect3
 * @author Jorge Urbelz Alonso-Cortés - juacmola
 */
public class MonitorMesa {
        private static final int N_MESAS = 4;
```

```
private ReentrantLock l = new ReentrantLock(true);
private Condition colaMesa[] = new Condition[N_MESAS];
private int tiempoColaMesa[] = new int[N_MESAS];
private int clienteColaMesa[] = new int[N_MESAS];

/**
 * El constructor inicializa a 0 el array que registra el tiempo de cola de cada
 * mesa y a -1 el array que registra que clientes están ocupando las mesas.
 */
public MonitorMesa() {
	for (int i = 0; i < N_MESAS; i++) {
		tiempoColaMesa[i] = 0;
		clienteColaMesa[i] = -1;
		colaMesa[i] = l.newCondition();
	}
}

/**
 * Este método devuelve un entero que representa la mesa con menor tiempo de
 * espera.
 *
 * @return mesaActual entero.
 */
private int mesaMenorTiempoEspera() {
	int tiempoActual = tiempoColaMesa[0];
	int mesaActual = 0;

	for (int i = 1; i < N_MESAS; i++) {
		if (tiempoColaMesa[i] < tiempoActual) {
			tiempoActual = tiempoColaMesa[i];
			mesaActual = i;
		}
	}

	return mesaActual;
}

/**
 * Método para imprimir por pantalla los tiempos de espera que le llevarán al
 * cliente a tomar la decisión de a qué mesa dirigirse y solicitar el acceso a
 * ella.
 *
 * @param id   entero que representa el identificador del cliente.
 * @param maquinaAsignada   entero que representa la máquina que ha sido asginada
 * al cliente.
 * @param tiempoMaquina   entero que representa el tiempo en la máquina.
 * @param tiempoMesa   entero que representa el tiempo en la mesa.
 * @return mesaAsignada   entero que representa la mesa que ha sido asginada al
 * cliente.
 * @throws InterruptedException
 */
public int solicitarMesa(int id, int maquinaAsignada, int tiempoMaquina, int
                                    tiempoMesa) throws InterruptedException {
	l.lock();
	int mesaAsignada;
	try {
		mesaAsignada = mesaMenorTiempoEspera();
		int numeroMaquina = maquinaAsignada + 1;
		System.out.println("Cliente " + id + " ha solicitado su servicio
                                        en la máquina: " + numeroMaquina);
		System.out.println("Tiempo en solicitar el servicio: " +
                                                    tiempoMaquina);
		int numeroMesa = mesaAsignada + 1;
		System.out.println("Será atendido en la mesa: " + numeroMesa);
		System.out.println("Tiempo en la mesa = " + tiempoMesa);
		numeroMesa = 1;
		System.out.printf("Tiempo de espera en la mesa" + numeroMesa + "=
                                            " + tiempoColaMesa[0]);
		numeroMesa++;
		for (int i = 1; i < N_MESAS; i++) {
			System.out.printf(", mesa" + numeroMesa + "= " +
                                                    tiempoColaMesa[i]);
			numeroMesa++;
		}
		System.out.println("\n");

		tiempoColaMesa[mesaAsignada] = tiempoColaMesa[mesaAsignada] +
```

```
                                                                        tiempoMesa;

                        while(clienteColaMesa[mesaAsignada] > -1) {
                                colaMesa[mesaAsignada].await();
                        }

                        clienteColaMesa[mesaAsignada] = id;

                } finally {
                        l.unlock();
                }

                return mesaAsignada;
        }

        /**
         * Método mediante el cuál un cliente abandona una mesa.
         * En caso de que haya otro cliente durmiendo en la cola de esta mesa porque esté
         * a la espera de solicitarla se le despertará.
         *
         * @param mesaAsignada  entero que representa la mesa que ha sido asginada al
         * cliente.
         * @param tiempoMesa  entero que representa el tiempo en la mesa.
         */
        public void liberarMesa(int mesaAsignada, int tiempoMesa) {
                l.lock();
                try {
                        clienteColaMesa[mesaAsignada] = -1;
                        tiempoColaMesa[mesaAsignada] = tiempoColaMesa[mesaAsignada] -
                                                                        tiempoMesa;
                        colaMesa[mesaAsignada].signal();

                } finally {
                        l.unlock();
                }
        }
}

/**
 * En esta clase se define la funcionalidad de los hilos clientes, cada uno de ellos,
 * solicitará la primera máquina que encuentre disponible para seleccionar un servicio.
 * Tras la selección, se dirigirá a la cola de la mesa con menor tiempo de espera para
 * realizar la gestión que desea.
 *
 * @author Joaquín Gálvez Díaz - Effect3
 * @author Jorge Urbelz Alonso-Cortés - juacmola
 */
public class HiloCliente extends Thread{
        private int id;
        private int tiempoMaquina;
        private int tiempoMesa;
        private MonitorMaquina monitorMaquina;
        private MonitorMesa monitorMesa;

        /**
         * El constructor de la clase recibe el identificador del cliente,
         * el tiempo que tardará en seleccionar el servicio en la máquina,
         * el tiempo que tardará en realizar la gestión en la mesa
         * y los monitores que definen la funcionalidad que gestiona el acceso a las
         * máquinas y a las mesas.
         *
         * @param id   entero que representa el identificador del cliente.
         * @param tiempoMaquina      entero que representa el tiempo en la máquina.
         * @param tiempoMesa         entero que representea el tiempo en la mesa.
         * @param monitorMaquina     monitor de tipo máquina.
         * @param monitorMesa        monitor de tipo mesa.
         */
        public HiloCliente(int id, int tiempoMaquina, int tiempoMesa, MonitorMaquina
                                                monitorMaquina, MonitorMesa monitorMesa) {
                this.id = id;
                this.tiempoMaquina = tiempoMaquina;
                this.tiempoMesa = tiempoMesa;
                this.monitorMaquina = monitorMaquina;
                this.monitorMesa = monitorMesa;
        }

        public void run() {
```

```
                try {
                        int maquinaAsignada = monitorMaquina.solicitarMaquina(id);
                        Thread.sleep(tiempoMaquina);
                        monitorMaquina.liberarMaquina(maquinaAsignada);

                        int mesaAsignada = monitorMesa.solicitarMesa(id, maquinaAsignada,
                                                                tiempoMaquina, tiempoMesa);
                        Thread.sleep(tiempoMesa);
                        monitorMesa.liberarMesa(mesaAsignada, tiempoMesa);

                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
        }
}

/**
 * La clase principal del programa. En ella, se declaran e inicializan los monitores y
 * los hilos clientes.
 *
 * @author Joaquín Gálvez Díaz - Effect3
 * @author Jorge Urbelz Alonso-Cortés - juacmola
 */
public class Main {
        private static final int N_CLIENTES = 50;

        /**
         * Método main donde se realiza la ejecución del programa principal.
         *
         * @param args .
         */
        public static void main(String[] args) {
                Random r = new Random();
                MonitorMaquina monitorMaquina = new MonitorMaquina();
                MonitorMesa monitorMesa = new MonitorMesa();
                HiloCliente clientes[] = new HiloCliente[N_CLIENTES];

                for (int i = 0; i < N_CLIENTES; i++) {
                        clientes[i] = new HiloCliente(i, r.nextInt(1000), r.nextInt(1000),
                                                                monitorMaquina, monitorMesa);
                        clientes[i].start();
                }
        }
}
```

Ejercicio 4

Recursos no compartibles

En este ejercicio, el único recurso no compartible es la pantalla, que solamente podrá ser usada por un cliente cada vez. El resto de las variables, al ser utilizadas exclusivamente por el controlador, no necesitan ser parte de la exclusión mutua.

Condiciones de sincronización

La condición principal es que los clientes no pueden imprimir por pantalla si ya hay un cliente imprimiendo. Deben esperar a que termine el que la está usando. Por otro lado, siempre que se envía un send(), debemos recibir de vuelta un receive() ya que corremos el riesgo de que se quede el cliente o el controlador siempre esperando.

Pseudocódigo

```
***HILO CONTROLADOR***
/**La clase controlador se encarga de recoger los mensajes de los compradores
 * y enviarles otros devuelta. Definirá de manera aleatoria el tiempo que un
 * comprador estará en una caja (dependiendo del valor aleatorio se le envía
 * a la caja A o la B). Antes del constructor hemos decidido crear las variables
 * locales.
 */
buzonPregunta: Mailbox;
buzonCajaA: Mailbox;
buzonCajaB: Mailbox;
buzonAbandonoCaja: Mailbox;
arrayBuzonComunicacion: Mailbox[];

tiempo: entero;
devuelveAsignacion: String;
idPreguntaCaja: entero;
idCajaA: entero;
idCajaB: entero;
idAbandono: entero;

cajaAOcupada: booleano;
cajaBOcupada: booleano;

Constructor HiloControlador(pregunta: Mailbox, cajaA: Mailbox, cajaB: Mailbox, abandono:
                                                Mailbox, array Mailbox[])
begin
        buzonPregunta <- pregunta;
        buzonCajaA <- cajaA;
        buzonCajaB <- cajaB;
        buzonAbandonoCaja <- abandono;
        arrayBuzonComunicacion <- array;
        impresora <- impresora;

        cajaAOcupada <- false;
        cajaBOcupada <- false;
end

/**El controlador recibe todo tipo de mensajes por parte de los compradores,
 * a los que tendrá que responder una cosa u otra. El proceso se realiza
 * mediante un select (visto en clase de teoría). Proviene de una librería
 * creada exclusivamente para esta práctica. Dependiendo de unas condiciones
 * u otras, el controlador abre uno de los buzones y recorre el código de esa
 * opción.
 */
Procedimiento run()
begin
        mientras true hacer
                select entre
                        idPreguntaCaja <- receive(buzonPregunta);
```

```
                        tiempo <- Random[1,10];
                        si tiempo >= 5 hacer
                                devuelveAsignacion <- tiempo.convertiraString() + ",A";
                        sino devuelveAsignacion <- tiempo.convertiraString() + ",B";
                        fin_si
                        send(arrayBuzonComunicacion[idPreguntaCaja], devuelveAsignacion);
                or
                        when !cajaAOcupada =>
                        idCajaA <- receive(buzonCajaA);
                        cajaAOcupada <- true;
                        send(arrayBuzonComunicacion[idCajaA], "ok");
                or
                        when !cajaBOcupada =>
                        idCajaB <- receive(buzonCajaB);
                        cajaBOcupada <- true;
                        send(arrayBuzonComunicacion[idCajaB], "ok");
                or
                        idAbandono <- receive(buzonAbandonoCaja);
                        si idCajaA == idAbandono hacer
                                cajaAOcupada <- false;
                        sino cajaBOcupada <- false;
                        fin_si
                        send(arrayBuzonComunicacion[idAbandono], "ok");
                fin_select
        fin_mientras
end

***HILO COMPRADOR***
/**La clase comprador es usada por cada uno de los hilos comprador. Se
 * encargará de enviar y recibir mensajes popr sus buzones. Establecerá
 * conversación con el controlador. Antes del constructor hemos decidido
 * crear las variables locales.
 */
buzonPregunta: Mailbox;
buzonCajaA: Mailbox;
buzonCajaB: Mailbox;
buzonAbandonoCaja: Mailbox;
arrayBuzonComunicacion: Mailbox[];
impresora: Mailbox;

id: entero;
tiempo: String;
caja: String;
mensajeAsignacion: String;
mensajeImprimir: entero;

Constructor HiloComprador(id: entero, pregunta: Mailbox, cajaA: Mailbox,
                cajaB: Mailbox, abandono: Mailbox, array Mailbox[], impresora: Mailbox)
begin
        id <- id;
        buzonPregunta <- pregunta;
        buzonCajaA <- cajaA;
        buzonCajaB <- cajaB;
        buzonAbandonoCaja <- abandono;
        arrayBuzonComunicacion <- array;
        impresora <- impresora;
end

/**Cada comprador tendrá 5 fases a la hora de comprar (están recogidas dentro
* del método. Lo recorreremos un total de 5 veces, ya que nos lo pide
* expresamente el enunciado. Aquí se realiza la comunicación con el
* controlador. Se recogeran sus mensajes en el buzón de la posición del
* array que le corresponda, y se enviarán otros mensajes al controlador a
* través de alguno de los buzones (cada buzón contiene un tipo de mensaje
* específico).
*/
Procedimiento run()
begin
        for i in 0..5 hacer
```

```
                // PASO 1: REALIZA LA COMPRA
                sleep(Random()*1000);


                // PASO 2: SOLICITAR CAJA
                // PRIMERO PREGUNTA POR LA CAJA QUE ESTÉ LIBRE
                send(buzonPregunta, id);
                mensajeAsignacion <- receive(arrayBuzonComunicacion[id]);
                partes: String[];
                partes <- split(mensajeAsignacion, ",");
                tiempo <- partes[0];
                caja <- partes[1];

                // LUEGO SOLICITA ENTRAR EN ESA CAJA
                si caja == "A" hacer
                        send(buzonCajaA, id);
                sino send(buzonCajaB, id);
                fin_si
                receive(arrayBuzonComunicacion[id]);


                // PASO 3: PAGAR EN CAJA
                sleep(tiempo.convertiraInt()*1000);


                // PASO 4: LIBERAR CAJA
                send(buzonAbandonoCaja, id);
                receive(arrayBuzonComunicacion[id]);


                // PASO 5: IMPRIMIR POR PANTALLA
                mensajeImprimir <- receive(impresora);
                Imprimir("Persona " + (id+1) + " ha usado la caja " + caja + "\n");
                Imprimir("Tiempo de pago=" + tiempo + "\n");
                Imprimir("Thread.sleep(" + tiempo + ")\n");
                Imprimir("Persona " + (id+1) + " liberando la caja " + caja + "\n");
                send(impresora, mensajeImprimir);
        fin_for
end

***MAIN***
/**La clase principal del programa. Se encargará de inicializar todas
 * la variable de la pantalla (para activarla) y los buzones.
 */
preguntaComprador: Mailbox;
cajaA: Mailbox;
cajaB: Mailbox;
abandonoCaja: Mailbox;
impresora: Mailbox;

arrayBuzon: arrayList de Mailbox;
compradores: arrayList de HiloComprador;

controlador: HiloControlador;

activaImpresora: entero estatico;

/**Crea los buzones (como habrá 30 cleintes debemos crear un array de
* 30 posiciones, uno para cada cliente. Aquí recibiran sus mensajes),
* el hilo controlador y los 30 hilos de los clientes/compradores.
*/
Procedimiento main()
begin
        PreguntaComprador <- Creacion Mailbox;
        cajaA <- Creacion Mailbox;
        cajaB <- Creacion Mailbox;
        AbandonoCaja <- Creacion Mailbox;
        Impresora <- Creacion Mailbox;
```

```
        arrayBuzon <- Creacion Mailbox[30];
        for i in 0..30 hacer
                arrayBuzon[i] <- Creacion Mailbox;
        fin_for

        send(impresora, activaImpresora);

        controlador <- Creacion HiloControlador(preguntaComprador, cajaA, cajaB,
                                                abandonoCaja, arrayBuzon, impresora);
        controlador.iniciar();

        compradores <- Creacion HiloComprador[30];
        for i in 0..30 hacer
                compradores[i] <- Creacion HiloComprador(i, preguntaComprador, cajaA,
                                        cajaB, abandonoCaja, arrayBuzon, impresora);
                compradores[i].iniciar();
        fin_for
end
```

Cuestiones planteadas en los ejercicios

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

El ejercicio hace uso de dos cajas: A y B. Estas dos cajas van por separado y no dependen de la otra para funcionar. Funciona como la vida real, en un supermercado hay distintas cajas para poder pagar y cada una lleva su ritmo. En nuestro caso, el controlador da el acceso a uno de los clientes a la caja que esté libre y se olvida. Al controlador no le importa cuanto tiempo se quede dentro, simplemente espera que se le envíe el mensaje de liberación. Mientras una de las cajas está ocupada, si otro cliente quiere entrar en esa misma caja, deberá esperas. Sin embargo, si pide entrar en la otra caja y resulta que está libre, entrará (NO debe esperar a que las dos cajas estén libres).

b) ¿Cómo has resuelto la exclusión mutua de la pantalla?

Para resolver la exclusión mutua hemos construido un nuevo buzón. De esta manera, cuando un cliente quiere imprimir, enviará un receive() dando a entender que está esperando que se le otorgue acceso a la pantalla. Una vez que lo recibe, imprime y envía un send() al buzón. Así, libera el buzón para que otro cliente pueda imprimir.

En cuanto a cómo el primer cliente que imprime recibe la exclusión mutua, es muy sencillo. Simplemente, es el programa principal quien envía el primer send(). Como la comunicación es asíncrona, esta no es bloqueante en el caso del send().

Código Java

```
package ejercicio4;

import messagepassing.MailBox;
import messagepassing.Selector;

/**La clase comprador es usada por cada uno de los hilos comprador. Se
 * encargará de enviar y recibir mensajes popr sus buzones. Establecerá
 * conversación con el controlador. Antes del constructor hemos decidido
 * crear las variables locales.
 */
public class HiloComprador extends Thread{
        private MailBox buzonPregunta;
        private MailBox buzonCajaA;
        private MailBox buzonCajaB;
        private MailBox buzonAbandonoCaja;
        private MailBox[] arrayBuzonComunicacion;
        private MailBox imprimir;

        private int id;
        private String tiempo;
```

```
private String caja;
private String mensajeAsignacion;
private String mensajeConfirmacion;
private int mensajeImprimir;

/**El constructor recogerá todos los buzones creados por el programa
 * principal y los ligará a otros buzones, además del id del hilo.
 * @param id - Sirve para saber que hilo está imprimiendo
 * @param pregunta - Envía las veces que un comprador pregunta por una caja
 * @param cajaA - Envía los mensajes pertenecientes al buzón A
 * @param cajaB - Envía los mensajes pertenecientes al buzón B
 * @param abandono - Envía los mensajes de abandono de una caja
 * @param array - Recoge los mensajes del controlador a él específicamente
 * @param imprimir - Sirve para pedir permiso de impresión
 */
public HiloComprador(int id, MailBox pregunta, MailBox cajaA, MailBox cajaB,
                     MailBox abandono, MailBox[] array, MailBox imprimir) {
        this.id=id;
        this.buzonPregunta=pregunta;
        this.buzonCajaA=cajaA;
        this.buzonCajaB=cajaB;
        this.buzonAbandonoCaja=abandono;
        this.arrayBuzonComunicacion = array;
        this.imprimir = imprimir;

        this.tiempo="";
        this.caja="";
}
/**Cada comprador tendrá 5 fases a la hora de comprar (están recogidas dentro
 * del método. Lo recorreremos un total de 5 veces, ya que nos lo pide
 * expresamente el enunciado. Aquí se realiza la comunicación con el
 * controlador. Se recogeran sus mensajes en el buzón de la posición del
 * array que le corresponda, y se enviarán otros mensajes al controlador a
 * través de alguno de los buzones (cada buzón contiene un tipo de mensaje
 * específico).
 */
public void run() {

        for(int i = 0; i < 5; i++) {

                // PASO 1: REALIZA LA COMPRA
                try { Thread.sleep((long)(Math.random() * 1000)); }
                catch (InterruptedException e) { e.printStackTrace(); }


                // PASO 2: SOLICITAR CAJA
                // PRIMERO PREGUNTA POR LA CAJA QUE ESTÉ LIBRE
                buzonPregunta.send(id);
                mensajeAsignacion = (String) arrayBuzonComunicacion[id].receive();
                String[] partes = mensajeAsignacion.split(",");
                tiempo = partes[0];
                caja = partes[1];

                // LUEGO SOLICITA ENTRAR EN ESA CAJA
                if (caja.equals("A")) buzonCajaA.send(id);
                else buzonCajaB.send(id);
                mensajeConfirmacion= (String)arrayBuzonComunicacion[id].receive();


                // PASO 3: PAGAR EN CAJA
                try { Thread.sleep(Integer.parseInt(tiempo) * 1000); }
                catch (InterruptedException e) { e.printStackTrace(); }


                // PASO 4: LIBERAR CAJA
                buzonAbandonoCaja.send(id);
                mensajeConfirmacion= (String)arrayBuzonComunicacion[id].receive();


                // PASO 5: IMPRIMIR POR PANTALLA
                mensajeImprimir = (int) imprimir.receive();
                System.out.println("Persona " + (id+1) + " ha usado la caja " +
                                                                        caja);
                System.out.println("Tiempo de pago=" + tiempo);
                System.out.println("Thread.sleep(" + tiempo + ")");
                System.out.println("Persona " + (id+1) + " liberando la caja " +
                                                                        caja);
```

```
                        imprimir.send(mensajeImprimir);
                }
        }
}

/**La clase controlador se encarga de recoger los mensajes de los compradores
 * y enviarles otros devuelta. Definirá de manera aleatoria el tiempo que un
 * comprador estará en una caja (dependiendo del valor aleatorio se le envía
 * a la caja A o la B). Antes del constructor hemos decidido crear las variables
 * locales.
 */
public class HiloControlador extends Thread{
        private MailBox buzonPregunta;
        private MailBox buzonCajaA;
        private MailBox buzonCajaB;
        private MailBox buzonAbandonoCaja;
        private Mailbox[] arrayBuzonComunicacion;

        private Selector s;

        private int tiempo;
        private String devuelveAsignacion;
        private int idPreguntaCaja;
        private int idCajaA;
        private int idCajaB;

        private int idAbandono;
        private boolean cajaAOcupada;
        private boolean cajaBOcupada;

        /** El constructor recogerá todos los buzones creados por el programa
         * principal y los ligará a otros buzones.
         * @param pregunta - Recoge las veces que un comprador pregunta por una caja
         * @param A - Recoge los mensajes pertenecientes al buzón A
         * @param B - Recoge los mensajes pertenecientes al buzón B
         * @param abandono - Recoge los mensajes de abandono de una caja
         * @param array - Envía un mensaje a un comprador específico
         */
        public HiloControlador(Mailbox pregunta, Mailbox A, Mailbox B, Mailbox abandono,
                                                                Mailbox[] array) {
                this.buzonPregunta=pregunta;
                this.buzonCajaA=A;
                this.buzonCajaB=B;
                this.buzonAbandonoCaja=abandono;
                this.arrayBuzonComunicacion= array;
                this.tiempo=0;
                this.devuelveAsignacion="";
                this.cajaAOcupada=false;
                this.cajaBOcupada=false;

                s = new Selector();
                s.addSelectable(pregunta, false);
                s.addSelectable(A, false);
                s.addSelectable(B, false);
                s.addSelectable(abandono, false);
        }

        /**El controlador recibe todo tipo de mensajes por parte de los compradores,
         * a los que tendrá que responder una cosa u otra. El proceso se realiza
         * mediante un select (visto en clase de teoría). Proviene de una librería
         * creada exclusivamente para esta práctica. Dependiendo de unas condiciones
         * u otras, el controlador abre uno de los buzones y recorre el código de esa
         * opción.
         */
        public void run() {
                while(true) {
                        buzonPregunta.setGuardValue(true);
                        buzonCajaA.setGuardValue(!cajaAOcupada);
                        buzonCajaB.setGuardValue(!cajaBOcupada);
                        buzonAbandonoCaja.setGuardValue(true);
                        switch(s.selectOrBlock()){
                                case 1:
                                        idPreguntaCaja = (int) buzonPregunta.receive();
                                        tiempo = (int)(Math.random()*10) + 1;
                                        if (tiempo >= 5) {
                                                devuelveAsignacion= Integer.toString(tiempo)
                                                                        + "A";
```

```
                                        }
                                        else devuelveAsignacion= Integer.toString(tiempo) +
                                                                        ",B";
                                        arrayBuzonComunicacion[idPreguntaCaja].
                                                                send(devuelveAsignacion);
                                        break;
                                case 2:
                                        idCajaA = (int) buzonCajaA.receive();
                                        cajaAOcupada = true;
                                        arrayBuzonComunicacion[idCajaA].send("ok");
                                        break;
                                case 3:
                                        idCajaB = (int) buzonCajaB.receive();
                                        cajaBOcupada = true;
                                        arrayBuzonComunicacion[idCajaB].send("ok");
                                        break;
                                case 4:
                                        idAbandono = (int) buzonAbandonoCaja.receive();
                                        if (idCajaA == idAbandono) cajaAOcupada=false;
                                        else cajaBOcupada=false;
                                        arrayBuzonComunicacion[idAbandono].send("ok");
                                        break;
                        }
                }
        }
}

/**La clase principal del programa. Se encargará de inicializar todas
 * la variable de la pantalla (para activarla) y los buzones.
 */
public class Main {
        private static int activaImpresora;
        /**Crea los buzones (como habrá 30 cleintes debemos crear un array de
         * 30 posiciones, uno para cada cliente. Aquí recibiran sus mensajes),
         * el hilo controlador y los 30 hilos de los clientes/compradores.
         * @param args
         */
        public static void main(String[] args) {
                MailBox buzonPregunta = new MailBox();
                MailBox cajaA = new MailBox();
                MailBox cajaB = new MailBox();
                MailBox buzonRespuesta = new MailBox();
                MailBox imprimir = new MailBox();

                MailBox[] arrayBuzon = new MailBox[30];
                for(int i = 0; i < 30; i++) arrayBuzon[i] = new MailBox();

                imprimir.send(activaImpresora);

                HiloControlador controlador = new HiloControlador(buzonPregunta, cajaA,
                                                        cajaB, buzonRespuesta, arrayBuzon);

                controlador.start();

                HiloComprador[] compradores = new HiloComprador[30];

                for(int i=0;i<30;i++) {
                        compradores[i] = new HiloComprador(i, buzonPregunta, cajaA, cajaB,
                                                buzonRespuesta, arrayButon, imprimir);
                        compradores[i].start();
                }
        }

}
```