BOLETÍN DE PRÁCTICAS

Jose Antonio Lorencio Abril

PROGRAMACIÓN CONCURRENTE Y DISTRIBUIDA, 2019

Profesor: Pedro J. Fernández Ruiz

2º PCEO

Índice

1. INTRODUCCIÓN A LA PROGRAMACIÓN CONCURRENTE

1.1. Recursos no compartibles

La pantalla, para imprimir en ella.

1.2. Condiciones de sincronización

Un hilo no puede escribir en la pantalla hasta que esta esté libre, es decir, hasta que ningún otro hilo esté haciendo uso de la misma.

1.3. Pseudocódigo

```
Process type P
begin
  array A[10][10] de entero
  array contador[10] de entero
  para fila=1..10 hacer
    para columna=1..10 hacer
      j = random(0..9)
      contador[j]++
      A[fila][columna] = j
    finmientras
  finmientras
  wait(mutexPantalla)
    escribir(A)
    escribir(contador)
  signal(mutexPantalla)
end

main
var P1, P2, P3: P
initial(mutexPantalla, 1)
begin
  cobegin
    P1
    P2
    P3
  coend
end
```

1.4. Código Java

Clase Hilo

```
public class Hilo extends Thread {
  private String id;
  private Pantalla p;

  public Hilo(String idd, Pantalla pan) {
    id=idd;
    p=pan;
  }

  public void run() {
    for(int i=0; i<10; i++) {
      //Creamos la matriz
      int[][] M = new int[10][10];

      //Creamos un array en el que guardaremos la cantidad de ocurrencias de cada
    número generado aleatoriamente
      int[] cont = new int[10];
      for(int j=0; j<10; j++) {
        for(int k=0; k<10; k++) {
          int m= (int) (Math.random() * 10);
          cont[m]++;
          M[j][k]=m;
        }
      }

      //Imprimos por pantalla, en exclusión mutua
      p.imprime(id, M, cont);
    }
  }
}
```

Clase Pantalla

```
//Esta clase la usamos para imprimir la información requerida por pantalla en exclusión
    mutua
public class Pantalla {
  public synchronized void imprime(String id, int[][] M, int[] cont) {
    System.out.println("Hilo "+id);
    System.out.println();
    for(int i=0; i<10; i++) {
      for(int j=0; j<10; j++) {
        System.out.print(M[i][j]+" "); // Imprimimos el elemento (i,j)
      }
      System.out.println();
    }
    for(int i=0; i<10; i++) {
      System.out.println("Contador de "+ i +"="+cont[i]); // Imprimimos la cantidad de
    ocurrencias del número i
    }
    System.out.println("Terminando Hilo "+id);
    System.out.println();
  }
}
```

Clase Main

```
public class Main {

  public static void main(String[] args) {
    Pantalla p = new Pantalla();
    Hilo h1 = new Hilo("hilo 1", p);
    Hilo h2 = new Hilo("hilo 2", p);
    Hilo h3 = new Hilo("hilo 3", p);

    h1.start();
    h2.start();
    h3.start();
    try {
      h1.join();
      h2.join();
      h3.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

1.5. Cuestiones planteadas:

a) ¿Qué acciones pueden realizar los hilos concurrentemente?

Los hilos pueden generar las matrices concurrentemente, pues cada matriz es una variable local a cada hilo. También puede darse de forma concurrente el aumento del contador de un número, pues se hace, de nuevo, mediante una variable local de cada hilo.

Sin embargo, la impresión en pantalla debe ser sincronizada de forma no concurrente, pues en otro caso podríamos obtener los arrays impresos de forma entremezclada.

b) Las 10 impresiones que hace cada hilo, ¿son consecutivas o se entremezclan con las de los demás hilos? ¿Cuál de las opciones consideras que es la más correcta? Justifica la respuesta.

Se entremezclan, ya que esto permite que mientras un hilo escribe en pantalla, los otros dos puedan estar generando la siguiente matriz que deben imprimir. Considero que esta forma de proceder es la más correcta, pues si lo hiciésemos de la otra manera, la ejecución del programa sería casi lineal y no aprovecharíamos suficiente la concurrencia, pues la única operación que se realizaría de forma concurrente sería la generación de la primera matriz de cada hilo.

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

Podría llegar a ser un verdadero desastre, con matrices de distintos hilos entremezcladas, con los contadores de un hilo junto a la matriz de otro,... Aunque esto no tiene por qué ser así necesariamente,

podría darse el caso de que la salida saliese tal cual la queremos, aunque esto no podemos asegurarlo a priori. Por tanto, para poder asegurar que la salida será tal cual queremos que sea, debemos sincronizarla.

2. SEMÁFOROS.

2.1. Recursos no compartibles

La pantalla, para imprimir en ella.

2.2. Condiciones de sincronización

Los hilos deben esperar a que todos los demás hilos terminen de generar su palabra y la introduzcan en el array, pues, de otro modo, podrían dejarse palabras sin contar. Un hilo no puede escribir en la pantalla hasta que esta esté libre, es decir, hasta que ningún otro hilo esté haciendo uso de la misma.

2.3. Pseudocódigo

```
var sem, pantalla : semaforo
var palabras : array [1..30] de string

Process type P

begin
  string palabra = generarPalabraAleatoria

  palabras.añadir(palabra)

  signal(sem)
  wait(sem)

  entero cont = contarPalabras

  wait(mutexPantalla)
    escribir(palabras)
    escribir(cont)
  signal(mutexPantalla)
end

main
var Pi , i=1..30 : P
begin
initial(mutexPantalla, 1)
initial(sem, -29)
  cobegin
    Pi , i=1..30
  coend
end
```

2.4. Código Java

Clase Hilo

```
import java.util.LinkedList;

public class Hilo extends Thread {
  private int id;
  private String[] palabras; // Array con las palabras generadas por todos los hilos
  private LinkedList<String> words; // Lista con las palabras que empiezan por la misma
     letra que la palabra generada por este hilo

  public Hilo(int idd, String[] pals) {
    id=idd;
    palabras = pals;
    words = new LinkedList<String>();
  }

  public void run() {
    //Determinamos la longitud de la palabra
    int len = (int) (Math.random()*10+1);

    //Comenzamos con una palabra vacía
    String palabra="";
    for(int i=0; i<len; i++) {
      //y la vamos completando
      int n=(int) (Math.random()*26+'a');
      palabra+=(char) n;
      try {
        Thread.sleep(n);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }

    //La añadimos al array de palabras, podemos introducirlo sin exclusión mutua en la
    posición id, pues somos los únicos que accedemos a esta posición
    palabras[id]=palabra;

    //Sumamos uno al semáforo, mientras este sea negativo
    Main.sem.release();
    try {

    //Cuando sea positivo, podremos cogerlo y seguir con la ejecución del programa
      Main.sem.acquire();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    Main.sem.release(); // Para que no se queden bloqueados los demás hilos, al hacer
    el acquire

    //Contamos cuantas palabras comienzan con la misma letra que la nuestra, sin
    exclusión mutua, pues no modificamos la información del array
    for (String string : palabras) {
      if(string.startsWith(palabra.substring(0,1))) {
        words.add(string);
      }
    }

    //Pedimos la exclusión mutua para imprimir por pantalla
    try {
```

```
      Main.pantalla.acquire();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    System.out.println("Hilo "+id);
    System.out.println();
    System.out.print("Todas las palabras = ");
    for (String string : palabras) {
      System.out.print(string + " ");
    }
    System.out.println();
    System.out.print("Palabra hilo " + id + " = " + palabra);
    System.out.println();
    System.out.print("Palabras que empiezan con mi letra = ");
    for (String string : words) {
      System.out.print(string + " ");
    }
    System.out.println();
    System.out.println("Terminando Hilo "+id);
    System.out.println();

    //Una vez hemos imprimido toda la información necesaria, liberamos la pantalla
    Main.pantalla.release();
  }
}
```

Clase Main

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

public class Main {
  //Semáforo inicializado a -29, como está indicado en el pseudocódigo y explicado en
    la cuestión b)
  public static Semaphore sem = new Semaphore(-29);
  //Mutex para la exclusión mutua de la pantalla
  public static Semaphore pantalla = new Semaphore(1);

  public static void main(String[] args) {
    String[] palabras = new String[30];
    Hilo[] hilos = new Hilo[30];
    for(int i=0; i<30; i++) {
      hilos[i]=new Hilo(i, palabras);
    }

    for (Hilo hilo : hilos)
      hilo.start();

    try {
      for (Hilo hilo : hilos)
        hilo.join();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

2.5. Cuestiones planteadas

a) ¿Qué acciones pueden realizar simultáneamente los hilos?

Los hilos pueden generar su palabra concurrentemente, así como introducirla en el array. Tras terninar esta tarea, cada hilo debe esperar a que lo hayan hecho todos los demás, de forma que el array de palabras quede lleno, y puedan proceder a agrupar las palabras que comienzan por la misma letra que la suya. Esta última acción también puede ser llevada a cabo concurrentemente, para después turnarse, en exclusión mutua, el uso de la pantalla.

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

Uso dos semáforos. Sem, inicializado a -29, sirve para que los 30 hilos generen su palabra y la introduzcan al array de palabras, haciendo un signal(sem) cada vez que esto sucede. De este modo, hasta que no han terminado todos los hilos de generar palabras, ninguno de ellos procederá a la revisión de palabras con igual inicio que la suya. Pantalla, que es un semáforo binario inicializado a 1, simplemente hace de mutex, evitando que un recurso no compartible, como es la pantalla en este caso (se podrían imprimir los datos desordenados), sea compartido.

3. Monitores

3.1. Recursos no compartibles

El buffer, los productores y consumidores no pueden acceder a este simultáneamente. Tampoco pueden entrar ambos productores a la vez. Además, los productores y consumidores de un tipo, no deben poder acceder a elementos del otro tipo.

3.2. Condiciones de sincronización

Un consumidor de tipo i, solo puede acceder al buffer si no hay ningún productor de tipo i accediendo, y el buffer no puede estar vacío. Por el lado de un productor de tipo i, el buffer no puede estar lleno, ni estar el otro productor produciendo, tampoco puede haber consumidores del tipo i accediendo.

3.3. Pseudocódigo

```
monitor buffer;
var
  tam, cantidad1, cantidad2: integer
export insertar, extraer;
var
  puedoIntroducirElem, puedoSacarElemT1, puedoSacarElemT2 : condition
  recurso : contenedor[10] of item(1,2)

procedure insertar (ele : item, tipo : integer)
begin
  if(tam==10) then
  begin
    delay(puedoIntroducirElem);
  end
  tam++;
  if(tipo == 1)
    begin
      recurso.insertarTipo1(ele);
      cantidad1++;
      resume(puedoSacarElemT1);
    end
  else
    begin
      recurso.insertarTipo2(ele);
      cantidad2++;
      resume(puedoSacarElemT2);
    end
end;

procedure c1 (var it1 : item1)
begin
  if(cantidad1==0 ) then
  begin
    delay(puedoSacarElemT1);
  end
  it1=recurso.extraerTipo1();
  cantidad1--;
  resume (puedoIntroducirElem);
end;

```

```
procedure c2 (var it2 : item2)
begin
  if(cantidad2==0) then
  begin
    delay(puedoSacarElemT2);
  end;
  it2=recurso.extraerTipo2();
  cantidad2--;
  resume (puedoIntroducirElem);
end;

Process type Productor1
var b : buffer
para i=1..10 hacer b.p1(ele1);

Process type Productor2
var b : buffer
para i=1..10 hacer b.p2(ele2);

Process type Consumidor1
var b : buffer
para i=1..10 hacer b.c1(ele1);

Process type Consumidor2
var b : buffer
para i=1..10 hacer b.c2(ele2);

main
var
   array[3] prod1 : Productor1
   array[3] prod2 : Productor2
   array[3] cons1 : Consumidor1
   array[3] cons2 : Consumidor2
begin
  cobegin
    para i=1..3 hacer
    begin
      prod1;
      prod2;
      cons1;
      cons2;
    end
  coend
end
```

3.4. Código Java

Nota: los elementos de cada tipo son simplemente un entero, que será un 1 si el elemento es de tipo 1, o un 2 si es del tipo 2. Este hecho no afecta en absoluto al objetivo del ejercicio.

Clase Buffer

```
import java.util.concurrent.locks.*;
import java.util.LinkedList;

public class Buffer {
  ReentrantLock l = new ReentrantLock();
  Condition puedoIntroducirElem, puedoSacarElemT1, puedoSacarElemT2;
  //El buffer lo represento con dos listas, una para los elementos de cada tipo, que,
    como mucho podrán contener 10 elementos entre las dos
  LinkedList<Integer> tipo1;
  LinkedList<Integer> tipo2;
  int tam, cantidad1, cantidad2; //Tam es la cantidad de elementos en las dos listas,
    cantidadi indica cuántos elementos hay del tipo i

  public Buffer() {
    puedoIntroducirElem = l.newCondition();
    puedoSacarElemT1 = l.newCondition();
    puedoSacarElemT2 = l.newCondition();
    tipo1 = new LinkedList<Integer>();
    tipo2 = new LinkedList<Integer>();
    tam=cantidad1=cantidad2=0;
  }

  //Método que usarán los productores para insertar elementos
  public void insertar(int tipo) throws InterruptedException {
    l.lock();
    try {
      //Para poder introducir elementos, el buffer no puede estar lleno
      while(tam==10) {
        puedoIntroducirElem.await();
      }
      tam++;

      //Distinguimos el tipo de productor que quiere introducir elementos
      if(tipo==1) {
        tipo1.add(tipo);
        cantidad1++;
        //Imprimimos el estado del buffer
        info();
        //Avisamos a los consumidores del tipo 1 de que hay productos consumibles
        puedoSacarElemT1.signalAll();
      }
      else {
        tipo2.add(tipo);
        cantidad2++;
        info();
        //Avisamos a los consumidores del tipo 2 de que hay productos consumibles
        puedoSacarElemT2.signalAll();
      }
    } finally {
      l.unlock();
    }
  }
```

```
  //Método que usarán los consumidores para extraer elementos
  public void extraer(int tipo) throws InterruptedException {
    l.lock();
    //Distinguimos el tipo de consumidor que quiere acceder al buffer
    if(tipo==1) {
      try {
        //Si no hay elementos de tipo 1, los consumidores de tipo 1 quedan bloqueados
        while(cantidad1==0) {
          puedoSacarElemT1.await();
        }
        tipo1.remove();
        cantidad1--;
        tam--;
        puedoIntroducirElem.signal();
        info();
      } finally {
        l.unlock();
      }

    }
    else {
      try {
        //Lo mismo ocurre con los consumidores de tipo 2
        while(cantidad2==0) {
          puedoSacarElemT2.await();
        }
        tipo2.remove();
        cantidad2--;
        tam--;
        puedoIntroducirElem.signal();
        info();
      } finally {
        l.unlock();
      }
    }
  }

  //Método auxiliar que imprime el estado del buffer, para poder hacer seguimiento del
    mismo
  private void info() {
    for (Integer integer : tipo1) {
      System.out.print(integer + " ");
    }
    for (Integer integer : tipo2) {
      System.out.print(integer + " ");
    }
    System.out.println();
  }
}
```

Clase Consumidor

```
package Ejercicio3;

public class Consumidor extends Thread{
  private Buffer buffer;
  private int tipo;
  private int id;
```

```
  public Consumidor(Buffer b, int t, int idd) {
    buffer = b;
    tipo = t;
    id = idd;
  }

  public void run() {
    //Hacemos 20 extracciones
    for(int i=0; i<20; i++) {
      try {
        buffer.extraer(tipo);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Clase Productor

```
package Ejercicio3;

public class Productor extends Thread{
  private Buffer buffer;
  private int tipo;
  private int id;

  public Productor(Buffer b, int t, int idd) {
    buffer = b;
    tipo = t;
    id = idd;
  }

  public void run() {
    //Hacemos 20 inserciones
    for(int i=0; i<20; i++) {
      try {
        buffer.insertar(tipo);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Clase Main

```
package Ejercicio3;

public class Main {

  public static void main(String[] args) {
    Buffer buf = new Buffer();
    Consumidor[] c1 = new Consumidor[3];
    Consumidor[] c2 = new Consumidor[3];
    Productor[] p1 = new Productor[3];
    Productor[] p2 = new Productor[3];
    for(int i=0; i<3; i++) {
      c1[i]=new Consumidor(buf, 1, i);
      c2[i]=new Consumidor(buf, 2, i);
      p1[i]=new Productor(buf, 1, i);
      p2[i]=new Productor(buf, 2, i);
    }
    for(int i=0; i<3; i++) {
      c1[i].start();
      c2[i].start();
      p1[i].start();
      p2[i].start();
    }
    try {
      for(int i=0; i<3; i++) {
        c1[i].join();
        c2[i].join();
        p1[i].join();
        p2[i].join();
      }
    } catch(InterruptedException e) {
      e.printStackTrace();
    }
  }

}
```

3.5. Cuestiones planteadas

Contesta las siguientes cuestiones relativas al guión 3 de prácticas:

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

Sí. Como esta invocación se hace justo tras sumar la cantidad a compar, se asegura que la condición de sincronización de los hilos que decrementan se cumple, y los hilos que incrementan siempre pueden hacerlo. Por tanto, no existe la posibilidad de que todos los hilos queden bloqueados esperando a ser despertados.

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

Sí que se podría. Como hemos explicado en el ejercicio anterior, el hecho de hacer la suma antes del notify(), asegura que la condición de sincronización de cualquier hilo despertado se cumpla y pueda

proseguir su ejecución, imposibilitando el hecho de quedar todos los hilos bloqueados.

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

Debemos hacerlo con notifyAll(), pues tendremos 4 tipos de hilos (uno imprimirá cada palabra de la frase), de los que solo se verificará la condición de sincronización de uno de ellos. Si usásemos notify(), cabría la posibilidad de despertar un hilo que no puede imprimir aún, provocando un consecuente bloqueo. Por lo tanto, debemos despertar a todos los hilos, para que el correcto tome la exclusión mutua y el programa proporcione el resultado esperado.

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

Con el monitor reentrantlock, ya que estos permiten establecer varias condiciones de sincronización, y no solo una, como los monitores synchronized. Como estamos manejando 4 hilos, cada uno con una condición de sincronización (condiciones, además, excluyentes entre sí), es más eficiente poder despertar específicamente al hilo cuya condición sabemos que será cierta, y no despertarlos a todos, y que luchen por la exclusión mutua hasta que, en algún momento, entre el correspondiente hilo. Esto nos lo permiten los monitores reentrantlock.

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

He resuelto el ejercicio utilizando monitores reentrantlock, ya que necesitaba manejar 3 condiciones de sincronización diferentes. Como he explicado en d), esto se hace mucho mas eficientemente con este tipo de monitores. A la mejora de eficiencia, se le suma una mayor facilidad de manejo de las condiciones de sincronización.

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

Lo he realizado usando signal para los productores y signalAll para los consumidores.
La explicación es sencilla:

- Un consumidor, al extraer un elemento, despierta a un productor de su mismo tipo, usando signal. Uso signal porque tan solo un productor podrá acceder al buffer.
- Un productor, al insertar un elemento, despierta a consumidores de su mismo tipo, usando signal All. Uso signalAll porque podría ocurrir que varios consumidores pudieran acceder al buffer tras la inserción de un elemento, al haber elementos introducidos anteriormente. **Observación:** en este caso podría usar signal también, puesto que si hay elementos ya introducidos, y comenzasen a entrar consumidores, no se bloquearían.

4. Paso de mensajes

4.1. Recursos no compartibles

Las impresoras, pueden usarse las dos a la vez, pero una sola de ellas solo puede ser usada por un único hilo. Tampoco es compartible la pantalla, a la hora de escribir la información que nos pide el enunciado.

4.2. Condiciones de sincronización

Una impresora no puede estar siendo usada para que un hilo pueda imprimir en ella. La pantalla debe estar libre para poder escribir en ella, debe usarse en exclusión mutua.

4.3. Pseudocódigo

```
main
const nhilos=30
var
  solicitud : mailbox of integer
  liberacion : mailbox of Mensaje
  buzonOK : array 1..30 of mailbox of Mensaje
  h1,..,hnhilos : Hilo
  c : Controlador

begin
  cobegin
    c;
    for i=1..30 do
      hi;
    done
  coend
end

type Mensaje
var
  t : integer
  imp : char
  id : integer

Process type Hilo (id)
var
  id : integer
begin
  for i=1..5 do
    send (solicitud, id);
    receive (buzonOK[id], Msj);
    {Imprimir durante Msj.t segundos en la impresora Msj.id*}
    send (liberacion, Msj);
    {Escribe en pantalla la información requerida por el enunciado, en exclusión}
    {mutua, claro}
  done
end
```

```
45
Process type Controlador
var
  A, B : boolean {inicializado a true}
  colaA, colaB : queue of Mensaje
begin
  for i=1..nhilos*5*2 do
    select:
      when (true) => receive (solicitud, id);
        int t = random(1..10);
        if (A && colaA.empty && t>=5) then
          Msj = new Mensaje(t, "A", id);
          send (buzonOK[id], Msj);
          A=false;
        else if (B && colaB.empty && t<5) then
          Msj = new Mensaje(t, "B", id);
          send (buzonOK[id], Msj);
          B=false;
        else if(A && !colaA.empty && t>=5) then
          Msj = colaA.dequeue();
          send (buzonOK[Msj.id], Msj);
          A=false;
          Msj = new Mensaje(t, "A", id);
        else if(B && !colaB.empty && t<5) then
          Msj = colaB.dequeue();
          send (buzonOK[Msj.id], Msj);
          B=false;
          Msj = new Mensaje(t, "B", id);
        else if(!A && t>=5) then
          Msj = new Mensaje(t, "A", id);
          colaA.enqueue(Msj);
        else
          Msj = new Mensaje(t, "B", id);
          colaB.enqueue(Msj);
      or when (!A or !B) => receive (liberacion, Msj);
        if(Msj.imp == "A") then
          A=true;
          if(!colaA.empty) then
            Msj = colaA.dequeue();
            send (buzonOK[Msj.id], Msj);
            A=false;
        else if(Msj.imp == "B") then
          B=true;
          if(!colaB.empty) then
            Msj = colaB.dequeue();
            send (buzonOK[Msj.id], Msj);
            B=false;
  done
end
```

4.4. Código Java

Clase Mensaje

```
package Ejercicio4;

import java.io.Serializable;

//Clase usada para poder enviar la información necesaria para la tarea

public class Mensaje implements Serializable  {
  private int time; //Tiempo de impresión
  private char imp; //Impresora asignada
  private int id; //Id del hilo

  public Mensaje(int t, char i, int idd) {
    time=t;
    imp=i;
    id=idd;
  }

  public int getTime() {
    return time;
  }

  public char getImp() {
    return imp;
  }

  public int getId() {
    return id;
  }

}
```

Clase Controlador

```
package Ejercicio4;

import messagepassing.*;
import java.util.LinkedList;

public class Controlador extends Thread{
  private boolean A,B; //Indican si cada impresora está libre (true) o no (false)
  private LinkedList<Mensaje> colaA, colaB; //Colas de espera de cada impresora
  private MailBox[] buzonOK; //Array que contiene los buzones de recepción de cada hilo
     para poder transmitirles la información relativa a la impresora asignada y el
    tiempo de impresión estimado
  private MailBox solicitud, liberacion; //Buzones de recepción del controlador,
    solicitud para pedir impresora, liberacion para liberarla
  private Selector s; //Para la estructura Select

  public Controlador(MailBox sol, MailBox l, MailBox[] b) {
    A=B=true;
    colaA=new LinkedList<Mensaje>();
    colaB=new LinkedList<Mensaje>();
    buzonOK=b;
```

```
    solicitud=sol;
    liberacion=l;
    s = new Selector();
    s.addSelectable(solicitud, false);
    s.addSelectable(liberacion, false);
  }

  public void run() {
    for(int i=0;i<300;i++) { //Hacemos nhilos*5*2 loops
      solicitud.setGuardValue(true); //Siempre podemos enviar una petición
      liberacion.setGuardValue(!A || !B); //Podremos liberar impresora si alguna está
    ocupada
      switch(s.selectOrBlock()) {
      //Recibimos una solicitud de impresión
      case 1: Mensaje m = (Mensaje) solicitud.receive();

          // Calculamos el tiempo de impresión, y dependiendo de este
          // hacemos una u otra acción
          int t = (int)(Math.random()*10+1);
          if(A && colaA.isEmpty() && t>=5) {/ /Directamente a la impresora rápida
            Mensaje msj = new Mensaje(t, 'A', m.getId());
            A=false;
            buzonOK[m.getId()].send(msj);
          }
          else if(B && colaB.isEmpty() && t<5) { //Directamente a la impresora lenta
            Mensaje msj = new Mensaje(t, 'B', m.getId());
            B=false;
            buzonOK[m.getId()].send(msj);
          }
          else if(A && !colaA.isEmpty() && t>=5) { //A la cola de la rápida, también
    sacamos primer elemento
            colaA.addFirst(new Mensaje(t, 'A', m.getId()));
            Mensaje msj = colaA.getLast();
            colaA.removeLast();
            A=false;
            buzonOK[msj.getId()].send(msj);
          }
          else if(B && !colaB.isEmpty() && t<5) { //A la cola de la lenta, sacamos
    primer elemento
            colaB.addFirst(new Mensaje(t, 'B', m.getId()));
            Mensaje msj = colaB.getLast();
            colaB.removeLast();
            B=false;
            buzonOK[msj.getId()].send(msj);
          }
          else if(!A && t>=5) { //A la cola de la rápida
            colaA.addFirst(new Mensaje(t, 'A', m.getId()));
          }
          else { //A la cola de la lenta
            colaB.addFirst(new Mensaje(t, 'B', m.getId()));
          }
          break;

      //Recibimos información de liberación de una impresora
      case 2: Mensaje men = (Mensaje) liberacion.receive();
          //Distinguimos si debemos liberar la A o la B
          if(men.getImp()=='A') {
            A=true;
            //Si hay trabajos en espera, le asignamos, en orden FIFO, la impresora
    liberada
            if(!colaA.isEmpty()) {
```

```
                Mensaje msj = colaA.getLast();
                colaA.removeLast();
                buzonOK[msj.getId()].send(msj);
                A=false;
              }
            }
            else {
              B=true;
              if(!colaB.isEmpty()) {
                Mensaje msj = colaB.getLast();
                colaB.removeLast();
                buzonOK[msj.getId()].send(msj);
                B=false;
              }
            }
            break;
        }
      }
    }
}
```

Clase Hilo

```
package Ejercicio4;

import messagepassing.*;

public class Hilo extends Thread {
  private int id;
  private MailBox buzonOK, solicitud, liberacion, pantalla; //buzonOK sirve para
    recibir el mensaje de asignación de impresora y notificación de tiempo de impresión
    ; pantalla sirve para efectuar la exclusión mutua de la pantalla

  public Hilo(int idd, MailBox b, MailBox s, MailBox l, MailBox p) {
    id=idd;
    buzonOK=b;
    solicitud=s;
    liberacion=l;
    pantalla=p;
  }

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

      //Enviamos la solicitud de impresora
      solicitud.send(new Mensaje(0, '0', id));

      //Esperamos a recibir la información con el tiempo de impresión y la impresora
    asignada
      Mensaje msj=(Mensaje)buzonOK.receive();
      try {
        Thread.sleep(msj.getTime()); //Tardamos msj.getTime() en imprimir
      } catch (InterruptedException e) {
        e.printStackTrace();
      }

      //Informamos de que hemos acabado la impresión del trabajo
      liberacion.send(msj);
```

```
      //Buscamos conseguir la exclusión mutua para imprimir la información requerida
      Object token = pantalla.receive();
      System.out.println("Hilo "+id+" ha usado la impresora "+msj.getImp()+" para
    realizar la impresión número "+(i+1));
      System.out.println("Tiempo de uso = "+msj.getTime());
      System.out.println();

      pantalla.send(token);

    }
  }
}
```

Clase Main

```
package Ejercicio4;

import messagepassing.*;

public class Main {

  public static void main(String[] args) {
    //Este MailBox actúa como mutex para la impresión en pantalla, lanzamos el token
    para que puede ser pedido en un momento dado
    MailBox pantalla = new MailBox();
    pantalla.send("token");

    MailBox solicitud = new MailBox();
    MailBox liberacion = new MailBox();
    MailBox[] buzonOK = new MailBox[30];
    Hilo[] hilos = new Hilo[30];
    for(int i=0; i<30; i++) {
      buzonOK[i] = new MailBox();
      hilos[i] = new Hilo(i, buzonOK[i], solicitud, liberacion, pantalla);
    }
    Controlador c = new Controlador(solicitud, liberacion, buzonOK);

    c.start();
    for (Hilo hilo : hilos) {
      hilo.start();
    }

    try {
      for (Hilo hilo : hilos) {
        hilo.join();
      }
      c.join();
    } catch(InterruptedException e) {
      e.printStackTrace();
    }
  }

}
```

4.5. Cuestiones planteadas

a) ¿Se pueden usar simultáneamente las dos impresoras?

Sí, puesto que cada impresora imprimirá el trabajo de un único proceso en un momento dado, no tiene sentido que no puedan usarse las dos a la vez, imprimiendo trabajos de distintos procesos. Esto solo ralentizaría la impresión.

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

La he resuelto usando un MailBox llamado pantalla como mutex, tal y como se explica en el boletín 4 de prácticas. Se hace circular un token entre todos los hilos, de tal forma que para poder imprimir por pantalla, necesitas adquirir ese token. Para esto, todos los hilos comparten el mencionado MailBox, y esperan en un receive a que les llegue el token, imprimen y, tras esto, hacen un send, enviando el token, devolviéndolo a la circulación.